

This page displays common pseudocode functions shared by many pages

## Pseudocodes

### Library pseudocode for aarch32/at/AArch32.AT

```
// AArch32.AT()
// =====
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, TranslationStage
            stage = stage_in;
SecurityState ss;
Regime regime;
boolean eae;

// ATS1Hx instructions
if el == EL2 then
    regime = Regime EL2;
    eae = TRUE;
    ss = SS NonSecure;

// ATS1Cxx instructions
elsif stage == TranslationStage\_1 || (stage == TranslationStage\_12
    stage = TranslationStage\_1;
    ss = SecurityStateAtEL(PSTATE.EL);
    regime = if ss == SS Secure && ELUsingAArch32(EL3) then Regime
    eae = TTBCR.EAE == '1';

// ATS12NSOxx instructions
else
    regime = Regime EL10;
    eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR_EAE;
    ss = SS NonSecure;

AddressDescriptor addrdesc;
SDFType sdftype;
boolean aligned = TRUE;
bit supersection = '0';

accdesc = CreateAccDescAT(ss, el, ataccess);

// Prepare fault fields in case a fault is detected
fault = NoFault(accdesc);

if eae then
    (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, vaddress);
else
    (fault, addrdesc, sdftype) = AArch32.S1TranslateSD(fault, regime,
                                              accdesc);
    supersection = if sdftype == SDFType Supersection then '1' else
```

```

// ATS12NSOxx instructions
if stage == TranslationStage_12 && fault.statuscode == Fault_None
    (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned);

if fault.statuscode != Fault_None then
    // Take exception on External abort or when a fault occurs on translation
    if IsExternalAbort(fault) || (PSTATE.EL == EL1 && EL2Enabled())
        PAR = bits(64) UNKNOWN;
        AArch32.Abort(vaddress, fault);

addrdesc.fault = fault;

if (eae || (stage == TranslationStage_12 && (HCR.VM == '1' || HCR.DM == '1')) ||
    (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL1))
    AArch32.EncodePARLD(addrdesc, ss);
else
    AArch32.EncodePARSD(addrdesc, supersection, ss);
return;

```

## Library pseudocode for aarch32/at/AArch32.EncodePARLD

```

// AArch32.EncodePARLD()
// =====
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

if !IsFault(addrdesc) then
    bit ns;
    if ss == SS_NonSecure then
        ns = bit UNKNOWN;
    elseif addrdesc.paddress.paspace == PAS_Secure then
        ns = '0';
    else
        ns = '1';
    PAR.F      = '0';
    PAR.SH    = ReportedPARShareability(PAREncodeShareability(addrdesc));
    PAR.NS    = ns;
    PAR<10>  = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    PAR.LPAE   = '1';
    PAR.PA    = addrdesc.paddress.address<39:12>;
    PAR.ATTR  = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
else
    PAR.F      = '1';
    PAR.FST    = AArch32.PARFaultStatusID(addrdesc.fault);
    PAR.S2WLK  = if addrdesc.fault.s2fs1walk then '1' else '0';
    PAR.FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
    PAR.LPAE   = '1';
    PAR<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
return;

```

## Library pseudocode for aarch32/at/AArch32.EncodePARSD

```

// AArch32.EncodePARSD()
// =====
// Returns 32-bit format PAR on address translation instruction.

```

```

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, Se
AddressDescriptor addrdesc = addrdesc_in;
if !IsFault(addrdesc) then
    if (addrdesc.memattrs.memtype == MemType_Device ||
        (addrdesc.memattrs.inner.attrs == MemAttr_NC &&
         addrdesc.memattrs.outer.attrs == MemAttr_NC)) then
        addrdesc.memattrs.shareability = Shareability_OSH;
    bit ns;
    if ss == SS_NonSecure then
        ns = bit UNKNOWN;
    elsif addrdesc.paddress.paspace == PAS_Secure then
        ns = '0';
    else
        ns = '1';
    bits(2) sh = if addrdesc.memattrs.shareability != Shareability
PAR.F      = '0';
PAR.SS     = supersection;
PAR.Outer  = AArch32.ReportedOuterAttrs(AArch32.PAROuterAttrs(a
PAR.Inner  = AArch32.ReportedInnerAttrs(AArch32.PARIInnerAttrs(a
PAR.SH     = ReportedPARShareability(sh);
PAR<8>    = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
PAR.NS     = ns;
PAR.NOS    = if addrdesc.memattrs.shareability == Shareability
PAR.LPAE   = '0';
PAR.PA     = addrdesc.paddress.address<39:12>;
else
    PAR.F      = '1';
    PAR.FST   = AArch32.PARFaultStatusSD(addrdesc.fault);
    PAR.LPAE   = '0';
    PAR<31:16> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
return;

```

### Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

```

// AArch32.PARFaultStatusLD()
// =====
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(6) syndrome;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome<1:0> = if fault.level == 1 then '01' else '10';
        syndrome<5:2> = '1111';
    else
        syndrome = EncodeLDFSC(fault.statuscode, fault.level);
    return syndrome;

```

### Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```

// AArch32.PARFaultStatusSD()
// =====
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)

```

```

        bits(6) syndrome;

        syndrome<5> = if IsExternalAbort(fault) then fault.extflag else '0';
        syndrome<4:0> = EncodeSDFSC(fault.statuscode, fault.level);
        return syndrome;
    
```

### Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

```

// AArch32.PARInnerAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;

    if memattrs.memtype == MemType\_Device then
        if memattrs.device == DeviceType\_nGnRNE then
            result = '001'; // Non-cacheable
        elseif memattrs.device == DeviceType\_nGnRE then
            result = '011'; // Non-cacheable
    else
        MemAttrHints inner = memattrs.inner;
        if inner.attrs == MemAttr\_NC then
            result = '000'; // Non-cacheable
        elseif inner.attrs == MemAttr\_WB && inner.hints<0> == '1' then
            result = '101'; // Write-Back, Write-Allocate
        elseif inner.attrs == MemAttr\_WT then
            result = '110'; // Write-Through
        elseif inner.attrs == MemAttr\_WB && inner.hints<0> == '0' then
            result = '111'; // Write-Back, no Write-Allocate
    return result;

```

### Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

```

// AArch32.PAROuterAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;

    if memattrs.memtype == MemType\_Device then
        result = bits(2) UNKNOWN;
    else
        MemAttrHints outer = memattrs.outer;
        if outer.attrs == MemAttr\_NC then
            result = '00'; // Non-cacheable
        elseif outer.attrs == MemAttr\_WB && outer.hints<0> == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elseif outer.attrs == MemAttr\_WT && outer.hints<0> == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elseif outer.attrs == MemAttr\_WB && outer.hints<0> == '0' then
            result = '11'; // Write-Back, no Write-Allocate
    return result;

```

## Library pseudocode for aarch32/at/AArch32.ReportedInnerAttrs

```
// AArch32.ReportedInnerAttrs()
// =====
// The value returned in this field can be the resulting attribute, as
// implementation choices and any applicable configuration bits, instead
// in the translation table descriptor.

bits(3) AArch32.ReportedInnerAttrs(bits(3) attrs);
```

## Library pseudocode for aarch32/at/AArch32.ReportedOuterAttrs

```
// AArch32.ReportedOuterAttrs()
// =====
// The value returned in this field can be the resulting attribute, as
// implementation choices and any applicable configuration bits, instead
// in the translation table descriptor.

bits(2) AArch32.ReportedOuterAttrs(bits(2) attrs);
```

## Library pseudocode for aarch32/dc/AArch32.DC

```
// AArch32.DC()
// =====
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
           CacheRecord cache;

    cache.acctype      = AccessType_DC;
    cache.cacheop     = cacheop;
    cache.opscope     = opscope;
    cache.cachetype   = CacheType_Data;
    cache.security    = SecurityStateAtEL(PSTATE.EL);

    if opscope == CacheOpScope_SetWay then
        cache.shareability = Shareability_NSH;
        (cache.setnum, cache.waynum, cache.level) = DecodeSW(ZeroExtend
                                                CacheType);

        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled()
            (!ELUsingAArch32(EL2) && (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != 1)
             || (ELUsingAArch32(EL2) && (HCR.SWIO == '1' || HCR.<DC,VM> != 1))
            cache.cacheop = CacheOp_CleanInvalidate;
        CACHE_OP(cache);
        return;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid         = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;
```

```

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid          = ASID[];;
else
    cache.is_asid_valid = FALSE;

need_translate = DCInstNeedsTranslation(opscope);
vaddress = regval;

size = 0;           // by default no watchpoint address
if cacheop == CacheOp_Invalidate then
    size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Wa
    assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is pow
    vaddress = Align(regval, size);

cache.translated = need_translate;
cache.vaddress   = ZeroExtend(vaddress, 64);

if need_translate then
    boolean aligned = TRUE;
    AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch32.TranslateAddress(vaddre
    if IsFault(memaddrdesc) then
        AArch32.Abort(regval, memaddrdesc.fault);

    cache.paddress = memaddrdesc.paddress;
    if opscode == CacheOpScope_PoC then
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability_NSH;
else
    cache.shareability = Shareability_UNKNOWN;
    cache.paddress     = FullAddress UNKNOWN;

if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled
    (!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') ||
    (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))) then
    cache.cacheop = CacheOp_CleanInvalidate;

CACHE_OP(cache);
return;

```

## Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```

// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

boolean match;
if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
    // Each bit position in this string corresponds to a bit in DBG
    match_word = Zeros(32);

    ss = CurrentSecurityState();
    if vaddress<31:5> == ExcVectorBase()<31:5> then
        if HaveEL(EL3) && ss == SS_NonSecure then

```

```

        match_word<UInt>(vaddress<4:2>) + 24> = '1'; // Non-
    else
        match_word<UInt>(vaddress<4:2>) + 0> = '1'; // Secu

    if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVE
        ss == SS_Secure) then
        match_word<UInt>(vaddress<4:2>) + 8> = '1'; // Moni

    // Mask out bits not corresponding to vectors.
    bits(32) mask;
    if !HaveEL(EL3) then
        mask = '00000000':'00000000':'00000000':'11011110'; // DBGW
    elseif !ELUsingAArch32(EL3) then
        mask = '11011110':'00000000':'00000000':'11011110'; // DBGW
    else
        mask = '11011110':'00000000':'11011100':'11011110';

    match_word = match_word AND DBGVCR AND mask;
    match = !IsZero(match_word);

    // Check for UNPREDICTABLE case - match on Prefetch Abort and D
    if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PST
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAF

    if !IsZero(vaddress<1:0>) && match then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHAL
    else
        match = FALSE;

    return match;

```

### Library pseudocode for aarch32/debug/authentication/ AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivileged
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE;
    return DBGEN == Signal_High && SPIDEN == Signal_High;

```

### Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```

// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,
                                            integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    enabled      = DBGBCR[n].E == '1';
    isbreakpnt = TRUE;

```

```

linked      = DBGBCR[n].BT IN {'0x01'};
linked_to   = FALSE;
linked_n    = UInt(DBGBCR[n].LBN);

state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGE
                                 linked, linked_n, isbreakpt, acc
(value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vac

if size == 4 then                                // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint m
    // second halfword of an instruction, but not the address of th
    // CONSTRAINED UNPREDICTABLE whether or not this breakpoint ge
    // event.
    (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress

if !value_match && match_i then
    value_match = ConstrainUnpredictableBool(Unpredictable_BPMA

if value_mismatch && !mismatch_i then
    value_mismatch = ConstrainUnpredictableBool(Unpredictable_E

if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
    // The above notwithstanding, if DBGBCR[n].BAS == '1111', then
    // UNPREDICTABLE whether or not a Breakpoint debug event is ger
    // at the address DBGBVR[n]+2.
    if value_match then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMA

if !value_mismatch then
    value_mismatch = ConstrainUnpredictableBool(Unpredictable_E

match      = value_match && state_match && enabled;
mismatch  = value_mismatch && state_match && enabled;

return (match, mismatch);

```

## Library pseudocode for aarch32/debug/breakpoint/ AArch32.BreakpointValueMatch

```

// AArch32.BreakpointValueMatch()
// =====
// The first result is whether an Address Match or Context breakpoint is
// instruction at "address". The second result is whether an Address Mi
// programmed on the instruction, that is, whether the instruction shou

(boolean, boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked
    // matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linkin
integer n = n_in;
Constraint c;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTAB
    // no match or the breakpoint is mapped to another UNKNOWN implemen
if n >= NumBreakpointsImplemented() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImpleme

```

```

    Unpredictable\_BPNOTIMPL\(\)
    assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
    if c == Constraint\_DISABLED then return (FALSE, FALSE);

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if DBGBCR[n].E == '0' then return (FALSE, FALSE);

    dbgtype = DBGBCR[n].BT;

    (c, dbgtype) = AArch32.ReservedBreakpointType(n, dbgtype);
    if c == Constraint\_DISABLED then return (FALSE, FALSE);
    // Otherwise the dbgtype value returned by AArch32.ReservedBreakpoi

    // Determine what to compare against.
    match_addr      = (dbgtype IN {'0x0x'});
    mismatch        = (dbgtype IN {'010x'});
    match_vmid      = (dbgtype IN {'10xx'});
    match_cid1      = (dbgtype IN {'xx1x'});
    match_cid2      = (dbgtype IN {'11xx'});
    linking_enabled = (dbgtype IN {'xxx1'});

    // If called from StateMatch, is is CONSTRAINED UNPREDICTABLE if th
    // breakpoint is not programmed with linking enabled.
    if linked_to && !linking_enabled then
        if !ConstrainUnpredictableBool(Unpredictable\_BLINKINGDISABLED)
            return (FALSE, FALSE);

    // If called from BreakpointMatch return FALSE for Linked context I
    if !linked_to && linking_enabled && !match_addr then
        return (FALSE, FALSE);

    boolean bvr_match = FALSE;
    boolean bxvr_match = FALSE;

    // Do the comparison.
    if match_addr then
        integer byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                                // "vaddress" is half

        boolean byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        bvr_match = (vaddress<31:2> == DBGBVR[n]<31:2>) && byte_select_<

    elsif match_cid1 then
        bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>)

    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;

        if ELUsingAArch32\(EL2\) then
            vmid = ZeroExtend(VTTBR.VMID, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        elsif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0'
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBXVR[n]<15:0>;

```

```

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == CONTEXTIDR_EL1<31:0>);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32(DBGBXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

        bvr_match_valid = (match_addr || match_cid1);
        bxvr_match_valid = (match_vmid || match_cid2);

        match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bxvr_match);
    end;

    return (match && !mismatch, !match && mismatch);
end;

```

### Library pseudocode for aarch32/debug/breakpoint/ AArch32.ReservedBreakpointType

```

// AArch32.ReservedBreakpointType()
// =====
// Checks if the given DBGBCR<n>.BT value is reserved and will generate
// behavior, otherwise returns Constraint_NONE.

(Constraint, bits(4)) AArch32.ReservedBreakpointType(integer n, bits(4)
    bits(4) bt      = bt_in;
    boolean reserved = FALSE;
    context_aware = n >= (NumBreakpointsImplemented() - NumContextAware());

    // Address mismatch
    if bt IN {'010x'} && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;

    // Context matching
    if !(bt IN {'0x0x'}) && !context_aware then
        reserved = TRUE;

    // EL2 extension
    if bt IN {'1xxx'} && !HaveEL(EL2) then
        reserved = TRUE;

    // Context matching
    if (bt IN {'011x', '11xx'} && !IsFeatureImplemented(FEAT_VHE) &&
        !IsFeatureImplemented(FEAT_Debugv8p2)) then
        reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable RESBPTYPE, 4);
        assert c IN {Constraint DISABLED, Constraint UNKNOWN};
        if c == Constraint DISABLED then
            return (c, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be
        // valid for the current context
    end;

    return (Constraint NONE, bt);
end;

```

### Library pseudocode for aarch32/debug/breakpoint/AArch32.StateMatch

```

// AArch32.StateMatch()
// =====

```

```

// Determine whether a breakpoint or watchpoint is enabled in the current context

boolean AArch32.StateMatch(bits(2) ssc_in, bit hmc_in, bits(2) pxc_in,
                           integer linked_n_in, boolean isbreakpt, AccDesc accdesc);

// "ssc_in", "hmc_in", "pxc_in" are the control fields from the DBGBCR[n]
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type
// "linked_n_in" is the linked breakpoint number from the DBGBCR[n]
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "accdesc" describes the properties of the access being matched.
bit hmc           = hmc_in;
bits(2) ssc       = ssc_in;
bits(2) pxc       = pxc_in;
boolean linked    = linked_in;
integer linked_n = linked_n_in;

// If parameters are set to a reserved type, behaves as either disabled or
Constraint c;
// SSCE value discarded as there is no SSCE bit in AArch32.
(c, ssc, -, hmc, pxc) = CheckValidStateMatch(ssc, '0', hmc, pxc, isbreakpt);
if c == Constraint DISABLED then return FALSE;
// Otherwise the hmc,ssc,pxc values are either valid or the values
// CheckValidStateMatch are valid.

p12_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) ||
p11_match = pxc<0> == '1';
p10_match = pxc<1> == '1';
ssu_match = isbreakpt && hmc == '0' && pxc == '00' && ssc != '11';

boolean priv_match;
if ssu_match then
    priv_match = PSTATE.M IN {M32\_User, M32\_Svc, M32\_System};
else
    case accdesc.el of
        when EL3 priv_match = p11_match;                                // EL3 and EL1 are
        when EL2 priv_match = p12_match;
        when EL1 priv_match = p11_match;
        when EL0 priv_match = p10_match;

// Security state match
boolean ss_match;
case ssc of
    when '00' ss_match = TRUE;
    when '01' ss_match = accdesc.ss == SS\_NonSecure;
    when '10' ss_match = accdesc.ss == SS\_Secure;
    when '11' ss_match = (hmc == '1' || accdesc.ss == SS\_Secure);

boolean linked_match = FALSE;

if linked then
    // "linked_n" must be an enabled context-aware breakpoint unit.
    // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE
    // this gives no match, gives a match without linking, or linked
    // UNKNOWN breakpoint that is context-aware.
    if !IsContextMatchingBreakpoint(linked_n) then
        (first_ctx_cmp, last_ctx_cmp) = ContextMatchingBreakpointRange(c, linked_n);
        (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
        assert c IN {Constraint DISABLED, Constraint NONE, Constraint CONSTRAINED, Constraint UNPREDICTABLE, Constraint UNKNOWN};

    if !IsContextAware(linked_n) then
        linked_n = linked_n_in;
    else
        linked_n = linked_n_in;

```

```

        case c of
            when Constraint_DISABLED return FALSE;           // Disable
            when Constraint_NONE      linked = FALSE;       // No link
            // Otherwise ConstrainUnpredictableInteger returned a constraint

    vaddress = bits(32) UNKNOWN;
    linked_to = TRUE;
    (linked_match, -) = AArch32.BreakpointValueMatch(linked_n, vaddr);

    return priv_match && ss_match && (!linked || linked_match);

```

### Library pseudocode for aarch32/debug/enables/ AArch32.GenerateDebugExceptions

```

// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);

```

### Library pseudocode for aarch32/debug/enables/ AArch32.GenerateDebugExceptionsFrom

```

// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState ss)
    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        mask = '0';          // No PSTATE.D in AArch32 state
        return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, ss);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        assert from_el != EL2;      // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b0
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
            if from_el == EL0 then enabled = enabled || SDER.SUIDEN == '1';
        else
            enabled = from_el != EL2;
    return enabled;

```

## Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```
// AArch32.ClearEventCounters()
// =====
// Zero all the event counters.

AArch32.ClearEventCounters()
    if HaveAArch64\(\) then
        // Force the counter to be cleared as a 64-bit counter.
        AArch64.ClearEventCounters\(\);
        return;

    integer counters = AArch32.GetNumEventCountersAccessible\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR[idx] = Zeros(32);
```

## Library pseudocode for aarch32/debug/pmu/ AArch32.GetNumEventCountersAccessible

```
// AArch32.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current EL.

integer AArch32.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters\(\);
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled\(\) then
        n = UInt(if !ELUsingAArch32\(EL2\) then MDCR_EL2.HPMN else HDCR.HPMN);
        if n > total_counters || (!IsFeatureImplemented(FEAT_HPMN0) &&
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable\_PMUEVE));
    else
        n = total_counters;

    return n;
```

## Library pseudocode for aarch32/debug/pmu/AArch32.IncrementCycleCounter

```
// AArch32.IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

AArch32.IncrementCycleCounter()
    if !CountPMUEvents\(CYCLE\_COUNTER\_ID\) then return;
    bit d = PMCR.D; // Check divide-by-64
    bit lc = PMCR.LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles\(\) then return;

    integer old_value = UInt(PMCCNTR);
    integer new_value = old_value + 1;
    PMCCNTR = new_value<63:0>;
```

```

constant integer ovflw = if lc == '1' then 64 else 32;

if old_value<64:ovflw> != new_value<64:ovflw> then
    PMOVSET.C = '1';
    PMOVSRC.C = '1';

return;

```

## Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

```

// AArch32.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
    if HaveAArch64\(\) then
        // Force the counter to be incremented as a 64-bit counter.
        AArch64.IncrementEventCounter(idx, increment);
        return;

    // In this model, event counters in an AArch32-only implementation
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is impl
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    PMEVCNTR[idx] = new_value<31:0>;
    constant integer ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSET<idx> = '1';
        PMOVSRC<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters\(\) then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

    return;

```

## Library pseudocode for aarch32/debug/pmu/AArch32.PMUCycle

```

// AArch32.PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
    if HaveAArch64\(\) then
        AArch64.PMUCycle();
        return;

    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

```

```

integer counters = GetNumEventCounters\(\);
if counters != 0 then
    for idx = 0 to counters - 1
        if CountPMUEvents\(idx\) then
            integer accumulated = PMUEventAccumulator[idx];
            AArch32.IncrementEventCounter\(idx, accumulated\);
            PMUEventAccumulator[idx] = 0;
AArch32.IncrementCycleCounter\(\);
CheckForPMUOverflow\(\);

```

### **Library pseudocode for aarch32/debug/pmu/AArch32.PMUSwIncrement**

```

// AArch32.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch32.GetNumEventCountersAccessible\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

```

### **Library pseudocode for aarch32/debug/takeexceptiondbg/ AArch32.EnterHypModeInDebugState**

```

// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord except)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState\(\) == SS_NonSecure && ELU

    AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR_curr[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in A
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                                // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = HSCTRL.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
UpdateEDSCRFields();

EndOfInstruction();

```

## Library pseudocode for aarch32/debug/takeexceptiondbg/ AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// =====
// Take an exception in Debug state to a mode other than Monitor and Hy

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in A
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                                // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0' then PSTATE.
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFIELDS();                         // Update EDSCR process
    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/ AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in A
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                                // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
```

```

        PSTATE.PAN = '0';
    elsif SCLTR.SPAN == '0' then
        PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFIELDS(); // Update EDSCR process
EndOfInstruction();

```

## Library pseudocode for aarch32/debug/watchpoint/ AArch32.WatchpointByteMatch

```

// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
constant integer dbgtop = 31;
constant integer cmpbottom = if DBGWVR[n]<2> == '1' then 2 else 3;
integer bottom = cmpbottom;
constant integer select = UInt(vaddress<cmpbottom-1:0>);
byte_select_match = (DBGWCR[n].BAS<select> != '0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is a nonzero value and DBGWCR[n].BAS is not set
// DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is
// UNPREDICTABLE.
if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
    byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASK);
else
    LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1)); MSB = (DBGWCR[n].BAS - 1);
    if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASK);
        bottom = 3; // For the whole double word
    end

// If the address mask is set to a reserved value, the behavior is
if mask > 0 && mask <= 2 then
    Constraint c;
    (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_WPMASK);
    assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_Undefined} case c of
        when Constraint_DISABLED return FALSE; // Disabled
        when Constraint_NONE mask = 0; // No masking
        // Otherwise the value returned by ConstrainUnpredictableInteger
        // is used as the mask value

constant integer cmpmsb = dbgtop;
constant integer cmplsb = if mask > bottom then mask else bottom;
constant integer bottombit = bottom;
boolean WVR_match = (vaddress<cmpmsb:cmplsb> == DBGWVR[n]<cmpmsb:cmplsb>);
if mask > bottom then
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is
    if WVR_match && !IsZero(DBGWVR[n]<cmplsb-1:bottombit>) then
        WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASK);

return (WVR_match && byte_select_match);

```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer s
                                 AccessDescriptor accdesc)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumWatchpointsImplemented\(\);

    boolean enabled      = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;
    linked_n = UInt(DBGWCR_EL1[n].LBN);
    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].LSC,
                                         linked, linked_n, isbreakpnt, accdesc);

    boolean ls_match;
    case DBGWCR[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType.WRITE;
        when '11' ls_match = TRUE;

    boolean value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress[byte], accdesc);

    return value_match && state_match && ls_match && enabled;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\);

    if !route_to_aarch64 && EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage\(fault\)
                            (IsFeatureImplemented\(FEAT\_RAS\) && HCR_EL2.TGE == '1') ||
                            IsExternalAbort\(fault\)) || IsDebugException\(fault\) && MDCR_EL2.TDE == '1';

    if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
        route_to_aarch64 = EffectiveEA\(\) == '1' && IsExternalAbort\(fault\) || IsDataAbort\(fault\) == '1';

    if route_to_aarch64 then
        AArch64.Abort\(ZeroExtend\)\(vaddress, 64\), fault;
    elseif fault.accessdesc.acctype == AccessType.IFETCH then
        AArch32.TakePrefetchAbortException\(vaddress, fault\);
    else
        AArch32.TakeDataAbortException\(vaddress, fault\);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort exceptions
// taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord
                                      bits(32) vaddress, bits(2) target
                                      except = ExceptionSyndrome(exceptype);

except.syndrome = AArch32.FaultSyndrome(exceptype, fault);
except.vaddress = ZeroExtend(vaddress, 64);

if IPAVValid(fault) then
    except.ipavvalid = TRUE;
    except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1'
    except.ipaddress = ZeroExtend(fault.ipaddress.address, 56);
else
    except.ipavvalid = FALSE;

return except;
```

## Library pseudocode for aarch32/exceptions/aborts/ AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()
    bits(32) pc = ThisInstrAddr(32);

    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> ==
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmer

        AccessDescriptor accdesc = CreateAccDescIFetch();
        FaultRecord fault = NoFault(accdesc);
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32.Abort(pc, fault);
```

## Library pseudocode for aarch32/exceptions/aborts/ AArch32.CommonFaultStatus

```
// AArch32.CommonFaultStatus()
// =====
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.

bits(32) AArch32.CommonFaultStatus(FaultRecord fault, boolean long_form
                                     bits(32) target = Zeros(32);
                                     if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
                                         ErrorState errstate = AArch32.PEErrorState(fault);
                                         target<15:14> = AArch32.EncodeAsyncErrorSyndrome(errstate);
```

```

if IsExternalAbort(fault) then target<12> = fault.extflag; // /
target<9> = if long_format then '1' else '0';
if long_format then
    target<5:0>      = EncodeLDFSC(fault.statuscode, fault.level);
else
    target<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
return target;

```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```

// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp
AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault,
                        bits(32) vaddress)
    boolean long_format;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = ((TTBCR_S.EAE == '1') ||
                       (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL1) ||
                           (fault.secondstage && (boolean IMPLEMENTATION_DEFINED
                           "Report abort using Long Format"))));
    else
        long_format = TTBCR.EAE == '1';
    bits(32) syndrome = AArch32.CommonFaultStatus(fault, long_format);

    // bits of syndrome that are not common to I and D side
    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_WnR}
        syndrome<13> = '1'; // CM
        syndrome<11> = '1'; // WnR
    else
        syndrome<11> = if fault.write then '1' else '0'; // WnR

    if !long_format then
        syndrome<7:4> = fault.domain; // Domain

    if fault.accessdesc.acctype == AccessType_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance")
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;

```

## Library pseudocode for aarch32/exceptions/aborts/ AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault)
    // The encoding used in the IFSR can be Long-descriptor format or S
    // Normally, the current translation table format determines the fo
    // Non-secure state to Monitor mode, the IFSR uses the Long-descrip
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is s
long_format = FALSE;
if route_to_monitor && CurrentSecurityState\(\) != SS\_Secure then
    long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.E
else
    long_format = TTBCR.EAE == '1';

bits(32) fsr = AArch32.CommonFaultStatus(fault, long_format);

if route_to_monitor then
    IFSR_S = fsr;
    IFAR_S = vaddress;
else
    IFSR = fsr;
    IFAR = vaddress;

return;
```

## Library pseudocode for aarch32/exceptions/aborts/ AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && EffectiveEA\(\) == '1' && IsExternal
    route_to_hyp = (EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (IsFeatureImplemented(FEAT_RAS) && HCR2.TEA == '1' ||
                      IsExternalAbort(fault)) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset,
    elseif PSTATE.EL == EL2 || route_to_hyp then
        except = AArch32.AbortSyndrome(Exception\_DataAbort, fault, vadc
```

```

        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, ve
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32 Abort, preferred_exception_return, lr_offset

```

### **Library pseudocode for aarch32/exceptions/aborts/ AArch32.TakePrefetchAbortException**

```

// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault
    route_to_monitor = HaveEL(EL3) && EffectiveEA() == '1' && IsExternal

    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (IsFeatureImplemented(FEAT_RAS) && HCR2.TEA == '1' ||
                      IsExternalAbort(fault)) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

ExceptionRecord except;
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x0C;
lr_offset = 4;

if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
if route_to_monitor then
    AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset,
elsif PSTATE.EL == EL2 || route_to_hyp then
    if fault.statuscode == Fault Alignment then                                // PC A
        except = ExceptionSyndrome(Exception PCAlignment);
        except.vaddress = ThisInstrAddr(64);
    else
        except = AArch32.AbortSyndrome(Exception InstructionAbort);
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, ve
    else
        AArch32.EnterHypMode(except, preferred_exception_return, 0x
else
    AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32 Abort, preferred_exception_return, lr_offset

```

### **Library pseudocode for aarch32/exceptions/async/ AArch32.TakePhysicalFIQException**

```

// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

```

```

if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' &&

if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.FIQ == '1';

if route_to_aarch64 then AArch64.TakePhysicalFIQException();
route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                (HCR.TGE == '1' || HCR.FMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x1C;
lr_offset = 4;
if route_to_monitor then
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset,
elseif PSTATE.EL == EL2 || route_to_hyp then
    except = ExceptionSyndrome(Exception_FIQ);
    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset);

```

### **Library pseudocode for aarch32/exceptions/async/ AArch32.TakePhysicalIRQException**

```

// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' &&
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset,
    elseif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset);

```

### **Library pseudocode for aarch32/exceptions/async/ AArch32.TakePhysicalSErrorException**

```

// AArch32.TakePhysicalSErrorException()
// =====

AArch32.TakePhysicalSErrorException(boolean implicit_esb)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_E
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(implicit_esb);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    bits(2) target_el;
    if route_to_monitor then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_hyp then
        target_el = EL2;
    else
        target_el = EL1;

    FaultRecord fault = GetPendingPhysicalSError();
    vaddress = bits(32) UNKNOWN;
    except = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    case target_el of
        when EL3
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset);
        when EL2
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return);
            else
                AArch32.EnterHypMode(except, preferred_exception_return);
        when EL1
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset);
    otherwise
        Unreachable();

```

## Library pseudocode for aarch32/exceptions/async/ AArch32.TakeVirtualFIQException

```

// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()

```

```

assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and F
    assert HCR.TGE == '0' && HCR.FMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtual

bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x1C;
lr_offset = 4;

AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, ve

```

### **Library pseudocode for aarch32/exceptions/async/ AArch32.TakeVirtualIRQException**

```

// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtual

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, ve

```

### **Library pseudocode for aarch32/exceptions/async/ AArch32.TakeVirtualSErrorException**

```

// AArch32.TakeVirtualSErrorException()
// =====

AArch32.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 a
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtual
    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

```

```

vaddress = bits(32) UNKNOWN;
parity = FALSE;
Fault fault = Fault_AsyncExternal;
integer level = integer UNKNOWN;
bits(32) fsr = Zeros(32);
if IsFeatureImplemented(FEAT_RAS) then
    if ELUsingAArch32(EL2) then
        fsr<15:14> = VDFSR.AET;
        fsr<12> = VDFSR.ExT;
    else
        fsr<15:14> = VSESR_EL2.AET;
        fsr<12> = VSESR_EL2.ExT;
else
    fsr<12> = bit IMPLEMENTATION_DEFINED "Virtual External abort ty
if TTBCR.EAE == '1' then // Long-descriptor format
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault, level);
else // Short-descriptor format
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault, level);
DFSR = fsr;
DFAR = bits(32) UNKNOWN;
ClearPendingVirtualSError();
AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset,

```

### Library pseudocode for aarch32/exceptions/debug/ AArch32.SoftwareBreakpoint

```

// AArch32.SoftwareBreakpoint()
// =====
AArch32.SoftwareBreakpoint(bits(16) immediate)

if (EL2Enabled() && !ELUsingAArch32(EL2) &&
    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL2)
    AArch64.SoftwareBreakpoint(immediate);

accdesc = CreateAccDescIFetch();
fault = NoFault(accdesc);
vaddress = bits(32) UNKNOWN;

fault.statuscode = Fault_Debug;
fault.debugmoe = DebugException_BKPT;

AArch32.Abort(vaddress, fault);

```

### Library pseudocode for aarch32/exceptions/debug/DebugException

```

constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '1010';

```

## Library pseudocode for aarch32/exceptions/exceptions/ AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =====
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

if PSTATE.EL == EL1 && EL2Enabled() then
    tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
    tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

    if ((tid0 == '1' && reg == '0000') ||
        (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then
        if ELUsingAArch32(EL2) then
            AArch32.SystemAccessTrap(M32_Hyp, 0x8);
        else
            AArch64.AArch32SystemAccessTrap(EL2, 0x8);
```

## Library pseudocode for aarch32/exceptions/exceptions/ AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported
// in an interrupt frame. The instruction length field is only valid for
// instructions that trap.

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception_Uncategorized                      ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap                            ec = 0x01;
        when Exception_CP15RTTTrap                      ec = 0x03;
        when Exception_CP15RRTTrap                      ec = 0x04;
        when Exception_CP14RTTTrap                      ec = 0x05;
        when Exception_CP14DTTTrap                      ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap              ec = 0x07;
        when Exception_FPIDTrap                         ec = 0x08;
        when Exception_PACTrap                          ec = 0x09;
        when Exception_TSTARTAccessTrap                ec = 0x1B;
        when Exception_GPC                             ec = 0x1E;
        when Exception_CP14RRTTrap                      ec = 0x0C;
        when Exception_BranchTarget                    ec = 0x0D;
        when Exception_IllegalState                    ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall                  ec = 0x11;
        when Exception_HypervisorCall                  ec = 0x12;
        when Exception_MonitorCall                     ec = 0x13;
        when Exception_InstructionAbort                ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAAlignment                   ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort                       ec = 0x24;
        when Exception_NV2DataAbort                   ec = 0x25;
        when Exception_FPTrappedException              ec = 0x28;
        when Exception_PMU                            ec = 0x3D;
```

```

        otherwise Unreachable\(\);

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength\(\) == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);

```

### **Library pseudocode for aarch32/exceptions/exceptions/ AArch32.GeneralExceptionsToAArch64**

```

// AArch32.GeneralExceptionsToAArch64()
// =====
// Returns TRUE if exceptions normally routed to EL1 are being handled
// level using AArch64, because either EL1 is using AArch64 or TGE is
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32\(EL1\)) ||
            (EL2Enabled\(\) && !ELUsingAArch32\(EL2\) && HCR_EL2.TGE == '1')

```

### **Library pseudocode for aarch32/exceptions/exceptions/ AArch32.ReportHypEntry**

```

// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord except)

Exception exceptype = except.exceptype;

(ec,il) = AArch32.ExceptionClass\(exceptype\);
iss = except.syndrome;
iss2 = except.syndrome2;

// IL is not valid for Data Abort exceptions without valid instruction
if ec IN {0x24,0x25} && iss<24> == '0' then
    il = '1';

HSR = ec<5:0>:il:iss;

if exceptype IN {Exception\_InstructionAbort, Exception\_PCAAlignment}
    HIFAR = except.vaddress<31:0>;
    HDFAR = bits(32) UNKNOWN;
elseif exceptype == Exception\_DataAbort then
    HIFAR = bits(32) UNKNOWN;
    HDFAR = except.vaddress<31:0>;

if except.ipavvalid then
    HPFAR<31:4> = except.ipaddress<39:12>;
else
    HPFAR<31:4> = bits(28) UNKNOWN;

```

```
    return;
```

## Library pseudocode for aarch32/exceptions/exceptions/ AArch32.ResetControlRegisters

```
// AArch32.ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have
// reset values to those values.

AArch32.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                                // Secure state
    elseif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness
    // SCTRLR values produced by the above call to ResetControlRegisters
    PSTATE.<A,I,F> = '111';                      // All asynchronous exceptions
    PSTATE.IT = '00000000';                        // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTRLR.TE;                     // Instruction set: TE=0:A32, T=1:TE
        PSTATE.E = HSCTRLR.EE;                     // Endianness: EE=0: little-endian
    else
        PSTATE.T = SCTRLR.TE;                      // Instruction set: TE=0:A32, T=1:TE
        PSTATE.E = SCTRLR.EE;                      // Endianness: EE=0: little-endian
    PSTATE.IL = '0';                                // Clear Illegal Execution state

    // All registers, bits and fields not reset by the above pseudocode
    // below are UNKNOWN bitstrings after reset. In particular, the register
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);
```

```

bits(32) rv;                                // IMPLEMENTATION_DEFINED reset vector

if HaveEL(EL3) then
    if MVBAR<0> == '1' then                // Reset vector in MVBAR
        rv = MVBAR<31:1>:'0';
    else
        rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
else
    rv = RVBAR<31:1>:'0';

// The reset vector must be correctly aligned
assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

boolean branch_conditional = FALSE;
EDPRSR.R = '0';                            // Leaving Reset State.
BranchTo(rv, BranchType_RESET, branch_conditional);

```

### Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```

// ExcVectorBase()
// =====

bits(32) ExcVectorBase()
if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
    return Ones(16):Zeros(16);
else
    return VBAR<31:5>:Zeros(5);

```

### Library pseudocode for aarch32/exceptions/ieeefp/ AArch32.FPTrappedException

```

// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
if AArch32.GeneralExceptionsToAArch64() then
    is_ase = FALSE;
    element = 0;
    AArch64.FPTrappedException(is_ase, accumulated_exceptions);
FPEXC.DEX      = '1';
FPEXC.TFV      = '1';
FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;
FPEXC<10:8>   = '111';                         // I
                                                       // V
AArch32.TakeUndefInstrException();

```

### Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```

// AArch32.CallHypervisor()
// =====
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
assert HaveEL(EL2);

```

```

if !ELUsingAArch32(EL2) then
    AArch64.CallHypervisor(immediate);
else
    AArch32.TakeHVCException(immediate);

```

### **Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor**

```

// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);

```

### **Library pseudocode for aarch32/exceptions/syscalls/ AArch32.TakeHVCException**

```

// AArch32.TakeHVCException()
// =====

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);

```

### **Library pseudocode for aarch32/exceptions/syscalls/ AArch32.TakeSMCEException**

```

// AArch32.TakeSMCEException()
// =====

AArch32.TakeSMCEException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    HSAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

```

```

    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_

```

### **Library pseudocode for aarch32/exceptions/syscalls/ AArch32.TakeSVCException**

```

// AArch32.TakeSVCEception()
// =====

AArch32.TakeSVCEception(bits(16) immediate)

AArch32.ITAdvance\(\);
SSAdvance\(\);
route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

bits(32) preferred_exception_return = NextInstrAddr(32);
vect_offset = 0x08;
lr_offset = 0;

if PSTATE.EL == EL2 || route_to_hyp then
    except = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_
    else
        AArch32.EnterHypMode(except, preferred_exception_return, 0x_
else
    AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset)

```

### **Library pseudocode for aarch32/exceptions/takeexception/ AArch32.EnterHypMode**

```

// AArch32.EnterHypMode()
// =====
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord except, bits(32) preferred_exception_
    integer vect_offset)
SynchronizeContext\(\);
assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELU

if Halted() then
    AArch32.EnterHypModeInDebugState(except);
    return;
bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
if !(except.exceptiontype IN {Exception IRQ, Exception FIQ}) then
    AArch32.ReportHypEntry(except);
AArch32.WriteMode(M32_Hyp);
SPSR_curr[] = spsr;
ELR_hyp = preferred_exception_return;
PSTATE.T = HSCTRLR.TE;                                // PSTATE.J is RES0
PSTATE.SS = '0';
if !HaveEL(EL3) || SCR_curr[].EA == '0' then PSTATE.A = '1';
if !HaveEL(EL3) || SCR_curr[].IRQ == '0' then PSTATE.I = '1';
if !HaveEL(EL3) || SCR_curr[].FIQ == '0' then PSTATE.F = '1';
PSTATE.E = HSCTRLR.EE;

```

```

PSTATE.IL = '0';
PSTATE.IT = '00000000';
if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = HSCLR.DSSBS;
boolean branch_conditional = FALSE;
BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_
CheckExceptionCatch(TRUE); // Check for debug even
EndOfInstruction();

```

### Library pseudocode for aarch32/exceptions/takeexception/ AArch32.EnterMode

```

// AArch32.EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if Halted() then
        AArch32.EnterModeInDebugState(target_mode);
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elseif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0' then PSTATE.IL = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase())<31:5>:vect_offset<4:0>, BranchType_EXCEPTION;
    CheckExceptionCatch(TRUE); // Check for debug even
    EndOfInstruction();

```

### Library pseudocode for aarch32/exceptions/takeexception/ AArch32.EnterMonitorMode

```

// AArch32.EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

```

```

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer 1
                        integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32.EnterMonitorModeInDebugState();
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                                     // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_
    CheckExceptionCatch(TRUE);                                // Check for debug even
    EndOfInstruction();

```

## Library pseudocode for aarch32/exceptions/traps/ AArch32.CheckAdvSIMDOrFPEnabled

```

// AArch32.CheckAdvSIMDOrFPEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check_in, boolean advsimd)
    boolean fpexc_check = fpexc_check_in;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        // When executing at EL0 using AArch32, if EL1 is using AArch64
        // FPEXC.EN is 1. This includes when EL2 is using AArch64 and e
        // Security state, HCR_EL2.TGE is 1, and the Effective value of
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState()
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then AArch32.Undefined();

```

```

// Check if access disabled in CPACR
boolean disabled;
case cpacr_cp10 of
    when '00' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '10' disabled = ConstrainUnpredictableBool(Unpredi
    when '11' disabled = FALSE;
if disabled then AArch32.Undefined();

// If required, check FPEXC enabled bit.
if (fpexc_check && PSTATE.EL == EL0 && EL2Enabled() && !ELUsinc
    HCR_EL2.TGE == '1') then
// When executing at EL0 using AArch32, if EL2 is using AA
// current Security state, HCR_EL2.TGE is 1, and the Effecti
// then it is IMPLEMENTATION_DEFINED whether the Effective v
// or the value of FPEXC32_EL2.EN.
fpexc_check = (boolean IMPLEMENTATION_DEFINED
                "Use FPEXC32_EL2.EN value when {TGE,RW} == {0,0}

if fpexc_check && FPEXC.EN == '0' then
    AArch32.Undefined();

AArch32.CheckFPAdvSIMDTrap(advsimd);      // Also check against H

```

### Library pseudocode for aarch32/exceptions/traps/ AArch32.CheckFPAdvSIMDTrap

```

// AArch32.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if (HaveEL(EL3) && !ELUsingAArch32(EL3) &&
            CPTR_EL3.TFP == '1' && EL3SDDUndefPriority()) then
            UNDEFINED;

        ss = CurrentSecurityState();
        if HaveEL(EL2) && ss != SS_Secure then
            hcptr_tase = HCPTER.TASE;
            hcptr_cp10 = HCPTER.TCP10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
            if NSACR.cp10 == '0' then hcptr_cp10 = '1';

        // Check if access disabled in HCPTER
        if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
            except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTra
            except.syndrome<24:20> = ConditionSyndrome();

            if advsimd then
                except.syndrome<5> = '1';
            else

```

```

        except.syndrome<5> = '0';
        except.syndrome<3:0> = '1010';                                // coproc fie

        if PSTATE.EL == EL2 then
            AArch32.TakeUndefInstrException(except);
        else
            AArch32.TakeHypTrapException(except);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then
                if EL3SDDUndef() then
                    UNDEFINED;
                else
                    AArch64.AdvSIMDFPAccessTrap(EL3);

```

### Library pseudocode for aarch32/exceptions/traps/ AArch32.CheckForSMCUndefOrTrap

```

// AArch32.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = EL2Enabled() && PSTATE.EL == EL1 && HCR.TSC == ...
        if route_to_hyp then
            except = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(except);

```

### Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```

// AArch32.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if IsFeatureImplemented(FEAT_FGT) then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFG...
                           (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3))

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_retu

```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, WFxType wfxtYPE)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, wfxtYPE);
        return;

    boolean is_wfe = wfxtYPE == WFxType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCLR.nTWE else SCLR.nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) &&
            AArch64.WFxTrap(wfxtYPE, target_el);

        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            except = ExceptionSyndrome(Exception_WFxTrap);
            except.syndrome<24:20> = ConditionSyndrome();

        case wfxtYPE of
            when WFxType_WFI
                except.syndrome<0> = '0';
            when WFxType_WFE
                except.syndrome<0> = '1';

            AArch32.TakeHypTrapException(except);
        else
            AArch32.TakeUndefInstrException();
    
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEEnabled

```
// AArch32.CheckITEEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCLR.ITD else SCLR.
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;
```

```

accdesc = CreateAccDescIFetch\(\);
aligned = TRUE;
// Otherwise whether the IT block is allowed depends on hw1 of
next_instr = AArch32.MemSingle\[NextInstrAddr\]\(32\), 2, accdesc, a

if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxxx', '1010
                  '01001xxxxxxxxxxxx', '010001xxx1111xxx', '0100
                  // It is IMPLEMENTATION DEFINED whether the Undefined Instru
                  // taken on the IT instruction or the next instruction. Thi
                  // the pseudocode, which always takes the exception on the
                  // also does not take into account cases where the next ins
UNDEFINED;

return;

```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```

// AArch32.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if
AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64\(\) then
        AArch64.CheckIllegalState\(\);
    elseif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled\(\) && HCR.TGE == '1'
        bits(32) preferred_exception_return = ThisInstrAddr(32);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            except = ExceptionSyndrome\(Exception\_IllegalState\);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return);
            else
                AArch32.EnterHypMode(except, preferred_exception_return);
        else
            AArch32.TakeUndefInstrException\(\);

```

## Library pseudocode for aarch32/exceptions/traps/ AArch32.CheckSETENDEnabled

```

// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTRL.SED;
    else
        setend_disabled = (if ELUsingAArch32\(EL1\) then SCTLR.SED else S
    if setend_disabled == '1' then
        UNDEFINED;

return;

```

## Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// =====
// Trapped System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFfromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) < 16;

    if target_el == EL2 then
        except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(except);
    else
        AArch32.TakeUndefInstrException();
```

## Library pseudocode for aarch32/exceptions/traps/ AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4    except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6    except = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7    except = ExceptionSyndrome(Exception_AdvSIMDFPAccess);
        when 0x8    except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptiontype == Exception_Uncategorized then
        return except;
    elsif except.exceptiontype IN {Exception_FPIDTrap, Exception_CP14RTTTrap,
                                  Exception_CP15RTTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;           // CRn, Reg in case of VMRS
        iss<8:5>    = instr<15:12>;           // Rt
        iss<9>       = '0';                   // RES0

        if except.exceptiontype != Exception_FPIDTrap then      // When trap
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;           // opc1
            iss<4:1>   = instr<3:0>;           // CRm
        else // VMRS Access
            iss<19:17> = '000';                 // opc2 - Hardcoded for VMRS
            iss<16:14> = '111';                 // opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';                // CRm - Hardcoded for VMRS
```

```

        elseif except.exceptiontype IN {Exception_CP14RRTTrap, Exception_AdvSIMD
                                         Exception_CP15RRTTrap} then
            // Trapped MRRC/MCRR, VMRS/VMSR
            iss<19:16> = instr<7:4>;                                // opc1
            iss<13:10> = instr<19:16>;                                // Rt2
            iss<8:5> = instr<15:12>;                                // Rt
            iss<4:1> = instr<3:0>;                                // CRM
        elseif except.exceptiontype == Exception_CP14DTTrap then
            // Trapped LDC/STC
            iss<19:12> = instr<7:0>;                                // imm8
            iss<4> = instr<23>;                                    // U
            iss<2:1> = instr<24,21>;                                // P,W
            if instr<19:16> == '1111' then // Rn==15, LDC(Literal address)
                iss<8:5> = bits(4) UNKNOWN;
                iss<3> = '1';
            iss<0> = instr<20>;                                    // Direction

        except.syndrome<24:20> = ConditionSyndrome();
        except.syndrome<19:0> = iss;

        return except;
    
```

## Library pseudocode for aarch32/exceptions/traps/ AArch32.TakeHypTrapException

```

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(except);

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord except)
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELU

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x14;

    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);

```

## Library pseudocode for aarch32/exceptions/traps/ AArch32.TakeMonitorTrapException

```

// AArch32.TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;

```

```

    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_

```

### **Library pseudocode for aarch32/exceptions/traps/ AArch32.TakeUndefInstrException**

```

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException()
except = ExceptionSyndrome(Exception_Uncategorized);
AArch32.TakeUndefInstrException(except);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord except)

route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
bits(32) preferred_exception_return = ThisInstrAddr(32);
vect_offset = 0x04;
lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
elseif route_to_hyp then
    AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
else
    AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset);

```

### **Library pseudocode for aarch32/exceptions/traps/AArch32.Undefined**

```

// AArch32.Undefined()
// =====

AArch32.Undefined()

if AArch32.GeneralExceptionsToAArch64() then AArch64.Undefined();
AArch32.TakeUndefInstrException();

```

### **Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid**

```

// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnW_
            return level == 2;

```

```

        otherwise
            return FALSE;

```

## Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```

// AArch32.FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address
// exceptions taken to AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(Exception exceptype, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros(25);

    boolean d_side = exceptype == Exception_DataAbort;
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        ErrorState errstate = AArch32.PEErrorState(fault);
        iss<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fs1walk &&
            (!IsExternalSyncAbort(fault) ||
             (!IsFeatureImplemented(FEAT_RAS) && fault.accessdesc.acctype
              boolean IMPLEMENTATION_DEFINED "ISV on second stage translation"))
            iss<24:14> = LSInstructionSyndrome();

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_ZI}
            iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_ZI}
            iss<6> = '1';
        elseif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_ExclusiveAccessFlag}
            iss<6> = bit UNKNOWN;
        elseif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            iss<6> = bit UNKNOWN;
        else
            iss<6> = if fault.write then '1' else '0';

        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss);

```

## Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```

// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types

bits(5) EncodeSDFSC(Fault statuscode, integer level)

    bits(5) result;
    case statuscode of
        when Fault_AccessFlag
            assert level IN {1,2};

```

```

        result = if level == 1 then '00011' else '00110';
when Fault Alignment
        result = '00001';
when Fault Permission
        assert level IN {1,2};
        result = if level == 1 then '01101' else '01111';
when Fault Domain
        assert level IN {1,2};
        result = if level == 1 then '01001' else '01011';
when Fault Translation
        assert level IN {1,2};
        result = if level == 1 then '00101' else '00111';
when Fault SyncExternal
        result = '01000';
when Fault SyncExternalOnWalk
        assert level IN {1,2};
        result = if level == 1 then '01100' else '01110';
when Fault SyncParity
        result = '11001';
when Fault SyncParityOnWalk
        assert level IN {1,2};
        result = if level == 1 then '11100' else '11110';
when Fault AsyncParity
        result = '11000';
when Fault AsyncExternal
        result = '10110';
when Fault Debug
        result = '00010';
when Fault TLBConflict
        result = '10000';
when Fault Lockdown
        result = '10100';      // IMPLEMENTATION DEFINED
when Fault Exclusive
        result = '10101';      // IMPLEMENTATION DEFINED
when Fault ICacheMaint
        result = '00100';
otherwise
    Unreachable();

return result;

```

### Library pseudocode for aarch32/functions/common/A32ExpandImm

```

// A32ExpandImm()
// =====
bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the
(imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

return imm32;

```

### Library pseudocode for aarch32/functions/common/A32ExpandImm\_C

```

// A32ExpandImm_C()
// =====

```

```

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRTtype_ROR, 2*UInt(im
    return (imm32, carry_out);

```

### Library pseudocode for aarch32/functions/common/DecodeImmShift

```

// DecodeImmShift()
// =====

(SRTtype, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    SRTtype shift_t;
    integer shift_n;
    case srtype of
        when '00'
            shift_t = SRTtype_LSL; shift_n = UInt(imm5);
        when '01'
            shift_t = SRTtype_LSR; shift_n = if imm5 == '00000' then 32
        when '10'
            shift_t = SRTtype_ASR; shift_n = if imm5 == '00000' then 32
        when '11'
            if imm5 == '00000' then
                shift_t = SRTtype_RRX; shift_n = 1;
            else
                shift_t = SRTtype_ROR; shift_n = UInt(imm5);

    return (shift_t, shift_n);

```

### Library pseudocode for aarch32/functions/common/DecodeRegShift

```

// DecodeRegShift()
// =====

SRTtype DecodeRegShift(bits(2) srtype)
    SRTtype shift_t;
    case srtype of
        when '00' shift_t = SRTtype_LSL;
        when '01' shift_t = SRTtype_LSR;
        when '10' shift_t = SRTtype_ASR;
        when '11' shift_t = SRTtype_ROR;
    return shift_t;

```

### Library pseudocode for aarch32/functions/common/RRX

```

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;

```

## Library pseudocode for aarch32/functions/common/RRX\_C

```
// RRX_C()
// =====

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/SRTYPE

```
// SRTYPE
// =====

enumeration SRTYPE {SRTYPE_LSL, SRTYPE_LSR, SRTYPE_ASR, SRTYPE_ROR, SRTYPE_RRX}
```

## Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

bits(N) Shift(bits(N) value, SRTYPE srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/Shift\_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRTYPE srtype, integer amount, bit carry_in)
    assert !(srtype == SRTYPE_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRTYPE_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRTYPE_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRTYPE_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRTYPE_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRTYPE_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm\_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
            carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));
    end
    return (imm32, carry_out);
```

## Library pseudocode for aarch32/functions/common/VBitOps

```
// VBitOps
// =====

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

## Library pseudocode for aarch32/functions/common/VCGEType

```
// VCGEType
// =====

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

## Library pseudocode for aarch32/functions/common/VCGTtype

```

// VCGTtype
// =====

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

```

### Library pseudocode for aarch32/functions/common/VFPNegMul

```

// VFPNegMul
// =====

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMU};

```

### Library pseudocode for aarch32/functions/coproc/ AArch32.CheckCP15InstrCoarseTraps

```

// AArch32.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
        (EL2Enabled() && !ELUsingAArch32(EL2))) then
            AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
                        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((!(major IN {4,14}) && HSTR<major> == '1') ||
            (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CRn");
                UNDEFINED;
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);

```

### Library pseudocode for aarch32/functions/exclusive/ AArch32.ExclusiveMonitorsPass

```

// AArch32.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all
// associated with the virtual address region of size bytes starting at
// The immediately following memory write must be to the same addresses

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts
// before or after the check on the local Exclusives monitor. As a result
// of this, the check may be performed at any time between the start of the
// write and the end of the write.

```

```

// of the local monitor can occur on some implementations even if the m
// access would give an memory abort.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
    boolean aligned = IsAligned(address, size);

    if !aligned then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if !AArch32.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, s

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size,
        ClearExclusiveLocal(ProcessorID()));

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;

```

### **Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA**

```

// AArch32.IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address
// of the local monitor can occur on some implementations even if the m
// access would give an memory abort.

boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size)

```

### **Library pseudocode for aarch32/functions/exclusive/ AArch32.MarkExclusiveVA**

```

// AArch32.MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region
// starting at address for processorid.

AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size)

```

## Library pseudocode for aarch32/functions/exclusive/ AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the address
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel,
    boolean aligned = IsAligned(address, size);

    if !aligned then
        AArch32.Abort(address, AlignmentFault(accdesc));

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD
    // is enabled

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// =====

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
```

```
AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);  
// Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access  
return;
```

### Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()  
// ======  
  
CheckCryptoEnabled32()  
    CheckAdvSIMDEnabled();  
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted  
    return;
```

### Library pseudocode for aarch32/functions/float/CheckVFPEnabled

```
// CheckVFPEnabled()  
// ======  
  
CheckVFPEnabled(boolean include_fpexc_check)  
    advsimd = FALSE;  
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);  
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access  
    return;
```

### Library pseudocode for aarch32/functions/float/FPHalvedSub

```

// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType Infinity); inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero); zero2 = (type2 == FPType Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1, N);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result
                result_sign = if rounding == FPRounding_NEGINF then '1'
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;

```

### Library pseudocode for aarch32/functions/float/FPRSqrtStep

```

// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType Infinity); inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero); zero2 = (type2 == FPType Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0', N);
        result = FPHalvedSub(three, product, fpcr);
    return result;

```

### Library pseudocode for aarch32/functions/float/FPRecipStep

```

// FPRecipStep()
// =====

```

```

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRTyp fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0', N);
        result = FPSub(two, product, fpcr);
    return result;

```

## Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```

// StandardFPSCRValue()
// =====
FPCRTyp StandardFPSCRValue()
    bits(32) value = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000;
    return ZeroExtend(value, 64);

```

## Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```

// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc, boolean aligned]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    if SPESampleInFlight then
        boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);

```

```

        return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc,
                  boolean aligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
AccessDescriptor accdesc = accdesc_in;
assert IsAligned(address, size);

AddressDescriptor memaddrdesc;
memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch32.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID());
    if SPESampleInFlight then
        boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

PhysMemRetStatus memstatus;
memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
return;

```

### **Library pseudocode for aarch32/functions/memory/ AArch32.UnalignedAccessFaults**

```

// AArch32.UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

boolean AArch32.UnalignedAccessFaults(AccessDescriptor accdesc)
    return (AlignmentEnforced()) ||
           accdesc.a32lsmd ||
           accdesc.exclusive ||
           accdesc.acqsc ||
           accdesc.relsc;

```

### **Library pseudocode for aarch32/functions/memory/Hint\_PreloadData**

```

// Hint_PreloadData()
// =====

Hint_PreloadData(bits(32) address);

```

### **Library pseudocode for aarch32/functions/memory/Hint\_PreloadDataForWrite**

```

// Hint_PreloadDataForWrite()
// =====

Hint_PreloadDataForWrite(bits(32) address);

```

### Library pseudocode for aarch32/functions/memory/Hint\_PreloadInstr

```

// Hint_PreloadInstr()
// =====

Hint_PreloadInstr(bits(32) address);

```

### Library pseudocode for aarch32/functions/memory/MemA

```

// MemA[] - non-assignment form
// =====

bits(8*size) MemA[bits(32) address, integer size]
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel,
        return Mem with type[address, size, accdesc];

// MemA[] - assignment form
// =====

MemA[bits(32) address, integer size] = bits(8*size) value
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
        Mem with type[address, size, accdesc] = value;
    return;

```

### Library pseudocode for aarch32/functions/memory/MemO

```

// MemO[] - non-assignment form
// =====

bits(8*size) MemO[bits(32) address, integer size]
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked,
        return Mem with type[address, size, accdesc];

// MemO[] - assignment form
// =====

MemO[bits(32) address, integer size] = bits(8*size) value
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked,
        Mem with type[address, size, accdesc] = value;
    return;

```

### Library pseudocode for aarch32/functions/memory/MemS

```

// MemS[] - non-assignment form
// =====
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS[bits(32) address, integer size]
    AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp_LOAD);
    return Mem with type[address, size, accdesc];

// MemS[] - assignment form
// =====
// Memory accessor for streaming store multiple instructions

MemS[bits(32) address, integer size] = bits(8*size) value
    AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp_STORE);
    Mem with type[address, size, accdesc] = value;
    return;

```

### Library pseudocode for aarch32/functions/memory/MemU

```

// MemU[] - non-assignment form
// =====

bits(8*size) MemU[bits(32) address, integer size]
    boolean nontemporal = FALSE;
    boolean privileged = PSTATE.EL != ELO;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal);
    return Mem with type[address, size, accdesc];

// MemU[] - assignment form
// =====

MemU[bits(32) address, integer size] = bits(8*size) value
    boolean nontemporal = FALSE;
    boolean privileged = PSTATE.EL != ELO;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal);
    Mem with type[address, size, accdesc] = value;
    return;

```

### Library pseudocode for aarch32/functions/memory/MemU\_unpriv

```

// MemU_unpriv[] - non-assignment form
// =====

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    boolean nontemporal = FALSE;
    boolean privileged = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal);
    return Mem with type[address, size, accdesc];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    boolean nontemporal = FALSE;

```

```

boolean privileged = FALSE;
boolean tagchecked = FALSE;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal);
Mem_with_type[address, size, accdesc] = value;
return;

```

## Library pseudocode for aarch32/functions/memory/Mem\_with\_type

```

// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for
// instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access
    integer alignment = if accdesc.ispair then halfsize else size;
    boolean aligned = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if aligned then
        value = AArch32.MemSingle[address, size, accdesc, aligned];
    else
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, accdesc, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether
        // access will generate an Alignment Fault, as to get this far
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a bit

Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc]
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access
    integer alignment = if accdesc.ispair then halfsize else size;

```

```

boolean aligned = IsAligned(address, alignment);

if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
    AArch32.Abort(address, AlignmentFault(accdesc));

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse(value);

if aligned then
    AArch32.MemSingle[address, size, accdesc, aligned] = value;
else
    assert size > 1;
    AArch32.MemSingle[address, 1, accdesc, aligned] = value<7:0>

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether
    // access will generate an Alignment Fault, as to get this far
    // not, so we must be changing to a new translation page.

    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;

    for i = 1 to size-1
        AArch32.MemSingle[address+i, 1, accdesc, aligned] = value<8:1>;
return;

```

## Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```

// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1';

    if route_to_aarch64 then
        AArch64.ESBOperation();
        return;

    route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && EffectiveEA();
    route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE

    bits(5) target;
    if route_to_monitor then
        target = M32_Monitor;
    elseif route_to_hyp || PSTATE.M == M32_Hyp then
        target = M32_Hyp;
    else
        target = M32_Abort;

    boolean mask_active;
    if CurrentSecurityState() == SS_Secure then

```

```

        mask_active = TRUE;
    elseif target == M32_Monitor then
        mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0')
    else
        mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32(target);
intdis = Halted() || ExternalDebugInterruptsDisabled(el);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending() then
    bits(32) syndrome = Zeros(32);
    syndrome<31> = '1'; // A
    syndrome<15:0> = AArch32.PhysicalSErrorSyndrome();
    DISR = syndrome;
    ClearPendingPhysicalSError();

return;

```

### **Library pseudocode for aarch32/functions/ras/ AArch32.EncodeAsyncErrorSyndrome**

```

// AArch32.EncodeAsyncErrorSyndrome()
// =====
// Return the corresponding encoding for ErrorState.

bits(2) AArch32.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC      return '00';
        when ErrorState_UEU     return '01';
        when ErrorState_UEO     return '10';
        when ErrorState_UER     return '11';
        otherwise Unreachable();

```

### **Library pseudocode for aarch32/functions/ras/ AArch32.PhysicalSErrorSyndrome**

```

// AArch32.PhysicalSErrorSyndrome()
// =====
// Generate SError syndrome.

bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    FaultRecord fault = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        ErrorState errstate = AArch32.PEErrorState(fault);
        syndrome<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate);
        syndrome<9>      = fault.extflag;
        syndrome<5:0>      = '010001';
    else
        boolean long_format = TTBCR.EAE == '1';
        syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;

```

## Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =====
// Perform the ESB operation for virtual SError interrupts executed in
AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;
    // If physical SError interrupts are routed to Hyp mode, and TGE is
    // then a virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked       = vintdis || PSTATE.A == '1';
    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1';                                // A
        syndrome<15:14> = VDFSR<15:14>;                // AET
        syndrome<12> = VDFSR<12>;                      // EXT
        syndrome<9> = TTBCR.EAE;                          // LPAE
        if TTBCR.EAE == '1' then                           // Long-descriptor format
            syndrome<5:0> = '010001';                     // STATUS
        else
            syndrome<10,3:0> = '10110';                  // FS
        VDISR = syndrome;
        HCR.VA = '0';                                     // Clear pending virtual SError
    return;
```

## Library pseudocode for aarch32/functions/registers/ AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// =====
AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN; // No
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32 IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
```

```

        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;
    return;

```

### **Library pseudocode for aarch32/functions/registers/ AArch32.ResetSIMDFPRegisters**

```

// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

for i = 0 to 15
    Q[i] = bits(128) UNKNOWN;

return;

```

### **Library pseudocode for aarch32/functions/registers/ AArch32.ResetSpecialRegisters**

```

// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

// AArch32 special registers
SPSR_fiq<31:0> = bits(32) UNKNOWN;
SPSR_irq<31:0> = bits(32) UNKNOWN;
SPSR_svc<31:0> = bits(32) UNKNOWN;
SPSR_abt<31:0> = bits(32) UNKNOWN;
SPSR_und<31:0> = bits(32) UNKNOWN;
if HaveEL(EL2) then
    SPSR_hyp = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
if HaveEL(EL3) then
    SPSR_mon = bits(32) UNKNOWN;

// External debug special registers
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;

return;

```

### **Library pseudocode for aarch32/functions/registers/ AArch32.ResetSystemRegisters**

```

// AArch32.ResetSystemRegisters()
// =====

AArch32.ResetSystemRegisters(boolean cold_reset);

```

### **Library pseudocode for aarch32/functions/registers/ALUExceptionReturn**

```

// ALUEExceptionReturn()
// =====

ALUEExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32 User,M32 System} then
        Constraint c = ConstrainUnpredictable(Unpredictable_ALUEEXCEPTIO
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF
                UNDEFINED;
            when Constraint_NOP
                EndOfInstruction();
    else
        AArch32.ExceptionReturn(address, SPSR_curr[]);

```

### Library pseudocode for aarch32/functions/registers/ALUWritePC

```

// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXwritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);

```

### Library pseudocode for aarch32/functions/registers/BXWritePC

```

// BXWritePC()
// =====

BXWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the pr
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC
        if address<1> == '1' && ConstrainUnpredictableBool(Unpredictabl
            address<1> = '0';
    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(address, branch_type, branch_conditional);

```

### Library pseudocode for aarch32/functions/registers/BranchWritePC

```

// BranchWritePC()
// =====

BranchWritePC(bits(32) address_in, BranchType branch_type)

```

```

bits(32) address = address_in;
if CurrentInstrSet() == InstrSet_A32 then
    address<1:0> = '00';
else
    address<0> = '0';
boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
BranchTo(address, branch_type, branch_conditional);

```

### **Library pseudocode for aarch32/functions/registers/CBWritePC**

```

// CBWritePC()
// =====
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
    bits(32) address = address_in;
    assert CurrentInstrSet() == InstrSet_T32;
    address<0> = '0';
    boolean branch_conditional = TRUE;
    BranchTo(address, BranchType_DIR, branch_conditional);

```

### **Library pseudocode for aarch32/functions/registers/D**

```

// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 2, 128];
    return Elem[vreg, n MOD 2, 64];

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 2, 128];
    Elem[vreg, n MOD 2, 64] = value;
    V[n DIV 2, 128] = vreg;
    return;

```

### **Library pseudocode for aarch32/functions/registers/Din**

```

// Din[] - non-assignment form
// =====

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];

```

### **Library pseudocode for aarch32/functions/registers/LR**

```

// LR - assignment form
// =====

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// =====

bits(32) LR
    return R[14];

```

### Library pseudocode for aarch32/functions/registers/LoadWritePC

```

// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);

```

### Library pseudocode for aarch32/functions/registers/LookUpRIndex

```

// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of // Select index by mode:    usr fiq irq svc abt und hyp
        when 8      result = RBankSelect(mode, 8, 24, 8, 8, 8, 8,
        when 9      result = RBankSelect(mode, 9, 25, 9, 9, 9, 9,
        when 10     result = RBankSelect(mode, 10, 26, 10, 10, 10, 10,
        when 11     result = RBankSelect(mode, 11, 27, 11, 11, 11, 11,
        when 12     result = RBankSelect(mode, 12, 28, 12, 12, 12, 12,
        when 13     result = RBankSelect(mode, 13, 29, 17, 19, 21, 23,
        when 14     result = RBankSelect(mode, 14, 30, 16, 18, 20, 22,
        otherwise   result = n;

    return result;

```

### Library pseudocode for aarch32/functions/registers/Monitor\_mode\_registers

```

bits(32) SP_mon;
bits(32) LR_mon;

```

### Library pseudocode for aarch32/functions/registers/PC32

```

// AArch32 program counter
// PC32 - non-assignment form
// =====

```

```
bits(32) PC32
    return R[15];                                // This includes the offset from AArch32
```

### Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before
    // AArch32, it is permitted to instead return PC+4, provided it does so consistently.
    // It is used only to describe A32 instructions, so it returns the address of the instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC32;
```

### Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n, 128];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n, 128] = value;
    return;
```

### Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// =====

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

### Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =====
```

```

bits(32) R[integer n]
  if n == 15 then
    offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
    return _PC<31:0> + offset;
  else
    return Rmode[n, PSTATE.M];

```

## Library pseudocode for aarch32/functions/registers/RBankSelect

```

// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                     integer svc, integer abt, integer und, integer hyp)

  integer result;
  case mode of
    when M32_User      result = usr; // User mode
    when M32_FIQ       result = fiq; // FIQ mode
    when M32 IRQ      result = irq; // IRQ mode
    when M32_Svc       result = svc; // Supervisor mode
    when M32_Abort     result = abt; // Abort mode
    when M32_Hyp       result = hyp; // Hyp mode
    when M32_Undef     result = und; // Undefined mode
    when M32_System   result = usr; // System mode uses User mode
    otherwise             result = Unreachable(); // Monitor mode

  return result;

```

## Library pseudocode for aarch32/functions/registers/Rmode

```

// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
  assert n >= 0 && n <= 14;

  // Check for attempted use of Monitor mode in Non-secure state.
  if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
  assert !BadMode(mode);

  if mode == M32_Monitor then
    if n == 13 then return SP_mon;
    elseif n == 14 then return LR_mon;
    else return _R[n]<31:0>;
  else
    return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
  assert n >= 0 && n <= 14;

  // Check for attempted use of Monitor mode in Non-secure state.
  if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;

```

```

assert !BadMode(mode);

if mode == M32_Monitor then
    if n == 13 then SP_mon = value;
    elseif n == 14 then LR_mon = value;
    else _R[n]<31:0> = value;
else
    // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of
    // register are unchanged or set to zero. This is also tested for
    // exception entry, as this applies to all AArch32 registers.
    if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZERO)
        _R[LookUpRIndex(n, mode)] = ZeroExtend(value, 64);
    else
        _R[LookUpRIndex(n, mode)]<31:0> = value;

return;

```

## Library pseudocode for aarch32/functions/registers/S

```

// S[] - non-assignment form
// =====

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 4, 128];
    return Elem[vreg, n MOD 4, 32];

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 4, 128];
    Elem[vreg, n MOD 4, 32] = value;
    V[n DIV 4, 128] = vreg;
    return;

```

## Library pseudocode for aarch32/functions/registers/\_Dclone

```

// _Dclone[]
// =====
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use
// in instruction pseudocode, to avoid read-after-write for Advanced SIMD

array bits(64) _Dclone[0..31];

```

## Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```

// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the
    // exception entry point

```

```

// mechanism
SetPSTATEFromPSR(spsr);
ClearExclusiveLocal(ProcessorID());
SendEventLocal();

if PSTATE.IL == '1' then
    // If the exception return is illegal, PC[1:0] are UNKNOWN
    new_pc<1:0> = bits(2) UNKNOWN;
else
    // LR[1:0] or LR[0] are treated as being 0, depending on the ta
    if PSTATE.T == '1' then
        new_pc<0> = '0';                                // T32
    else
        new_pc<1:0> = '00';                            // A32

boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
BranchTo(new_pc, BranchType_ERET, branch_conditional);

CheckExceptionCatch(FALSE);                         // Check for debug event c

```

### **Library pseudocode for aarch32/functions/system/ AArch32.ExecutingCP10or11Instr**

```

// AArch32.ExecutingCP10or11Instr()
// =====

boolean AArch32.ExecutingCP10or11Instr()
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && in
    else // InstrSet_T32
        return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' ||

```

### **Library pseudocode for aarch32/functions/system/AArch32.ITAdvance**

```

// AArch32.ITAdvance()
// =====

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;

```

### **Library pseudocode for aarch32/functions/system/AArch32.SysRegRead**

```

// AArch32.SysRegRead()
// =====
// Read from a 32-bit AArch32 System register and write the register's
AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);

```

### **Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64**

```

// AArch32.SysRegRead64()
// =====
// Read from a 64-bit AArch32 System register and write the register's
AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer

```

### **Library pseudocode for aarch32/functions/system/ AArch32.SysRegReadCanWriteAPSR**

```

// AArch32.SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can
boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32\(\);
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0
        return TRUE;

    return FALSE;

```

### **Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite**

```

// AArch32.SysRegWrite()
// =====
// Read the contents of R[t] and write to a 32-bit AArch32 System register
AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);

```

### **Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64**

```

// AArch32.SysRegWrite64()
// =====
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer

```

## Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```
// AArch32.SysRegWriteM()
// =====
// Read a value from a virtual address and write it to an AArch32 System
// register.
AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// =====
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP = (if mode IN {M32\_User,M32\_System} then '0' else '1');
    return;
```

## Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// =====
// Function for dealing with writes to PSTATE.M from an AArch32 instruction.
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if 'mode' is invalid for this implementation
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction
    // to change PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception Level
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change
    // to EL2 is illegal if the value of HCR.TGE is '1'.
    if PSTATE.M == M32\_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == 1
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

## Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this
    boolean valid;
    case mode of
        when M32_Monitor
            valid = HaveAArch32EL(EL3);
        when M32_Hyp
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes
            // state, and EL1 modes in Non-secure state. If EL3 is not
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports
            valid = HaveAArch32EL(EL1);
        when M32_User
            valid = HaveAArch32EL(EL0);
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    return !valid;
```

## Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to
// other than the SPSRs that are invalid. This includes ELR_hyp accesses

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'
            if mode != M32_FIQ then UNPREDICTABLE;          // R8_usr to R12_
        when '00101'
            if mode == M32_System then UNPREDICTABLE;       // SP_usr
        when '00110'
            if mode IN {M32_Hyp, M32_System} then UNPREDICTABLE; // LR_usr
        when '010xx', '0110x', '01110'
            if mode == M32_FIQ then UNPREDICTABLE;          // R8_fiq to R12_
        when '1000x'
            if mode == M32_IRQ then UNPREDICTABLE;          // LR_irq, SP_irq
        when '1001x'
            if mode == M32_Svc then UNPREDICTABLE;          // LR_svc, SP_svc
        when '1010x'
            if mode == M32_Abort then UNPREDICTABLE;         // LR_abt, SP_abt
        when '1011x'
            if mode == M32_Undef then UNPREDICTABLE;         // LR_und, SP_und
        when '1110x'
            if (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure || mode == M32_Monitor) then UNPREDICTABLE; // LR_mon, SP_mon
        when '11110'
            if !HaveEL(EL2) || !(mode IN {M32_Monitor, M32_Hyp}) then UNPREDICTABLE; // ELR_hyp, only
        when '11111'
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE; // SP_hyp, only
```

```
        otherwise
            UNPREDICTABLE;
```

```
    return;
```

## Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by
CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;                                // PSTATE.<A,I,F,M> are
                                                        // read-only
    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if IsFeatureImplemented(FEAT_SSBS) then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                                // PSTATE.E is writable
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is
            AArch32.WriteModeByInstr(value<4:0>);
return;
```

## Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond()) ;
```

## Library pseudocode for aarch32/functions/system/CurrentCond

```
// CurrentCond()
// =====
```

```
bits(4) AArch32.CurrentCond();
```

### Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()  
// =====  
  
boolean InITBlock()  
    if CurrentInstrSet() == InstrSet_T32 then  
        return PSTATE.IT<3:0> != '0000';  
    else  
        return FALSE;
```

### Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()  
// =====  
  
boolean LastInITBlock()  
    return (PSTATE.IT<3:0> == '1000');
```

### Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()  
// =====  
  
SPSRWriteByInstr(bits(32) value, bits(4) bytemask)  
  
    bits(32) new_spsr = SPSR_curr[];  
  
    if bytemask<3> == '1' then  
        new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0], J b  
  
    if bytemask<2> == '1' then  
        new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags  
  
    if bytemask<1> == '1' then  
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A intere  
  
    if bytemask<0> == '1' then  
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit,  
  
    SPSR_curr[] = new_spsr; // UNPREDICTABLE if User or Sy  
  
    return;
```

### Library pseudocode for aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()  
// =====  
// Checks for MRS (Banked register) or MSR (Banked register) accesses t
```

```

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'
            if mode == M32_FIQ then UNPREDICTABLE;
        when '10000'
            if mode == M32_IRO then UNPREDICTABLE;
        when '10010'
            if mode == M32_Svc then UNPREDICTABLE;
        when '10100'
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'
            if (!HaveEL(EL3) || mode == M32_Monitor ||
                CurrentSecurityState() != SS_Secure) then UNPREDICTABLE;
        when '11110'
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;

```

## Library pseudocode for aarch32/functions/system>SelectInstrSet

```

// SelectInstrSet()
// =====

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;

```

## Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI\_ALL

```

// AArch32.DTLBI_ALL()
// =====
// Invalidate all data TLB entries for the indicated translation regime
// the indicated security state for all TLBs within the indicated shareability
// Invalidations applies to all applicable stage 1 and stage 2 entries.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability
                    TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security    = security;
    r.regime      = regime;
    r.level       = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);

```

```
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;
```

### Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI\_ASID

```
// AArch32.DTLBI_ASID()
// =====
// Invalidate all data TLB stage 1 entries matching the indicated VMID
// and ASID in the parameter Rt in the indicated translation regime with
// indicated security state for all TLBs within the indicated shareability
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, TLBIMemAttr attr, bits(32) rt);
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp_DASID;
    r.from_aarch64 = FALSE;
    r.security    = security;
    r.regime      = regime;
    r.vmid        = vmid;
    r.level       = TLBILevel_Any;
    r.attr         = attr;
    r.asid        = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;
```

### Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI\_VA

```

// AArch32.DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability
// matching the indicated VMID and ASID (where regime supports VMID, ASID)
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr);
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_DVA;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level        = level;
r.attr         = attr;
r.asid         = Zeros(8) : Rt<7:0>;
r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

```

### Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI\_ALL

```

// AArch32.ITLBI_ALL()
// =====
// Invalidate all instruction TLB entries for the indicated translation
// the indicated security state for all TLBs within the indicated shareability
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability,
                  TLBIMemAttr attr);
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_IALL;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.level        = TLBILevel_Any;
r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

```

### Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI\_ASID

```

// AArch32.ITLBI_ASID()
// =====
// Invalidate all instruction TLB stage 1 entries matching the indicated ASID
// (where regime supports) and ASID in the parameter Rt in the indicated
// shareability

```

```

// regime with the indicated security state for all TLBs within the indicated
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.ITLBInvalidate_ASID(SecurityState security, Regime regime, bits(16) vmid,
                           Shareability shareability, TLBIMemAttr attr, bits(32) asid);

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRRecord r;
r.op          = TLBIOp_IASID;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = TLBILevel_Any;
r.attr         = attr;
r.asid        = Zeros(8) : Rt<7:0>;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

```

### Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBInvalidate\_VA

```

// AArch32.ITLBInvalidate_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated
// matching the indicated VMID and ASID (where regime supports VMID, ASID)
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.ITLBInvalidate_VA(SecurityState security, Regime regime, bits(16) vmid,
                           Shareability shareability, TLBILevel level, TLBIMemAttr attr,
                           assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRRecord r;
r.op          = TLBIOp_IVA;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = level;
r.attr         = attr;
r.asid        = Zeros(8) : Rt<7:0>;
r.address     = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r);
return;

```

### Library pseudocode for aarch32/functions/tlbi/AArch32.TLBInvalidate\_ALL

```

// AArch32.TLBInvalidate_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

```

```

AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability sh
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp_ALL;
    r.from_aarch64 = FALSE;
    r.security    = security;
    r.regime      = regime;
    r.level       = TLBILevel_Any;
    r.attr        = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_ASID

```

// AArch32.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where re
// and ASID in the parameter Rt in the indicated translation regime wit
// indicated security state for all TLBs within the indicated shareabil
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIMemAttr attr, bits(32)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security    = security;
    r.regime      = regime;
    r.vmid        = vmid;
    r.level       = TLBILevel_Any;
    r.attr        = attr;
    r.asid        = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

## Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_IPAS2

```

// AArch32.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated sha
// domain matching the indicated VMID in the indicated regime with the
// Note: stage 1 and stage 2 combined entries are not in the scope of t
// IPA and related parameters of the are derived from Rt.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid
                  Shareability shareability, TLBILevel level, TLBIMemA
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS_NonSecure;

```

```

TLBIRRecord r;
r.op          = TLBIOp_IPAS2;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level        = level;
r.attr         = attr;
r.address     = Zeros(24) : Rt<27:0> : Zeros(12);
r.ipaspace    = PAS_NonSecure;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;
```

### Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VA

```

// AArch32.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability
// matching the indicated VMID and ASID (where regime supports VMID, ASID)
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAtt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRRecord r;
r.op          = TLBIOp_VA;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level        = level;
r.attr         = attr;
r.asid        = Zeros(8) : Rt<7:0>;
r.address     = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;
```

### Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VAA

```

// AArch32.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability
// matching the indicated VMID (where regime supports VMID) and all ASIDs
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAtt
assert PSTATE.EL IN {EL3, EL2, EL1};
```

```

TLBIRRecord r;
r.op          = TLBIOp_VAA;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level        = level;
r.attr         = attr;
r.address     = Zeros(32) : Rt<31:12> : Zeros(12);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

### Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VMALL

```

// AArch32.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime
// the indicated security state for all TLBs within the indicated share
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this
// Note: stage 2 only entries are not in the scope of this operation.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmic
                     Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRRecord r;
r.op          = TLBIOp_VMALL;
r.from_aarch64 = FALSE;
r.security    = security;
r.regime      = regime;
r.level        = TLBILevel_Any;
r.vmid        = vmid;
r.attr         = attr;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

### Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI\_VMALLS12

```

// AArch32.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated transla
// regime with the indicated security state for all TLBs within the inc
// shareability domain that match the indicated VMID.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) v
                      Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2};

TLBIRRecord r;
r.op          = TLBIOp_VMALLS12;
r.from_aarch64 = FALSE;

```

```

        r.security      = security;
        r.regime       = regime;
        r.level        = TLBILevel_Any;
        r.vmid         = vmid;
        r.attr          = attr;

        TLBI(r);
        if shareability != Shareability_NSH then Broadcast(shareability, r)
        return;
    
```

### **Library pseudocode for aarch32/functions/v6simd/Sat**

```

// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
    
```

### **Library pseudocode for aarch32/functions/v6simd/SignedSat**

```

// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
    
```

### **Library pseudocode for aarch32/functions/v6simd/UnsignedSat**

```

// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
    
```

### **Library pseudocode for aarch32/ic/AArch32.IC**

```

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;
    
```

```

cache.acctype    = AccessType_IC;
cache.cachetype = CacheType_Instruction;
cache.cacheop   = CacheOp_Invalidate;
cache.opscope   = opscope;
cache.security  = SecurityStateAtEL(PSTATE.EL);

if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
    if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_A
        && EL2Enabled() && HCR.FB == '1') then
        cache.shareability = Shareability_ISH;
    else
        cache.shareability = Shareability_NSH;
    cache.regval = ZeroExtend(regval, 64);
    CACHE_OP(cache);
else
    assert opscope == CacheOpScope_PoU;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

need_translate = ICInstNeedsTranslation(opscope);

cache.shareability = Shareability_NSH;
cache.vaddress     = ZeroExtend(regval, 64);
cache.translated   = need_translate;

if !need_translate then
    cache.paddress = FullAddress UNKNOWN;
    CACHE_OP(cache);
    return;

integer size      = 0;
boolean aligned   = TRUE;
AccessDescriptor accdesc = CreateAccDescIC(cache);
AddressDescriptor memaddrdesc = AArch32.TranslateAddress(regval);
if IsFault(memaddrdesc) then
    AArch32.Abort(regval, memaddrdesc.fault);

cache.paddress = memaddrdesc.paddress;
CACHE_OP(cache);
return;

```

## Library pseudocode for aarch32/predictionrestrict/AArch32.RestrictPrediction

```

// AArch32.RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

ExecutionCtxt c;
target_el      = val<25:24>;

// If the target EL is not implemented or the instruction is execute
// EL lower than the specified level, the instruction is treated as
if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then Enc

bit ns  = val<26>;
bit nse = bit UNKNOWN;
ss = TargetSecurityState(ns, nse);

c.security  = ss;
c.target_el = target_el;

if EL2Enabled() then
    if PSTATE.EL IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid     = FALSE;
        c.vmid         = VMID[];
    elseif target_el IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid     = val<27> == '1';
        c.vmid         = ZeroExtend(val<23:16>, 16);           // Only
    else
        c.is_vmid_valid = FALSE;
else
    c.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    c.is_asid_valid = TRUE;
    c.all_asid     = FALSE;
    c.asid         = ASID[];
else
    c.is_asid_valid = FALSE;

c.restriction = restriction;
RESTRICT_PREDICTIONS(c);

```

## Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode

```

// AArch32.DefaultTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, without TEX

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit

```

```


MemoryAttributes memattrs;
bits(3) TEX = TEX_in;
bit C = C_in;
bit B = B_in;

// Reserved values map to allocated values
if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00')
    bits(5) texcb;
    (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB,
    TEX = texcb<4:2>; C = texcb<1>; B = texcb<0>;

// Distinction between Inner Shareable and Outer Shareable is not s
// A memory region is either Non-shareable or Outer Shareable
case TEX:C:B of
    when '00000'
        // Device-nGnRnE
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
    when '00001', '01000'
        // Device-nGnRE
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRE;
        memattrs.shareability = Shareability_OSH;
    when '00010'
        // Write-through Read allocate
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs   = MemAttr_WT;
        memattrs.inner.hints   = MemHint_RA;
        memattrs.outer.attrs   = MemAttr_WT;
        memattrs.outer.hints   = MemHint_RA;
        memattrs.shareability = if s == '1' then Shareability_OSH e
    when '00011'
        // Write-back Read allocate
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs   = MemAttr_WB;
        memattrs.inner.hints   = MemHint_RA;
        memattrs.outer.attrs   = MemAttr_WB;
        memattrs.outer.hints   = MemHint_RA;
        memattrs.shareability = if s == '1' then Shareability_OSH e
    when '00100'
        // Non-cacheable
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs   = MemAttr_NC;
        memattrs.outer.attrs   = MemAttr_NC;
        memattrs.shareability = Shareability_OSH;
    when '00110'
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    when '00111'
        // Write-back Read and Write allocate
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs   = MemAttr_WB;
        memattrs.inner.hints   = MemHint_RWA;
        memattrs.outer.attrs   = MemAttr_WB;
        memattrs.outer.hints   = MemHint_RWA;
        memattrs.shareability = if s == '1' then Shareability_OSH e
    when '1xxxx'
        // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
        memattrs.memtype = MemType_Normal;
        memattrs.inner   = DecodeSDFAAttr(C:B);


```

```

        memattrs.outer    = DecodeSDFAttr(TEX<1:0>);

        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_WB
            memattrs.shareability = Shareability_OSH;
        else
            memattrs.shareability = if s == '1' then Shareability_C
        otherwise
            // Reserved, handled above
            Unreachable();
    }

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags           = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_NC
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
}

```

### Library pseudocode for aarch32/translation/attrs/AArch32.MAIRAttr

```

// AArch32.MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR
bits(8) AArch32.MAIRAttr(integer index, MAIRType mair)
    assert (index < 8);
    return Elem[mair, index, 8];
}

```

### Library pseudocode for aarch32/translation/attrs/ AArch32.RemappedTEXDecode

```

// AArch32.RemappedTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, with TEX remapping
MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX,
                                         MemoryAttributes memattrs;
                                         PRRR_Type prrr;
                                         NMRR_Type nmrr;

                                         region = UInt(TEX<0>:C:B);           // TEX<2:1> are ignored in this
                                         if region == 6 then
                                             return MemoryAttributes IMPLEMENTATION_DEFINED;

                                         if regime == Regime_EL30 then
                                             prrr = PRRR_S;
                                             nmrr = NMRR_S;
                                         elseif HaveAArch32EL(EL3) then
                                             prrr = PRRR_NS;
                                             nmrr = NMRR_NS;
                                         else
                                             prrr = PRRR;
                                         
```

```

nmrr = NMRR;

constant integer base = 2 * region;
attrfield = ELEM[prrr, region, 2];

if attrfield == '11' then      // Reserved, maps to allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable RESPREF);

case attrfield of
    when '00'                  // Device-nGnRnE
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
    when '01'                  // Device-nGnRE
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRE;
        memattrs.shareability = Shareability_OSH;
    when '10'
        NSn  = if s == '0' then prrr.NS0 else prrr.NS1;
        NOSm = prrr<region+24> AND NSn;
        IRn  = nmrr<base+1:base>;
        ORn  = nmrr<base+17:base+16>;
        memattrs.memtype = MemType_Normal;
        memattrs.inner     = DecodeSDFAAttr(IRn);
        memattrs.outer     = DecodeSDFAAttr(ORn);
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_WB
            memattrs.shareability = Shareability_OSH;
        else
            bits(2) sh = NSn:NOSm;
            memattrs.shareability = DecodeShareability(sh);
    when '11'
        Unreachable();
// The Transient hint is not supported in this format
memattrs.inner.transient = FALSE;
memattrs.outer.transient = FALSE;
memattrs.tags           = MemTag_Untagged;

if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_NC
    memattrs.xs = '0';
else
    memattrs.xs = '1';

return memattrs;

```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```

// AArch32.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddr"
// in translation regime "regime", when either debug exceptions are enabled,
// or halting is allowed.

FaultRecord AArch32.CheckBreakpoint(FaultRecord fault_in, bits(32) vaddr,
                                    AccessDescriptor accdesc, integer size,
                                    assert ELUsingAArch32(S1TranslationRegime()) );
assert size IN {2,4};

```

```

FaultRecord fault = fault_in;
match = FALSE;
mismatch = FALSE;

for i = 0 to NumBreakpointsImplemented() - 1
    (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, ac
    match = match || match_i;
    mismatch = mismatch || mismatch_i;

if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Breakpoint;
    Halt(reason);
elseif (match || mismatch) then
    fault.statuscode = Fault_Debug;
    fault.debugmoe = DebugException_Breakpoint;

return fault;

```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```

// AArch32.CheckDebug()
// =====
// Called on each access to check for a debug exception or entry to Debug

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccessDescriptor accdesc)

FaultRecord fault = NoFault(accdesc);

boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType_IFETCH);
boolean i_side = (accdesc.acctype == AccessType_IFETCH);
generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCReg[0];
halt = HaltOnBreakpointOrWatchpoint();
// Relative priority of Vector Catch and Breakpoint exceptions not
vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTOR);

if i_side && vector_catch_first && generate_exception then
    fault = AArch32.CheckVectorCatch(fault, vaddress, size);

if fault.statuscode == Fault_None && (generate_exception || halt)
    if d_side then
        fault = AArch32.CheckWatchpoint(fault, vaddress, accdesc, size);
    elseif i_side then
        fault = AArch32.CheckBreakpoint(fault, vaddress, accdesc, size);

if fault.statuscode == Fault_None && i_side && !vector_catch_first && !halt
    return AArch32.CheckVectorCatch(fault, vaddress, size);

return fault;

```

## Library pseudocode for aarch32/translation/debug/ AArch32.CheckVectorCatch

```

// AArch32.CheckVectorCatch()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress"
// translation regime, when debug exceptions are enabled.

```

```

FaultRecord AArch32.CheckVectorCatch(FaultRecord fault_in, bits(32) vaddr,
                                     assert ELUsingAArch32(S1TranslationRegime()));

FaultRecord fault = fault_in;
match = AArch32.VCRMatch(vaddress);
if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
    match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

if match then
    fault.statuscode = Fault_Debug;
    fault.debugmoe = DebugException_VectorCatch;

return fault;

```

## Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```

// AArch32.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "addr"
// when either debug exceptions are enabled for the access, or halting
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(FaultRecord fault_in, bits(32) vaddr,
                                    AccessDescriptor accdesc, integer size,
                                    assert ELUsingAArch32(S1TranslationRegime()));
FaultRecord fault = fault_in;

if accdesc.acctype == AccessType_DC then
    if accdesc.cacheop != CacheOp_Invalidate then
        return fault;
    elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoints")
        return fault;
elsif !IsDataAccess(accdesc.acctype) then
    return fault;

match = FALSE;
for i = 0 to NumWatchpointsImplemented() - 1
    if AArch32.WatchpointMatch(i, vaddress, size, accdesc) then
        match = TRUE;

if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Watchpoint;
    EDWAR = ZeroExtend(vaddress, 64);
    Halt(reason);
elsif match then
    fault.statuscode = Fault_Debug;
    fault.debugmoe = DebugException_Watchpoint;

return fault;

```

## Library pseudocode for aarch32/translation/fauluts/AArch32.IPAIsOutOfRange

```

// AArch32.IPAIsOutOfRange()
// =====
// Check intermediate physical address bits not resolved by translation

```

```

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    constant integer iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize);

```

### Library pseudocode for aarch32/translation/faulsts/ AArch32.S1HasAlignmentFault

```

// AArch32.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data access
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                    bit ntlsmd, MemoryAttributes memattr)
{
    if accdesc.acctype == AccessType IFETCH then
        return FALSE;
    elsif accdesc.a32lsm && ntlsmd == '0' then
        return memattr.memtype == MemType Device && memattr.device != 0;
    elsif accdesc.acctype == AccessType DCZero then
        return memattr.memtype == MemType Device;
    else
        return memattr.memtype == MemType Device && !aligned;
}

```

### Library pseudocode for aarch32/translation/faulsts/ AArch32.S1LDHasPermissionsFault

```

// AArch32.S1LDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, S1TTWParams walkparams,
                                         MemType memtype, PASpace paspac)
{
    bit r, w, x;
    bit pr, pw;
    bit ur, uw;
    bit xn;

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 or any
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL1 or any

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw, '0', '0'); // Privileged
            when '10' (pr,pw,ur,uw) = ( pr, '0', ur, '0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr, '0', '0', '0'); // Read-only

        xn = perms.xn OR perms.xn_table;
        pxn = perms.pxn OR perms.pxn_table;
}

```

```

ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn)) OR (uw AND
if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
    pan = PSTATE.PAN AND (ur OR uw);
    pr = pr AND NOT(pan);
    pw = pw AND NOT(pan);

(r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

// Prevent execution from Non-secure space by PE in Secure state
if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
    x = x AND NOT(walkparams.sif);
else
    // Apply leaf permissions
    case perms.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only

    // Apply hierarchical permissions
    case perms.ap_table<1> of
        when '0' (r,w) = ( r , w ); // No effect
        when '1' (r,w) = ( r , '0' ); // Read-only

xn = perms.xn OR perms.xn_table;
x = NOT(xn OR (w AND walkparams.wxn));

if accdesc.acctype == AccessType_IFETCH then
    constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    if constraint == Constraint_FAULT && memtype == MemType_Device
        return TRUE;
    else
        return x == '0';
elseif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
    return FALSE;
elseif accdesc.write then
    return w == '0';
else
    return r == '0';

```

## Library pseudocode for aarch32/translation/faults/ AArch32.S1SDHasPermissionsFault

```

    else
        sctlr = SCLTR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short c

        case perms.ap of
            when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1
            when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1
            when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any
            // '100' is reserved
            when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any

    else // Simplified access permissions model
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw

    if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Prevent execution from Non-secure space by PE in Secure state if
    if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.

    if accdesc.acctype == AccessType_IFETCH then
        if (memtype == MemType_Device &&
            ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == ConstrainUnpredictable(ConstrainUnpredictable_INSTRDEVICE))
            return TRUE;
        else
            return x == '0';
    elseif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
    elseif accdesc.write then
        return w == '0';
    else
        return r == '0';

```

## Library pseudocode for aarch32/translation/faults/ AArch32.S2HasAlignmentFault

```

// AArch32.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data a
// to Device memory

```

```

boolean AArch32.S2HasAlignmentFault(AccessDescriptor accdesc, boolean MemoryAttributes memattrs)
{
    if accdesc.acctype == AccessType_IFETCH then
        return FALSE;
    elseif accdesc.acctype == AccessType_DCZero then
        return memattrs.memtype == MemType_Device;
    else
        return memattrs.memtype == MemType_Device && !aligned;
}

```

### **Library pseudocode for aarch32/translation/faulsts/ AArch32.S2HasPermissionsFault**

```

// AArch32.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(S2TTWParams walkparams, Permissions perms,
                                      AccessDescriptor accdesc)
{
    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if IsFeatureImplemented(FEAT_XNX) then
        case perms.s2xn:perms.s2nxn of
            when '00' (px, ux) = (r, r);
            when '01' (px, ux) = ('0', r);
            when '10' (px, ux) = ('0', '0');
            when '11' (px, ux) = (r, '0');

            x = if accdesc.el == EL0 then ux else px;
    else
        x = r AND NOT(perms.s2xn);

    if accdesc.acctype == AccessType_TTW then
        return (walkparams.ptw == '1' && memtype == MemType_Device) ||

    elseif accdesc.acctype == AccessType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        return (constraint == Constraint_FAULT && memtype == MemType_Device);

    elseif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;

    elseif accdesc.write then
        return w == '0';

    else
        return r == '0';
}

```

### **Library pseudocode for aarch32/translation/faulsts/AArch32.S2InconsistentSL**

```

// AArch32.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

```

```

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch32.S2StartLevel(walkparams.s10);
    levels      = FINAL LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial

    // Can accomodate 1 more stride in the level + concatenation of up
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize         = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;

```

### Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```

// AArch32.VAIsOutOfRange()
// =====
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, b
    if regime == Regime EL2 then
        // Input Address size
        constant integer iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elseif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        constant integer lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        constant integer up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize) && !IsOnes(va<31:up_iasize);
    else
        return FALSE;

```

### Library pseudocode for aarch32/translation/tlbcontext/ AArch32.GetS1TLBContext

```

// AArch32.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLE

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bit
    TLBContext tlbcontext;

    case regime of
        when Regime EL2 tlbcontext = AArch32.TLBContextEL2(va);
        when Regime EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime EL30 tlbcontext = AArch32.TLBContextEL30(va);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stag

```

```

    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
}

```

## Library pseudocode for aarch32/translation/tlbcontext/ AArch32.GetS2TLBContext

```

// AArch32.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS\_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS\_NonSecure;
    tlbcontext.regime      = Regime\_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
    tlbcontext.tg          = TGx\_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    tlbcontext.cnp         = if IsFeatureImplemented(FEAT_TTCNP) then V

    return tlbcontext;
}

```

## Library pseudocode for aarch32/translation/tlbcontext/ AArch32.TLBContextEL10

```

// AArch32.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcn;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR_Type contextidr;

    if HaveAArch32EL(EL3) then
        ttbcn     = TTBCR_NS;
        ttbr0     = TTBR0_NS;
        ttbr1     = TTBR1_NS;
        contextidr = CONTEXTIDR_NS;
    else
        ttbcn     = TTBCR;
        ttbr0     = TTBR0;
        ttbr1     = TTBR1;
        contextidr = CONTEXTIDR;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL10;

    if AArch32.EL2Enabled(ss) then

```

```

    tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if IsFeatureImplemented(FEAT_TTCNP) && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange_LC
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;

```

### **Library pseudocode for aarch32/translation/tlbcontext/ AArch32.TLBContextEL2**

```

// AArch32.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS_NonSecure;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.ia     = ZeroExtend(va, 64);
    tlbcontext.tg     = TGx_4KB;
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then HTTBR0.CnP;

    return tlbcontext;

```

### **Library pseudocode for aarch32/translation/tlbcontext/ AArch32.TLBContextEL30**

```

// AArch32.TLBContextEL30()
// =====
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS_Secure;
    tlbcontext.regime = Regime_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);

```

```

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if IsFeatureImplemented(FEAT_TTCNP) && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARanc
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;

```

## Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```

// AArch32.EL2Enabled()
// =====
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS_Secure then
        if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2)) then
            return FALSE;
        elseif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implemen"
    else
        return HaveEL(EL2);

```

## Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```

// AArch32.FullTranslate()
// =====
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccessDescriptor accdesc)

    // Prepare fault fields in case a fault is detected
    FaultRecord fault = NoFault(accdesc);
    Regime regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    AddressDescriptor ipa;
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, va, aligned);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, va, aligned);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va, 64);
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, aligned, accdesc);

```

```

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fa);
        else
            return pa;
    else
        return ipa;

```

### Library pseudocode for aarch32/translation/translation/ AArch32.OutputDomain

```

// AArch32.OutputDomain()
// =====
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    if regime == Regime_EL30 then
        Dn = Elem[DACR_S, UInt(domain), 2];
    elseif HaveAArch32EL(EL3) then
        Dn = Elem[DACR_NS, UInt(domain), 2];
    else
        Dn = Elem[DACR, UInt(domain), 2];

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable RESDACP, 2);

    return Dn;

```

### Library pseudocode for aarch32/translation/translation/ AArch32.S1DisabledOutput

```

// AArch32.S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default
// values for memory attributes and access descriptor fields.

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord f,
                                                       bits(32) va,
                                                       AccessDescriptor accdesc)

FaultRecord fault = fault_in;
// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
bit default_cacheable;
if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
    if ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
        default_cacheable = HCR.DC;
    else
        default_cacheable = HCR_EL2.DC;
else
    default_cacheable = '0';

if default_cacheable == '1' then
    // Use default cacheable settings
    memattrs.memtype      = MemType_Normal;

```

```

memattrs.inner.attrs = MemAttr_WB;
memattrs.inner.hints = MemHint_RWA;
memattrs.outer.attrs = MemAttr_WB;
memattrs.outer.hints = MemHint_RWA;
memattrs.shareability = Shareability_NSH;
if (EL2Enabled() && !ELStateUsingAArch32(EL2, accdesc.ss == SS
    IsFeatureImplemented(FEAT_MTE2) && HCR_EL2.DCT == '1') th
    memattrs.tags = MemTag_AllocationTagged;
else
    memattrs.tags = MemTag_Untagged;
memattrs.xs = '0';
elseif accdesc.acctype == AccessType_IFETCH then
    memattrs.memtype = MemType_Normal;
    memattrs.shareability = Shareability_OSH;
    memattrs.tags = MemTag_Untagged;
    if AArch32.S1ICacheEnabled(regime) then
        memattrs.inner.attrs = MemAttr_WT;
        memattrs.inner.hints = MemHint_RA;
        memattrs.outer.attrs = MemAttr_WT;
        memattrs.outer.hints = MemHint_RA;
    else
        memattrs.inner.attrs = MemAttr_NC;
        memattrs.outer.attrs = MemAttr_NC;
    memattrs.xs = '1';
else
    // Treat memory region as Device
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
    memattrs.tags = MemTag_Untagged;
    memattrs.xs = '1';

bit ntlsmd;
if IsFeatureImplemented(FEAT_LSMAOC) then
    case regime of
        when Regime_EL30 ntlsmd = SCTRL_S.nTLSMD;
        when Regime_EL2 ntlsmd = HSCTRL.nTLSMD;
        when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTRL_
else
    ntlsmd = '1';

if AArch32.S1HasAlignmentFault(accdesc, aligned, ntlsmd, memattrs)
    fault.statuscode = Fault_Alignment;
    return (fault, AddressDescriptor_UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va, 56);
oa.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_No
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa);

```

## Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```

// AArch32.S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active transl

```

```

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elseif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elseif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) ==
    elseif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M ==
else
    return EL2Enabled() && HCR_EL2.<TGE,DC> == '00' && SCTLR.M ==

```

## Library pseudocode for aarch32/translation/translation/ AArch32.S1TranslateLD

```

// AArch32.S1TranslateLD()
// =====
// Perform a stage 1 translation using long-descriptor format mapping w
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord faul
bits(32) va, bo
AccessDescriptor accdesc)
FaultRecord fault = fault_in;

if !AArch32.S1Enabled(regime, accdesc.ss) then
    return AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);

walkparams = AArch32.GetS1TTWParams(regime, va);

if AArch32.VAIsOutOfRange(regime, walkparams, va) then
    fault.level      = 1;
    fault.statuscode = Fault_Translation;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, accdesc);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

if AArch32.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmd,
    fault.statuscode = Fault_Alignment;
elseif AArch32.S1LDHasPermissionsFault(regime, walkparams,
    walkstate.permissions,
    walkstate.memattrs.memtype,
    walkstate.baseaddress.paspace,
    accdesc) then
    fault.statuscode = Fault_Permission;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((accdesc.acctype == AccessType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1HasIFetchDevice(regime, walkstate.memattrs.memtype))
    )

```

```

        (accdesc.acctype != AccessType_IFETCH &&
         walkstate.memattrs.memtype == MemType_Normal && !AArch32.S
         // Treat memory attributes as Normal Non-Cacheable
         memattrs = NormalNCMemAttr();
         memattrs.xs = walkstate.memattrs.xs;
     else
         memattrs = walkstate.memattrs;

     // Shareability value of stage 1 translation subject to stage 2 is
     // to be either effective value or descriptor value
     if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
         (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM
          !boolean IMPLEMENTATION_DEFINED "Apply effective shareability a
          memattrs.shareability = walkstate.memattrs.shareability;
     else
         memattrs.shareability = EffectiveShareability(memattrs);

     // Output Address
     oa = StageOA(ZeroExtend(va, 64), walkparams.d128, walkparams.tgx,
     ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

     return (fault, ipa);

```

### **Library pseudocode for aarch32/translation/translation/ AArch32.S1TranslateSD**

```

// AArch32.S1TranslateSD()
// =====
// Perform a stage 1 translation using short-descriptor format mapping
// depending on the regime

(FaultRecord, AddressDescriptor, SDFType) AArch32.S1TranslateSD(FaultRe
bits(32
AccessD

FaultRecord fault = fault_in;

if !AArch32.S1Enabled(regime, accdesc.ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, va, alio
    return (fault, ipa, SDFType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, accdesc, va);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, SDFType UNKNOWN);

domain = AArch32.OutputDomain(regime, walkstate.domain);
SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

bit ntlsmd;
if IsFeatureImplemented(FEAT_LSMAOC) then
    case regime of
        when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
        when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_
else
    ntlsmd = '1';

```

```

if AArch32.S1HasAlignmentFault(accdesc, aligned, ntlsmd, walkstate.
    fault.statuscode = Fault Alignment;
elseif !(accdesc.acctype IN {AccessType IC, AccessType DC}) &&
    domain == Domain NoAccess) then
    fault.statuscode = Fault Domain;
elseif domain == Domain Client then
    if AArch32.S1SDHasPermissionsFault(regime, walkstate.permission,
        walkstate.memattrs.memtype,
        walkstate.baseaddress.paspace
        accdesc) then
        fault.statuscode = Fault Permission;

if fault.statuscode != Fault None then
    fault.domain = walkstate.domain;
    return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

MemoryAttributes memattrs;
if ((accdesc.acctype == AccessType IFETCH &&
    (walkstate.memattrs.memtype == MemType Device || !AArch32.S
    (accdesc.acctype != AccessType IFETCH &&
        walkstate.memattrs.memtype == MemType Normal && !AArch32.S
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is
// to be either effective value or descriptor value
if (regime == Regime EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS Secure) then HCR.VM
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability a
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype)
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa, walkstate.sdftype);

```

## Library pseudocode for aarch32/translation/translation/AArch32.S2Translate

```

// AArch32.S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_
                                                boolean aligned, A

FaultRecord fault = fault_in;
assert IsZero(ipa.paddress.address<55:40>);

if !ELStateUsingAArch32(EL2, accdesc.ss == SS Secure) then
    s1aarch64 = FALSE;
    return AArch64.S2Translate(fault, ipa, s1aarch64, aligned, acc

```

```

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None;
fault.secondstage = TRUE;
fault.s2fs1walk = accdesc.acctype == AccessType_TTW;
fault.ipaddress = ipa.paddress;

walkparams = AArch32.GetS2TTWParams();

if walkparams.vm == '0' then
    // Stage 2 is disabled
    return (fault, ipa);

if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>)
    fault.statuscode = Fault_Translation;
    fault.level = 1;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S2Walk(fault, walkparams, accdesc, ipa);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

if AArch32.S2HasAlignmentFault(accdesc, aligned, walkstate.memattrs)
    fault.statuscode = Fault_Alignment;
elseif AArch32.S2HasPermissionsFault(walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    accdesc) then
    fault.statuscode = Fault_Permission;
MemoryAttributes s2_memattrs;
if ((accdesc.acctype == AccessType_TTW &&
     walkstate.memattrs.memtype == MemType_Device) ||
    (accdesc.acctype == AccessType_IFETCH &&
     (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == ...
      accdesc.acctype != AccessType_IFETCH &&
      walkstate.memattrs.memtype == MemType_Normal) && HCR2.CD == ...
      // Treat memory attributes as Normal Non-Cacheable
      s2_memattrs = NormalNCMemAttr();
      s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

s2aarch64 = FALSE;
memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);

return (fault, pa);

```

## Library pseudocode for aarch32/translation/translation/AArch32.SDStageOA

```

// AArch32.SDStageOA()
// =====
// Given the final walk state of a short-descriptor translation walk,

```

```

// map the untranslated input address bits to the base output address
FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFSize sdftype);
    constant integer tsize = SDFSize(sdftype);

    // Output Address
    FullAddress oa;
    oa.address = baseaddress.address<55:tsize> : va<tsize-1:0>;
    oa.paspace = baseaddress.paspace;
    return oa;

```

### Library pseudocode for aarch32/translation/translation/ AArch32.TranslateAddress

```

// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccessDescriptor accdesc,
                                         boolean aligned, integer size);

    Regime regime = TranslationRegime(PSTATE.EL);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), accdesc, aligned);

    AddressDescriptor result = AArch32.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(va, accdesc, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;

```

### Library pseudocode for aarch32/translation/translation/SDFSize

```

// SDFSize()
// =====
// Returns the short-descriptor format translation granule size

integer SDFSize(SDFType sdftype)
    case sdftype of
        when SDFType_SmallPage      return 12;
        when SDFType_LargePage       return 16;
        when SDFType_Section        return 20;
        when SDFType_Supersection   return 24;
        otherwise Unreachable();

```

### Library pseudocode for aarch32/translation/walk/ AArch32.DecodeDescriptorTypeID

```

// AArch32.DecodeDescriptorTypeID()
// =====
// Determine whether the long-descriptor is a page, block or table

```

```

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Leaf;
    elseif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elseif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Leaf;
    else
        return DescriptorType_Invalid;

```

### Library pseudocode for aarch32/translation/walk/ AArch32.DecodeDescriptorTypeSD

```

// AArch32.DecodeDescriptorTypeSD()
// =====
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDFType_Table;
    elseif level == 1 && descriptor<18,1> == '01' then
        return SDFType_Section;
    elseif level == 1 && descriptor<18,1> == '11' then
        return SDFType_Supersection;
    elseif level == 2 && descriptor<1:0> == '01' then
        return SDFType_LargePage;
    elseif level == 2 && descriptor<1:0> IN {'1x'} then
        return SDFType_SmallPage;
    else
        return SDFType_Invalid;

```

### Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```

// AArch32.S1IASize()
// =====
// Retrieve the number of bits containing the input address for stage 1

integer AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);

```

### Library pseudocode for aarch32/translation/walk/AArch32.S1WalkLD

```

// AArch32.S1WalkLD()
// =====
// Traverse stage 1 translation tables in long format to obtain the final
// (FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime,
//                                         S1TTWParams walkparams, AccessType access,
//                                         bits(32) va)
//                                         FaultRecord fault = fault_in;
bits(3) txsz;
bits(64) ttbr;
bit epd;
VARange varange;

```

```

if regime == Regime_EL2 then
    ttbr = HTTBR;
    txsz = walkparams.t0sz;
    varange = VARange_LOWER;
else
    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1
bits(64) ttbr0;
bits(64) ttbr1;
TTBCR_Type ttbcr;
if regime == Regime_EL30 then
    ttbcr = TTBCR_S;
    ttbr0 = TTBR0_S;
    ttbr1 = TTBR1_S;
elsif HaveAArch32EL(EL3) then
    ttbcr = TTBCR_NS;
    ttbr0 = TTBR0_NS;
    ttbr1 = TTBR1_NS;
else
    ttbcr = TTBCR;
    ttbr0 = TTBR0;
    ttbr1 = TTBR1;

assert ttbcr.EAE == '1';
if varange == VARange_LOWER then
    txsz = walkparams.t0sz;
    ttbr = ttbr0;
    epd = ttbcr.EPD0;
else
    txsz = walkparams.t1sz;
    ttbr = ttbr1;
    epd = ttbcr.EPD1;

if regime != Regime_EL2 && epd == '1' then
    fault.level = 1;
    fault.statuscode = Fault_Translation;
    return (fault, TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S1IASize(txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
startlevel = FINAL LEVEL - (((iasize-1) - granulebits) DIV stride);
levels = FINAL LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
constant integer baselsb = (iasize - (levels*stride + granulebits));
baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure el
baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb), 5

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level = startlevel;
walkstate.istable = TRUE;
// In regimes that support global and non-global translations, tra
// table entries from lookup levels other than the final level of lo

```

```

// are treated as being non-global
walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgr);
walkstate.permissions.ap_table = '00';
walkstate.permissions.xn_table = '0';
walkstate.permissions.pxn_table = '0';

bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM
     !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability a
     walkaddress.memattrs.shareability = walkstate.memattrs.shareabi
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkac

DescriptorType desctype;
integer msb_residue = iasize - 1;
repeat
    fault.level = walkstate.level;
    constant integer indexlsb = (FINAL_LEVEL - walkstate.level)*str
    constant integer indexmsb = msb_residue;
    bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);

    walkaddress.paddress.address = walkstate.baseaddress.address OR
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    boolean toplevel = walkstate.level == startlevel;
    AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, var
    // If there are two stages of translation, then the first stage
    // are themselves subject to translation
    if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        s2aligned = TRUE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walka
                                         walkaccess);
        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walk
                                         fault, 64);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkac
                                         fault, 64);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);

    desctype = AArch32.DecodeDescriptorTypeID(descriptor, walkstate

```

```

        case desctype of
            when DescriptorType_Table
                if !IsZero(descriptor<47:40>) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, TTWState UNKNOWN);

                walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>);
                if walkstate.baseaddress.paspace == PAS_Secure && descriptor<5:4> == SS_Secure
                    walkstate.baseaddress.paspace = PAS_NonSecure;

                if walkparams.hpd == '0' then
                    walkstate.permissions.xn_table = (walkstate.permissions.xn_table & descriptor<60:47>)
                    walkstate.permissions.ap_table = (walkstate.permissions.ap_table & descriptor<62:47>)
                    walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table & descriptor<59:47>)

                walkstate.level = walkstate.level + 1;
                msb_residue = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, TTWState UNKNOWN);

            when DescriptorType_Leaf
                walkstate.istable = FALSE;

        until desctype == DescriptorType_Leaf;  

  

        // Check the output address is inside the supported range
        if !IsZero(descriptor<47:40>) then
            fault.statuscode = Fault_AddressSize;
            return (fault, TTWState UNKNOWN);

        // Check the access flag
        if descriptor<10> == '0' then
            fault.statuscode = Fault_AccessFlag;
            return (fault, TTWState UNKNOWN);

        walkstate.permissions.xn = descriptor<54>;
        walkstate.permissions.pxn = descriptor<53>;
        walkstate.permissions.ap = descriptor<7:6>:'1';
        walkstate.contiguous = descriptor<52>;
        if regime == Regime_EL2 then
            // All EL2 regime accesses are treated as Global
            walkstate.nG = '0';
        elseif accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_Secure
            // When a PE is using the Long-descriptor translation table for
            // and is in Secure state, a translation must be treated as non-secure
            // regardless of the value of the nG bit,
            // if NSTable is set to 1 at any level of the translation table
            walkstate.nG = '1';
        else
            walkstate.nG = descriptor<11>;  

  

        constant integer indexlsb = (FINAL_LEVEL - walkstate.level)*stride;
        walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>);
        if walkstate.baseaddress.paspace == PAS_Secure && descriptor<5> == SS_Secure

```

```
walkstate.baseaddress.paspace = PAS_NonSecure;  
  
memattr = descriptor<4:2>;  
sh      = descriptor<9:8>;  
attr    = AArch32.MAIRAttr(UInt(memattr), walkparams.mair);  
slaarch64 = FALSE;  
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64, walkpara  
  
return (fault, walkstate);
```

## Library pseudocode for aarch32/translation/walk/AArch32.S1WalkSD

```

// AArch32.S1WalkSD()
// =====
// Traverse stage 1 translation tables in short format to obtain the first
// page table entry

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime,
                                         AccessDescriptor accdesc, bits)

FaultRecord fault = fault_in;
SCTLR_Type sctlr;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;
// Determine correct translation control registers to use.
if regime == Regime_EL30 then
    sctlr = SCTLR_S;
    ttbcr = TTBCR_S;
    ttbr0 = TTBR0_S;
    ttbr1 = TTBR1_S;
elsif HaveAArch32EL(EL3) then
    sctlr = SCTLR_NS;
    ttbcr = TTBCR_NS;
    ttbr0 = TTBR0_NS;
    ttbr1 = TTBR1_NS;
else
    sctlr = SCTLR;
    ttbcr = TTBCR;
    ttbr0 = TTBR0;
    ttbr1 = TTBR1;

assert ttbcr.EAE == '0';
ee = sctlr.EE;
afe = sctlr.AFE;
tre = sctlr.TRE;
constant integer ttbcr_n = UInt(ttbcr.N);
constant boolean use_ttbr0 = IsZero(va<31:(32-ttbcr_n)>);
VARange varange = (if ttbcr_n == 0 || use_ttbr0 then VARange_LOWER
                     else VARange_UPPER);
constant integer n = if varange == VARange_LOWER then ttbcr_n else
bits(32) ttb;
bits(1) pd;
bits(2) irgn;
bits(2) rgn;
bits(1) s;
bits(1) nos;
if varange == VARange_LOWER then
    ttb = ttbr0.TTB0:Zeros(7);
    pd = ttbcr.PD0;

```

```

        irgn = ttbr0.IRGN;
        rgn  = ttbr0.RGN;
        s    = ttbr0.S;
        nos  = ttbr0.NOS;
    else
        ttb  = ttbr1.TTB1:Zeros(7);
        pd   = ttbcr.PD1;
        irgn = ttbr1.IRGN;
        rgn  = ttbr1.RGN;
        s    = ttbr1.S;
        nos  = ttbr1.NOS;

// Check if Translation table walk disabled for translations with t
if pd == '1' then
    fault.level      = 1;
    fault.statuscode = Fault_Translation;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure el
baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 56);

constant integer startlevel = 1;
TTWState walkstate;
walkstate.baseaddress = baseaddress;
// In regimes that support global and non-global translations, tra
// table entries from lookup levels other than the final level of lo
// are treated as being non-global. Translations in Short-Descriptor
// always support global & non-global translations.
walkstate.nG          = '1';
walkstate.memattrs    = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level       = startlevel;
walkstate.istable     = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM
     !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability a
     walkaddress.memattrs.shareability = walkstate.memattrs.shareabi
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkad

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;

```



```

56);
walkstate.istable = FALSE;

when SDFType_LargePage
    xn  = descriptor<15>;
    tex = descriptor<14:12>;
    nG  = descriptor<11>;
    s   = descriptor<10>;
    ap  = descriptor<9,5:4>;
    c   = descriptor<3>;
    b   = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<3
56);
walkstate.istable = FALSE;

when SDFType_Section
    ns   = descriptor<19>;
    nG  = descriptor<17>;
    s   = descriptor<16>;
    ap  = descriptor<15,11:10>;
    tex = descriptor<14:12>;
    domain = descriptor<8:5>;
    xn  = descriptor<4>;
    c   = descriptor<3>;
    b   = descriptor<2>;
    pxn = descriptor<0>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<3
56);
walkstate.istable = FALSE;

when SDFType_Supersection
    ns   = descriptor<19>;
    nG  = descriptor<17>;
    s   = descriptor<16>;
    ap  = descriptor<15,11:10>;
    tex = descriptor<14:12>;
    xn  = descriptor<4>;
    c   = descriptor<3>;
    b   = descriptor<2>;
    pxn = descriptor<0>;
    domain = '0000';

    walkstate.baseaddress.address = ZeroExtend(descriptor<8
56);
walkstate.istable = FALSE;

until walkstate.sdfstype != SDFType_Table;

if afe == '1' && ap<0> == '0' then
    fault.domain      = domain;
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState_UNKNOWN);

// Decode the TEX, C, B and S bits to produce target memory attribu
if tre == '1' then
    walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c,
elsif RemapRegsHaveResetValues() then
    walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);

```

```

    else
        walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

        walkstate.permissions.ap  = ap;
        walkstate.permissions.xn  = xn;
        walkstate.permissions.pxn = pxn;
        walkstate.domain = domain;
        walkstate.nG      = nG;

        if accdesc.ss == SS_Secure && ns == '0' then
            walkstate.baseaddress.paspace = PAS_Secure;
        else
            walkstate.baseaddress.paspace = PAS_NonSecure;

        return (fault, walkstate);

```

### Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```

// AArch32.S2IASize()
// =====
// Retrieve the number of bits containing the input address for stage 2

integer AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);

```

### Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```

// AArch32.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) s10)
    return 2 - UInt(s10);

```

### Library pseudocode for aarch32/translation/walk/AArch32.S2Walk

```

// AArch32.S2Walk()
// =====
// Traverse stage 2 translation tables in long format to obtain the final
// (FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParam
//                                         AccessDescriptor accdesc, Address)
//                                         FaultRecord fault = fault_in;

if walkparams.s10 IN {'1x'} || AArch32.S2InconsistentSL(walkparams)
    fault.statuscode = Fault_Translation;
    fault.level      = 1;
    return (fault, TTWState UNKNOWN);

// Input Address size
iasize      = AArch32.S2IASize(walkparams.t0sz);
startlevel  = AArch32.S2StartLevel(walkparams.s10);
levels      = FINAL_LEVEL - startlevel;
granulebits = TGxGranuleBits(walkparams.tgx);

```

```

stride      = granulebits - 3;

if !IsZero(VTTBR<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
constant integer baselsb = (iasize - (levels*stride + granulebits));
baseaddress.paspace = PAS_NonSecure;
baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb), 

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level      = startlevel;
walkstate.istable     = TRUE;
walkstate.memattrs   = WalkMemAttrs(walkparams.sh, walkparams.irgrn,
                                      walkparams.orgn);

bits(64) descriptor;
AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
AddressDescriptor walkaddress;

walkaddress.vaddress = ipa.vaddress;

if HCR2.CD == '1' then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

walkaddress.memattrs.shareability = EffectiveShareability(walkaddress);

integer msb_residual = iasize - 1;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;

    constant integer indexlsb = (FINAL_LEVEL - walkstate.level)*stride;
    constant integer indexmsb = msb_residual;
    bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>);

    walkaddress.paddress.address = walkstate.baseaddress.address OR
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);

    desctype = AArch32.DecodeDescriptorTypeID(descriptor, walkstate);

    case desctype of
        when DescriptorType_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault_AddressSize;
                return (fault, TTWState UNKNOWN);

            walkstate.baseaddress.address = ZeroExtend(descriptor<39:>;
            walkstate.level = walkstate.level + 1;

```

```

        msb_residual = indexlsb - 1;

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, TTWState_UNKNOWN);

        when DescriptorType_Leaf
            walkstate.istable = FALSE;

until desctype IN {DescriptorType_Leaf};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, TTWState_UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState_UNKNOWN);

// Unpack the descriptor into address and upper and lower block att
constant integer indexlsb = (FINAL_LEVEL - walkstate.level)*stride
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:

walkstate.permissions.s2ap = descriptor<7:6>;
walkstate.permissions.s2xn = descriptor<54>;
if IsFeatureImplemented(FEAT_XNX) then
    walkstate.permissions.s2xnx = descriptor<53>;
else
    walkstate.permissions.s2xnx = '0';

memattr = descriptor<5:2>;
sh      = descriptor<9:8>;
s2aarch64 = FALSE;
walkstate.memattrs  = S2DecodeMemAttrs(memattr, sh, s2aarch64);
walkstate.contiguous = descriptor<52>;

return (fault, walkstate);

```

## Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```

// AArch32.TranslationSizeSD()
// =====
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFType sdftype)
    integer tsize;
    case sdftype of
        when SDFType_SmallPage          tsize = 12;
        when SDFType_LargePage          tsize = 16;
        when SDFType_Section           tsize = 20;
        when SDFType_Supersection     tsize = 24;

    return tsize;

```

## Library pseudocode for aarch32/translation/walk/ RemapRegsHaveResetValues

```
// RemapRegsHaveResetValues()  
// =====  
  
boolean RemapRegsHaveResetValues();
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.GetS1TTWParams

```
// AArch32.GetS1TTWParams()  
// =====  
// Returns stage 1 translation table walk parameters from respective co  
// System registers.  
  
S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)  
    S1TTWParams walkparams;  
  
    case regime of  
        when Regime EL2 walkparams = AArch32.S1TTWParamsEL2();  
        when Regime EL10 walkparams = AArch32.S1TTWParamsEL10(va);  
        when Regime EL30 walkparams = AArch32.S1TTWParamsEL30(va);  
  
    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()  
// =====  
// Gather walk parameters for stage 2 translation  
  
S2TTWParams AArch32.GetS2TTWParams()  
    S2TTWParams walkparams;  
  
    walkparams.tgx  = TGx_4KB;  
    walkparams.s   = VTCR.S;  
    walkparams.t0sz = VTCR.T0SZ;  
    walkparams.sl0  = VTCR.SL0;  
    walkparams.irgn = VTCR.IRGN0;  
    walkparams.orgn = VTCR.ORGNO;  
    walkparams.sh   = VTCR.SH0;  
    walkparams.ee   = HSCTRL.EE;  
    walkparams.ptw   = HCR.PTW;  
    walkparams.vm   = HCR.VM OR HCR.DC;  
  
    // VTCR.S must match VTCR.T0SZ[3]  
    if walkparams.s != walkparams.t0sz<3> then  
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable);  
  
    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.GetVRange

```
// AArch32.GetVRange()
// =====
// Select the translation base address for stage 1 long-descriptor walk

VARange AArch32.GetVRange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    constant integer lo_iasize = AArch32.S1IASIZE(t0sz);
    // Upper range Input Address size
    constant integer up_iasize = AArch32.S1IASIZE(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange LOWER else VARange UNKNOWN;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange UPPER else VARange UNKNOWN;
    elsif IsZero(va<31:lo_iasize) then
        return VARange LOWER;
    elsif IsOnes(va<31:up_iasize) then
        return VARange UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime EL30 return SCTLR_S.C == '1';
        when Regime EL2 return HSCTLR.C == '1';
        when Regime EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.C == '1');
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime EL30 return SCTLR_S.I == '1';
        when Regime EL2 return HSCTLR.I == '1';
        when Regime EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.I == '1');
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.S1TTWParamsEL10

```

// AArch32.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sctlr;

    if ELUsingAArch32(EL3) then
        ttbcr = TTBCR_NS;
        ttbcr2 = TTBCR2_NS;
        sctlr = SCTLR_NS;
        mair = MAIR1_NS:MAIR0_NS;
        sif = SCR.SIF;
    else
        ttbcr = TTBCR;
        ttbcr2 = TTBCR2;
        sctlr = SCTLR;
        mair = MAIR1:MAIR0;
        sif = if HaveEL(EL3) then SCR_EL3.SIF else '0';

    assert ttbcr.EAE == '1';
S1TTWParams walkparams;

    walkparams.t0sz = ttbcr.T0SZ;
    walkparams.t1sz = ttbcr.T1SZ;
    walkparams.ee = sctlr.EE;
    walkparams.wxn = sctlr.WXN;
    walkparams.uwxn = sctlr.UWXN;
    walkparams.ntlsmd = if IsFeatureImplemented(FEAT_LSMAOC) then sctlr
    walkparams.mair = mair;
    walkparams.sif = sif;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange LOWER then
        walkparams.sh = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then t
    else
        walkparams.sh = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then t

    return walkparams;

```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh = HTCR.ORGN0;
    walkparams.hpd = if IsFeatureImplemented(FEAT_AA32HPD) then HTCR.HPD;
    walkparams.ee = HSCTRL.EE;
    walkparams.wxn = HSCTRL.WXN;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = HSCTRL.nTLSMD;
    else
        walkparams.ntlsmd = '1';
    walkparams.mair = HMAIR1:HMAIRO;
    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/ AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// =====
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = TTBCR_S.T0SZ;
    walkparams.t1sz = TTBCR_S.T1SZ;
    walkparams.ee = SCTLR_S.EE;
    walkparams.wxn = SCTLR_S.WXN;
    walkparams.uwxn = SCTLR_S.UWXN;
    walkparams.ntlsmd = if IsFeatureImplemented(FEAT_LSMAOC) then SCTLR.nTLSMD;
    walkparams.mair = MAIR1_S:MAIRO_S;
    walkparams.sif = SCR.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGN0;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then
            else '0');
    else
        walkparams.sh = TTBCR_S.SH1;
        walkparams.irgn = TTBCR_S.IRGN1;
```

```

walkparams.orgn = TTBCR_S.ORGN1;
walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then T
else '0');

return walkparams;

```

## Library pseudocode for aarch64/debug/brbe/BRBCycleCountingEnabled

```

// BRBCycleCountingEnabled()
// =====
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

boolean BRBCycleCountingEnabled()
    if HaveEL(EL2) && BRBCR_EL2.CC == '0' then return FALSE;
    if BRBCR_EL1.CC == '0' then return FALSE;
    return TRUE;

```

## Library pseudocode for aarch64/debug/brbe/BRBEBRANCH

```

// BRBEBRANCH()
// =====
// Called to write branch record for the following branches when BRB is
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

BRBEBRANCH(BranchType br_type, boolean cond, bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, co
        bits(6) branch_type;
        case br_type of
            when BranchType\_DIR
                branch_type = if cond then '001000' else '000000';
            when BranchType\_INDIR                      branch_type = '000001';
            when BranchType\_DIRCALL                   branch_type = '000010';
            when BranchType\_INDCALL                  branch_type = '000011';
            when BranchType\_RET                     branch_type = '000101';
            otherwise                                Unreachable();
        end;

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if IsFeatureImplemented(FEAT_TME) then BRBF
        bit transactional = if IsFeatureImplemented(FEAT_TME) && TSTATE
        bits(2) el = PSTATE.EL;
        bit mispredict = if BRBEMispredictAllowed() && BranchMispre
UpdateBranchRecordBuffer\(ccu, cc, lastfailed, transactional, bra
        '11', PC64, target\_address\);

        BRBFCR\_EL1.LASTFAILED = '0';

        PMUEvent\(PMU\_EVENT\_BRB\_FILTRATE\);

    return;

```

## Library pseudocode for aarch64/debug/brbe/BRBEBRANCHONISB

```
// BRBEBRANCHONISB()
// =====
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

boolean BRBEBRANCHONISB()
    return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records"
```

## Library pseudocode for aarch64/debug/brbe/BRBEDebugStateExit

```
// BRBEDebugStateExit()
// =====
// Called to write Debug state exit branch record when BRB is active.

BRBEDebugStateExit(bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, s
        bits(6) branch_type = '111001';
        bit ccu = '1';
        bits(14) cc = Zeros(14);
        bit lastfailed = if IsFeatureImplemented(FEAT_TME) then BRBFRCR_
        bit transactional = '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, bra
            '01', Zeros(64), target_address);

        BRBFRCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTERATE);

    return;
```

## Library pseudocode for aarch64/debug/brbe/BRBEECEPTION

```
// BRBEECEPTION()
// =====
// Called to write exception branch record when BRB is active.

BRBEECEPTION(ExceptionRecord erec, boolean source_valid,
            bits(64) source_address_in,
            bits(64) target_address_in, bits(2) target_el,
            boolean trappedsyscallinst)
    bits(64) target_address = target_address_in;
    Exception except = erec.exception_type;
    bits(25) iss = erec.syndrome;
    case target_el of
        when EL3
            if !IsFeatureImplemented(FEAT_BRBEV1P1) || (MDCR_EL3.E3BREC
                return;
        when EL2  if BRBCR_EL2.EXCEPTION == '0' then return;
        when EL1  if BRBCR_EL1.EXCEPTION == '0' then return;
```

```

boolean target_valid = BranchRecordAllowed(target_el);

if source_valid || target_valid then
  bits(6) branch_type;
  case except of
    when Exception_Uncategorized
    when Exception_WFxTrap
    when Exception_CP15RTTrap
    when Exception_CP15RRTTrap
    when Exception_CP14RTTrap
    when Exception_CP14DTTrap
    when Exception_AdvSIMDFPAccessTrap
    when Exception_FPIDTrap
    when Exception_PACTrap
    when Exception_TSTARTAccessTrap
    when Exception_CP14RRTTrap
    when Exception_BranchTarget
    when Exception_IllegalState
    when Exception_SupervisorCall
      if !trappedsyscallinst then
        else
    when Exception_HypervisorCall
    when Exception_MonitorCall
      if !trappedsyscallinst then
        else
    when Exception_SystemRegisterTrap
    when Exception_SystemRegister128Trap
    when Exception_SVEAccessTrap
    when Exception_SMEAccessTrap
    when Exception_ERetTrap
    when Exception_PACFail
    when Exception_InstructionAbort
    when Exception_PCAignment
    when Exception_DataAbort
    when Exception_NV2DataAbort
    when Exception_SPAlignment
    when Exception_FPTrappedException
    when Exception_SError
    when Exception_Breakpoint
    when Exception_SoftwareStep
    when Exception_Watchpoint
    when Exception_NV2Watchpoint
    when Exception_SoftwareBreakpoint
    when Exception_IRO
    when Exception_FIQ
    when Exception_MemCpyMemSet
    when Exception_GCSFail
      if iss<23:20> == '0000' then
        elseif iss<23:20> == '0001' then
        elseif iss<23:20> == '0010' then
        else
      otherwise

bit ccu;
bits(14) cc;
(ccu, cc) = BranchEncCycleCount();
bit lastfailed = if IsFeatureImplemented(FEAT_TME) then BRBFCR_
bit transactional = (if source_valid && IsFeatureImplemented(FEAT_TSTATE) then TSTATE.depth > 0 then '1' else '0');
bits(2) el = if target_valid then target_el else '00';

```

```

bit mispredict = '0';
bit sv = if source_valid then '1' else '0';
bit tv = if target_valid then '1' else '0';
bits(64) source_address = if source_valid then source_address_i

if !target_valid then
    target_address = Zeros(64);
else
    target_address = AArch64.BranchAddr(target_address, target_

UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional,
                           branch_type, el, mispredict,
                           sv:tv, source_address, target_address)

BRBFCR_EL1.LASTFAILED = '0';

PMUEvent(PMU_EVENT_BRB_FILTERATE);

return;

```

## Library pseudocode for aarch64/debug/brbe/BRBEEExceptionReturn

```

// BRBEEExceptionReturn()
// =====
// Called to write exception return branch record when BRB is active.

BRBEEExceptionReturn(bits(64) target_address_in, bits(2) source_el,
                     boolean source_valid, bits(64) source_address_in)
    bits(64) target_address = target_address_in;
    case source_el of
        when EL3
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3.E3BREC
                return;
        when EL2  if BRBCR_EL2.ERTN == '0' then return;
        when EL1  if BRBCR_EL1.ERTN == '0' then return;

    boolean target_valid = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        bits(6) branch_type = '000111';
        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if IsFeatureImplemented(FEAT_TME) then BRBFCR_
                        bit transactional = (if source_valid && IsFeatureImplemented(FEAT_TME)
                                         TSTATE.depth > 0 then '1' else '0');
        bits(2) el = if target_valid then PSTATE.EL else '00';
        bit mispredict = if (source_valid && BRBEMispredictAllowed() &&
                            BranchMispredict()) then '1' else '0';
        bit sv = if source_valid then '1' else '0';
        bit tv = if target_valid then '1' else '0';
        bits(64) source_address = if source_valid then source_address_i
        if !target_valid then
            target_address = Zeros(64);

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional,
                                   branch_type, el, mispredict,
                                   sv:tv, source_address, target_address)

```

```

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTERATE);

    return;

```

### Library pseudocode for aarch64/debug/brbe/BRBEFreeze

```

// BRBEFreeze()
// =====
// Generates BRBE freeze event.

BRBEFreeze()
    BRBFCR_EL1.PAUSED = '1';
    BRBTS_EL1 = GetTimestamp(BRBETimeStamp());

```

### Library pseudocode for aarch64/debug/brbe/BRBEISB

```

// BRBEISB()
// =====
// Handles ISB instruction for BRBE.

BRBEISB()
    boolean branch_conditional = FALSE;
    BRBEBRANCH(BranchType\_DIR, branch_conditional, PC64 + 4);

```

### Library pseudocode for aarch64/debug/brbe/BRBEMispredictAllowed

```

// BRBEMispredictAllowed()
// =====
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

boolean BRBEMispredictAllowed()
    if HaveEL(EL2) && BRBCR_EL2.MPRED == '0' then return FALSE;
    if BRBCR_EL1.MPRED == '0' then return FALSE;
    return TRUE;

```

### Library pseudocode for aarch64/debug/brbe/BRBETimeStamp

```

// BRBETimeStamp()
// =====
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
    if HaveEL(EL2) then
        TS_el2 = BRBCR_EL2.TS;
        if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable EL2T);
        case TS_el2 of
            when '00'

```

```

        // Falls out to check BRBCR_EL1.TS
when '01'
    return TimeStamp_Virtual;
when '10'
    assert IsFeatureImplemented(FEAT_ECV); // Otherwise Cor
                                         // removes this
    return TimeStamp_OffsetPhysical;
when '11'
    return TimeStamp_Physical;

TS_e11 = BRBCR_EL1.TS;
if TS_e11 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_e11 ==
    // Reserved value
    (-, TS_e11) = ConstrainUnpredictableBits(Unpredictable_EL1TIMES
case TS_e11 of
    when '01'
        return TimeStamp_Virtual;
    when '10'
        return TimeStamp_OffsetPhysical;
    when '11'
        return TimeStamp_Physical;
otherwise
    Unreachable();                      // ConstrainUnpredictableBits remo

```

### Library pseudocode for aarch64/debug/brbe/BRB\_IALL

```

// BRB_IALL()
// =====
// Called to perform invalidation of branch records

BRB_IALL()
    for i = 0 to GetBRBENumRecords() - 1
        Records_SRC[i] = Zeros(64);
        Records_TGT[i] = Zeros(64);
        Records_INF[i] = Zeros(64);

```

### Library pseudocode for aarch64/debug/brbe/BRB\_INJ

```

// BRB_INJ()
// =====
// Called to perform manual injection of branch records.

BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINF
                               BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBIN
                               BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID,
                               BRBTGTINJ_EL1.ADDRESS);
    BRBINFINJ_EL1 = bits(64) UNKNOWN;
    BRBSRCINJ_EL1 = bits(64) UNKNOWN;
    BRBTGTINJ_EL1 = bits(64) UNKNOWN;

    if ConstrainUnpredictableBool(Unpredictable_BRBFILTRATE) then PMUEv

```

### Library pseudocode for aarch64/debug/brbe/Branch

```
type BRBSRCType;
type BRBTGTTType;
type BRBINFTType;
```

### Library pseudocode for aarch64/debug/brbe/BranchEncCycleCount

```
// BranchEncCycleCount ()
// =====
// The first return result is '1' if either of the following is true, and
// - This is the first Branch record after the PE exited a Prohibited Region
// - This is the first Branch record after cycle counting has been enabled
// If the first return result is '0', the second return result is the error
// since the last branch.
// The format of this field uses a mantissa and exponent to express the
// - bits[7:0] indicate the mantissa M.
// - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(M)))
// A value of all ones in both the mantissa and exponent indicates the
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.
(bit, bits(14)) BranchEncCycleCount();
```

### Library pseudocode for aarch64/debug/brbe/BranchMispredict

```
// BranchMispredict ()
// =====
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.
boolean BranchMispredict();
```

### Library pseudocode for aarch64/debug/brbe/BranchRawCycleCount

```
// BranchRawCycleCount ()
// =====
// If the cycle count is known, the return result is the cycle count since
// the last branch.
integer BranchRawCycleCount();
```

### Library pseudocode for aarch64/debug/brbe/BranchRecordAllowed

```
// BranchRecordAllowed ()
// =====
// Returns TRUE if branch recording is allowed, FALSE otherwise.
boolean BranchRecordAllowed(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    if BRBFCR_EL1.PAUSED == '1' then
        return FALSE;
```

```

if el == EL3 && IsFeatureImplemented(FEAT_BRBEv1p1) then
    return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);

if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
    (CurrentSecurityState() == SS_Secure && MDCR_EL3.SBRBE == '01')
    return FALSE;

case el of
    when EL3  return FALSE;                      // FEAT_BRBEv1p1 not imp
    when EL2  return BRBCR_EL2.E2BRE == '1';
    when EL1  return BRBCR_EL1.E1BRE == '1';
    when EL0
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            return BRBCR_EL2.E0HBRE == '1';
        else
            return BRBCR_EL1.E0BRE == '1';

```

## Library pseudocode for aarch64/debug/brbe/Contents

```

// Contents of the Branch Record Buffer
//=====
array [0..63] of BRBSRCType Records_SRC;
array [0..63] of BRBTGTTtype Records_TGT;
array [0..63] of BRBINFTType Records_INF;

```

## Library pseudocode for aarch64/debug/brbe/FilterBranchRecord

```

// FilterBranchRecord()
// =====
// Returns TRUE if the branch record is not filtered out, FALSE otherwise

boolean FilterBranchRecord(BranchType br, boolean cond)
    case br of
        when BranchType_DIRCALL
            return BRBFCR_EL1.DIRCALL != BRBFCR_EL1.EnI;
        when BranchType_INDCALL
            return BRBFCR_EL1.INDCALL != BRBFCR_EL1.EnI;
        when BranchType_RET
            return BRBFCR_EL1.RTN != BRBFCR_EL1.EnI;
        when BranchType_DIR
            if cond then
                return BRBFCR_EL1.CONDDIR != BRBFCR_EL1.EnI;
            else
                return BRBFCR_EL1.DIRECT != BRBFCR_EL1.EnI;
        when BranchType_INDIR
            return BRBFCR_EL1.INDIRECT != BRBFCR_EL1.EnI;
            otherwise Unreachable();
    return FALSE;

```

## Library pseudocode for aarch64/debug/brbe/FirstBranchAfterProhibited

```

// FirstBranchAfterProhibited()
// =====
// Returns TRUE if branch recorded is the first branch after a prohibited
// FALSE otherwise.

FirstBranchAfterProhibited();

```

### Library pseudocode for aarch64/debug/brbe/GetBRBENumRecords

```

// GetBRBENumRecords()
// =====
// Returns the number of branch records implemented.

integer GetBRBENumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRB records";

```

### Library pseudocode for aarch64/debug/brbe/Getter

```

// Getter functions for branch records
// =====
// Functions used by MRS instructions that access branch records

BRBSRCType BRBSRC_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_SRC[record];
    else
        return Zeros(64);

BRBTGTTType BRBTGT_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_TGT[record];
    else
        return Zeros(64);

BRBINFTType BRBINF_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_INF[record];
    else
        return Zeros(64);

```

### Library pseudocode for aarch64/debug/brbe/ShouldBRBEFreeze

```

// ShouldBRBEFreeze()
// =====
// Returns TRUE if the BRBE freeze event conditions have been met, and

boolean ShouldBRBEFreeze()
    if !BranchRecordAllowed(PSTATE.EL) then return FALSE;

```

```

boolean check_e      = FALSE;
boolean check_cnten = FALSE;
boolean check_inten = FALSE;
boolean exclude_sync = FALSE;
boolean exclude_cyc  = TRUE;
boolean include_lo;
boolean include_hi;

if HaveEL(EL2) then
    include_lo = (BRBCR_EL1.FZP == '1');
    include_hi = (BRBCR_EL2.FZP == '1');
else
    include_lo = TRUE;
    include_hi = TRUE;

return PMUOverflowCondition(check_e, check_cnten, check_inten,
                           include_hi, include_lo, exclude_cyc,
                           exclude_sync);

```

## Library pseudocode for aarch64/debug/brbe/UpdateBranchRecordBuffer

```

// UpdateBranchRecordBuffer()
// =====
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit trans-
                        bits(6) branch_type, bits(2) el, bit mispredic-
                        bits(64) source_address, bits(64) target_address)

// Shift the Branch Records in the buffer
for i = GetBRBENumRecords() - 1 downto 1
    Records_SRC[i] = Records_SRC[i - 1];
    Records_TGT[i] = Records_TGT[i - 1];
    Records_INF[i] = Records_INF[i - 1];

    Records_INF[0].CCU          = ccu;
    Records_INF[0].CC           = cc;

    Records_INF[0].EL           = el;
    Records_INF[0].VALID        = valid;
    Records_INF[0].T             = transactional;
    Records_INF[0].LASTFAILED   = lastfailed;
    Records_INF[0].MPRED         = mispredict;
    Records_INF[0].TYPE          = branch_type;

    Records_SRC[0] = source_address;
    Records_TGT[0] = target_address;

return;

```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```

// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.
// Returns a pair of booleans, the first indicates if the match was successful
// the first value should be inverted because the breakpoint is configured
// to trigger on the inverse of the condition

```

```

(boolean, boolean) AArch64.BreakpointMatch(integer n, bits(64) vaddress
                                            integer size)
assert !ELUsingAArch32(S1TranslationRegime());
assert n < NumBreakpointsImplemented();

linking_enabled = (DBGBCR_EL1[n].BT IN {'0x11', '1xx1'} ||
                    HaveFeatABLE() && DBGBCR_EL1[n].BT2 == '1');

// A breakpoint that has linking enabled does not generate debug events
if linking_enabled then
    return (FALSE, FALSE);

enabled      = IsBreakpointEnabled(n);
linked       = DBGBCR_EL1[n].BT IN {'0x01'};
isbreakpnt   = TRUE;
linked_to    = FALSE;
lbnx         = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1[n].LBN;
linked_n     = UInt(lbnx : DBGBCR_EL1[n].LBN);
ssce         = if IsFeatureImplemented(FEAT_RME) then DBGBCR_EL1[n].SSCE else 0;
state_match  = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].PMC,
                                    DBGBCR_EL1[n].PMC, linked, linked_to,
                                    vaddress, accdesc);

(value_match, valid_mismatch) = AArch64.BreakpointValueMatch(n, vaddress,
                                                               isbreakpnt);

if HaveAArch32() && size == 4 then                                // Check second halfword
    // If the breakpoint address and BAS of an Address breakpoint match
    // second halfword of an instruction, but not the address of the instruction
    // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a debug event.
    (match_i, -) = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to,
                                                isbreakpnt);
    if !value_match && match_i then
        value_match = ConstrainUnpredictableBool(Unpredictable_BPMA(n));

if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
    // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', the
    // UNPREDICTABLE whether or not a Breakpoint debug event is generated
    // at the address DBGBCR_EL1[n]+2.
    if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMA(n));

match = value_match && state_match && enabled;
is_mismatch = valid_mismatch && state_match && enabled;

return (match, is_mismatch);

```

## Library pseudocode for aarch64/debug/breakpoint/ AArch64.BreakpointValueMatch

```

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked
// matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking
// "isbreakpnt" TRUE if this is a call from BreakpointMatch or from
// linked breakpoint.
integer n = n_in;
Constraint c;

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE
// no match or the breakpoint is mapped to another UNKNOWN implementation
if n >= NumBreakpointsImplemented() then
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented
                                              Unpredictable_BPNOTIMPL)
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE, FALSE);

// If this breakpoint is not enabled, it cannot generate a match.
// (This could also happen on a call from StateMatch for linking).
if !IsBreakpointEnabled(n) then return (FALSE, FALSE);

// If BT is set to a reserved type, behaves either as disabled or as
dbgtype = DBGBCR_EL1[n].BT;
bt2 = if HaveFeatABLE() then DBGBCR_EL1[n].BT2 else '0';

(c, bt2, dbgtype) = AArch64.ReservedBreakpointType(n, bt2, dbgtype)
if c == Constraint_DISABLED then return (FALSE, FALSE);
// Otherwise the value returned by ConstrainUnpredictableBits must
// be checked.

// Determine what to compare against.
match_addr      = (dbgtype IN {'0x0'});
mismatch        = (dbgtype IN {'010x'});
match_vmid      = (dbgtype IN {'10xx'});
match_cid       = (dbgtype IN {'001x'});
match_cid1      = (dbgtype IN {'101x', 'x11x'});
match_cid2      = (dbgtype IN {'11xx'});
linking_enabled = (dbgtype IN {'xx11', '1xx1'} || bt2 == '1');

// If this is a call from StateMatch, return FALSE if the breakpoint
// programmed with linking enabled.
if linked_to && !linking_enabled then
    return (FALSE, FALSE);

// If called from BreakpointMatch return FALSE for Linked context
if !linked_to && linking_enabled && !match_addr then
    return (FALSE, FALSE);

// If a linked breakpoint is linked to an address matching breakpoint
// the behavior is CONSTRAINED UNPREDICTABLE.
if linked_to && match_addr && isbreakpnt then
    if !ConstrainUnpredictableBool(Unpredictable_BPLINKEDADDRMATCH)
        return (FALSE, FALSE);

// A breakpoint programmed for address mismatch does not match in AArch32
if mismatch && UsingAArch32() then
    return (FALSE, FALSE);

boolean bvr_match = FALSE;
boolean bxvr_match = FALSE;
integer mask;

```

```

if HaveFeatABLE() then
    mask = UInt(DBGBCR_EL1[n].MASK);

    // If the mask is set to a reserved value, the behavior is CONstrained Unpredictable.
    if mask IN {1, 2} then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_RESERVED};
        case c of
            when Constraint_DISABLED return (FALSE, FALSE); // Disallowed.
            when Constraint_NONE mask = 0; // No constraint.
            // Otherwise the value returned by ConstrainUnpredictableInteger will be a not-reserved value.

    if mask != 0 then
        // When DBGBCR_EL1[n].MASK is a valid nonzero value, the behavior is constrained unpredictable if any of the following are true:
        // - DBGBCR_EL1[n].<BT2,BT> is programmed for a Context.
        // - DBGBCR_EL1[n].BAS is not '1111' and AArch32 is supported.
        if ((match_cid || match_cid1 || match_cid2) ||
            (DBGBCR_EL1[n].BAS != '1111' && HaveAArch32())) then
            if !ConstrainUnpredictableBool(Unpredictable_BPMASK) then
                return (FALSE, FALSE);
        else
            // A stand-alone mismatch of a single address is not supported.
            if mismatch then
                return (FALSE, FALSE);

    else
        mask = 0;

    // Do the comparison.
    if match_addr then
        boolean byte_select_match;
        integer byte = UInt(vaddress<1:0>);

        if HaveAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation context.
            assert byte IN {0,2}; // "vaddress" is always 0 or 2.
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0; // "vaddress" is always 0.
            byte_select_match = TRUE; // DBGBCR_EL1[n].BAS<byte> is always 0.

        // When FEAT_LVA3 is not implemented, if the DBGBVR_EL1[n].RESS bit is set and the sign extension of the MSB of DBGBVR_EL1[n].VA, it is UNPREDICTABLE.
        // appear to be included in the match.
        // If 'vaddress' is outside of the current virtual address space, it generates a Translation fault.
        constant integer dbgtop = DebugAddrTop();
        boolean unpredictable_ress = (dbgtop < 55 && !IsOnes(DBGBVR_EL1[n]<63:dbgtop>) && !IsZero(DBGBVR_EL1[n]<63:dbgtop>));
        unpredictable_ress = ConstrainUnpredictableBool(Unpredictable, unpredictable_ress);

        constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
        constant integer cmplsb = if mask > 2 then mask else 2;
        bvr_match = ((vaddress<cmpmsb:cmplsb> == DBGBVR_EL1[n]<cmpmsb:cmplsb>) && byte_select_match);
        if mask > 2 then
            // If masked bits of DBGBVR_EL1[n] are not zero, the behavior is CONstrained Unpredictable.
            constant integer masktop = mask - 1;

```

```

        if bvr_match && !IsZero(DBGBVR_EL1[n]<masktop:2>) then
            bvr_match = ConstrainUnpredictableBool(Unpredictable_BP);

    elseif match_cid then
        if IsInHost() then
            bvr_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
        else
            bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

    elseif match_cid1 then
        bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;

        if !IsFeatureImplemented(FEAT_VMID16) || VTCSR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
        else
            vmid      = VTTBR_EL2.VMID;
            bvr_vmid = DBGBVR_EL1[n]<47:32>;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<63:32>);

        bvr_match_valid = (match_addr || match_cid || match_cid1);
        bxvr_match_valid = (match_vmid || match_cid2);

        value_match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return (value_match, mismatch);

```

## Library pseudocode for aarch64/debug/breakpoint/ AArch64.ReservedBreakpointType

```

// AArch64.ReservedBreakpointType()
// =====
// Checks if the given DBGBCR<n>.EL1.BT2 and DBGBCR<n>.EL1.BT value is
// generate Constrained Unpredictable behavior, otherwise returns Constraint

(Constraint, bit, bits(4)) AArch64.ReservedBreakpointType(integer n, bit
    bit bt2          = bt2_in;
    bits(4) bt         = bt_in;
    boolean reserved = FALSE;
    context_aware = n >= (NumBreakpointsImplemented() - NumContextAware);

    if bt2 == '0' then
        // Context matching
        if !(bt IN {'0x0x'}) && !context_aware then
            reserved = TRUE;

        // EL2 extension
        if bt IN {'1xxx'} && !HaveEL(EL2) then

```

```

    reserved = TRUE;

    // Context matching
    if (bt IN {'011x', '11xx'} && !IsFeatureImplemented(FEAT_VHE) &&
        !IsFeatureImplemented(FEAT_Debugv8p2)) then
        reserved = TRUE;

    // Reserved
    if bt IN {'010x'} && !HaveFeatABLE() && !HaveAArch32EL(EL1) then
        reserved = TRUE;
    else
        // Reserved
        if !(bt IN {'0x0x'}) then
            reserved = TRUE;

    if reserved then
        Constraint c;
        (c, <bt2,bt>) = ConstrainUnpredictableBits(Unpredictable RESBPT);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bit UNKNOWN, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits m

    return (Constraint_NONE, bt2, bt);

```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

```

// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current
// context.

boolean AArch64.StateMatch(bits(2) ssc_in, bit ssce_in, bit hmc_in,
                           bits(2) pxc_in, boolean linked_in, integer 1
                           boolean isbreakpt, bits(64) vaddress, Access
if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0';

// "ssc_in", "ssce_in", "hmc_in", "pxc_in" are the control fields from
// the DBGBCR_EL1[n] or DBGWCR_EL1[n] register.
// "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
// "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1[n]
// or DBGWCR_EL1[n] register.
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "vaddress" is the program counter for a linked watchpoint or the
// AArch64.CheckBreakpoint for a linked breakpoint.
// "accdesc" describes the properties of the access being matched.
bits(2) ssc      = ssc_in;
bit ssce     = ssce_in;
bit hmc      = hmc_in;
bits(2) pxc      = pxc_in;
boolean linked  = linked_in;
integer linked_n = linked_n_in;

// If parameters are set to a reserved type, behaves as either disabled
Constraint c;
(c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc)
if c == Constraint_DISABLED then return FALSE;
// Otherwise the hmc, ssc, ssce, pxc values are either valid or the values
// from CheckValidStateMatch are valid.

```

```

EL3_match = HaveEL\(EL3\) && hmc == '1' && ssc<0> == '0';
EL2_match = HaveEL\(EL2\) && ((hmc == '1' && (ssc:pxc != '1000')) || 
EL1_match = pxc<0> == '1';
EL0_match = pxc<1> == '1';

boolean priv_match;
case accdesc.el of
    when EL3 priv_match = EL3_match;
    when EL2 priv_match = EL2_match;
    when EL1 priv_match = EL1_match;
    when EL0 priv_match = EL0_match;

// Security state match
boolean ss_match;
case ssce:ssc of
    when '000' ss_match = hmc == '1' || accdesc.ss != SS\_Root;
    when '001' ss_match = accdesc.ss == SS\_NonSecure;
    when '010' ss_match = (hmc == '1' && accdesc.ss == SS\_Root);
    when '011' ss_match = (hmc == '1' && accdesc.ss != SS\_Root) ||
    when '101' ss_match = accdesc.ss == SS\_Realm;

boolean linked_match = FALSE;
boolean is_linked_mismatch = FALSE;

if linked then
    // "linked_n" must be an enabled context-aware breakpoint unit.
    // then it is CONSTRAINED UNPREDICTABLE whether this gives no mismatch
    // linking, or linked_n is mapped to some UNKNOWN breakpoint that
    if !IsContextMatchingBreakpoint(linked_n) then
        (first_ctx_cmp, last_ctx_cmp) = ContextMatchingBreakpointRange(c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNPREDICTABLE};

    case c of
        when Constraint\_DISABLED return FALSE; // Disable
        when Constraint\_NONE linked = FALSE; // No linking
        // Otherwise ConstrainUnpredictableInteger returned a constraint

if linked then
    linked_to = TRUE;
    (linked_match, is_linked_mismatch) = AArch64.BreakpointValueMatch(priv_match, ss_match, linked_to);

return (priv_match && ss_match && (!linked ||
    (!is_linked_mismatch && linked_match) || (is_linked_mismatch && !linked)));

```

## Library pseudocode for aarch64/debug/breakpoint/DebugAddrTop

```

// DebugAddrTop()
// =====
// Returns the value for the top bit used in Breakpoint and Watchpoint

integer DebugAddrTop()
    if IsFeatureImplemented(FEAT_LVA3) then
        return 55;
    elseif IsFeatureImplemented(FEAT_LVA) then

```

```

        return 52;
    else
        return 48;

```

## Library pseudocode for aarch64/debug/breakpoint/ EffectiveMDSELR\_EL1\_BANK

```

// EffectiveMDSELR_EL1_BANK()
// =====
// Return the effective value of MDSELR_EL1.BANK.

bits(2) EffectiveMDSELR_EL1_BANK()
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented
    // then the field is RES0.
    integer num_bp = NumBreakpointsImplemented\(\);
    integer num_wp = NumWatchpointsImplemented\(\);
    if num_bp <= 16 && num_wp <= 16 then
        return '00';

    // At EL3, the Effective value of this field is zero if MDCR_EL3.EBWE
    // At EL2, the Effective value is zero if the Effective value of MDCR_EL2.EBWE
    // That is, if either MDCR_EL3.EBWE is 0 or MDCR_EL2.EBWE is 0.
    // At EL1, the Effective value is zero if the Effective value of MDSCR_EL1.EMBWE
    // That is, if any of MDCR_EL3.EBWE, MDCR_EL2.EBWE, or MDSCR_EL1.EMBWE
    if ((HaveEL\(EL3\) && MDCR_EL3.EBWE == '0') ||
        (PSTATE.EL != EL3 && EL2Enabled\(\) && MDCR_EL2.EBWE == '0') ||
        (PSTATE.EL == EL1 && MDSCR_EL1.EMBWE == '0')) then
        return '00';

    bits(2) bank = MDSELR_EL1.BANK;

    // Values are reserved depending on the number of breakpoints or watchpoints
    // implemented.
    if ((bank == '11' && num_bp <= 48 && num_wp <= 48) ||
        (bank == '10' && num_bp <= 32 && num_wp <= 32)) then
        // Reserved value
        (-, bank) = ConstrainUnpredictableBits\(Unpredictable\_RESMDSELR,
        // The value returned by ConstrainUnpredictableBits must be a reserved value
        return bank;

```

## Library pseudocode for aarch64/debug/breakpoint/IsBreakpointEnabled

```

// IsBreakpointEnabled()
// =====
// Returns TRUE if the effective value of DBGBCR_EL1[n].E is '1', and FALSE
// otherwise.

boolean IsBreakpointEnabled(integer n)
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint\(\) && !SelfHostedExtendedBPW\(\)) ||
        (HaltOnBreakpointOrWatchpoint\(\) && EDSCR2.EHBWE == '0'))) then
        return FALSE;

    return DBGBCR_EL1[n].E == '1';

```

## Library pseudocode for aarch64/debug/breakpoint/ SelfHostedExtendedBPWPEnabled

```
// SelfHostedExtendedBPWPEnabled()
// =====
// Returns TRUE if the extended breakpoints and watchpoints are enabled
// from a self-hosted debug perspective.

boolean SelfHostedExtendedBPWPEnabled()
    if NumBreakpointsImplemented\(\) <= 16 && NumWatchpointsImplemented\(\)
        return FALSE;

    if ((HaveEL\(EL3\) && MDCR_EL3.EBWE == '0') ||
        (EL2Enabled\(\) && MDCR_EL2.EBWE == '0')) then
        return FALSE;

    return MDSCR_EL1.EMBWE == '1';
```

## Library pseudocode for aarch64/debug/ebep/CheckForPMUException

```
// CheckForPMUException()
// =====
// Take a PMU exception if enabled, permitted, and unmasked.

CheckForPMUException()
    boolean enabled;
    bits(2) target_el;
    boolean pmu_exception;
    (enabled, target_el) = PMUExceptionEnabled\(\);
    if !enabled || PMUExceptionMasked\(\) then
        pmu_exception = FALSE;
    elseif IsFeatureImplemented(FEAT_SEBEP) && PSTATE.PPEND == '1' then
        pmu_exception = TRUE;
    else
        boolean check_cnten = FALSE;
        boolean check_e = TRUE;
        boolean check_inten = TRUE;
        boolean include_lo = TRUE;
        boolean include_hi = TRUE;
        boolean exclude_cyc = FALSE;
        boolean exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        pmu_exception = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                         include_hi, include_lo,
                                         exclude_cyc, exclude_sync);

    if pmu_exception then
        TakePMUException(target_el);
```

## Library pseudocode for aarch64/debug/ebep/ExceptionReturnPPEND

```
// ExceptionReturnPPEND()
// =====
// Sets ShouldSetPPEND to the value to write to PSTATE.PPEND
// on an exception return.
// This function is called before any change in Exception level.

ExceptionReturnPPEND(bits(64) spsr)
```

```

boolean enabled_at_source = FALSE;
boolean masked_at_source = FALSE;
if spsr<33> == '1' then // SPSR.PPEND
    (enabled_at_source, -) = PMUEExceptionEnabled();
    masked_at_source = PMUEExceptionMasked();

bits(2) target_el;
if IllegalExceptionReturn(spsr) then
    target_el = PSTATE.EL;
else
    boolean valid;
    (valid, target_el) = ELFfromSPSR(spsr);
    assert valid;

boolean masked_at_dest = PMUEExceptionMasked(target_el, spsr<32>);

if enabled_at_source && masked_at_source && !masked_at_dest then
    PSTATE.PPEND = '1';
    ShouldSetPPEND = FALSE;
    // PSTATE.PPEND will not be changed again by this instruction

// If PSTATE.PPEND has not been set by this function, ShouldSetPPEND
// unchanged, meaning PSTATE.PPEND might either be set by the current
// causing a counter overflow, or cleared to zero at the end of instruction

return;

```

### **Library pseudocode for aarch64/debug/ebep/ IsSupportingPMUSynchronousMode**

```

// IsSupportingPMUSynchronousMode()
// =====
// Returns TRUE if the event support synchronous mode,
// and FALSE otherwise.

boolean IsSupportingPMUSynchronousMode(bits(16) pmuevent);

```

### **Library pseudocode for aarch64/debug/ebep/PMUEExceptionEnabled**

```

// PMUEExceptionEnabled()
// =====
// The first return value is TRUE if the PMU exception is enabled, and FALSE
// The second return value is the target Exception level for an enabled PMU

(boolean, bits(2)) PMUEExceptionEnabled()

if !IsFeatureImplemented(FEAT_EBEP) then
    return (FALSE, bits(2) UNKNOWN);

boolean enabled;
bits(2) target = bits(2) UNKNOWN;

if HaveEL(EL3) && MDCR_EL3.PMEE != '01' then
    enabled = MDCR_EL3.PMEE == '11';
    if enabled then target = EL3;

elseif EL2Enabled() && MDCR_EL2.PMEE != '01' then
    enabled = MDCR_EL2.PMEE == '11';

```

```

        if enabled then target = EL2;

    else
        bits(2) pmee_el1 = PMEGR_EL1.PMEE;
        if pmee_el1 == '01' then                                // Reserved value
            Constraint c;
            (c, pmee_el1) = ConstrainUnpredictableBits(Unpredictable RE
assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then pmee_el1 = '10';
        // Otherwise the value returned by ConstrainUnpredictableBi
        // a non-reserved value

        enabled = pmee_el1 == '11';
        if enabled then
            target = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 els

return (enabled, target);

```

### Library pseudocode for aarch64/debug/ebep/PMUExceptionMasked

```

// PMUExceptionMasked()
// =====
// Return TRUE if the PMU Exception is masked at the current Exception
// FALSE otherwise.

boolean PMUExceptionMasked()
    return PMUExceptionMasked(PSTATE.EL, PSTATE.PM);

// PMUExceptionMasked()
// =====
// Return TRUE if the PMU Exception is masked at the specified Exception
// and by the value of PSTATE.PM, and FALSE otherwise.

boolean PMUExceptionMasked(bits(2) el, bit pm)
    assert IsFeatureImplemented(FEAT_EBEP);

    (-, target) = PMUExceptionEnabled();

    if Halted() then
        return TRUE;
    elseif UInt(target) < UInt(el) then
        return TRUE;
    elseif el == EL2 && target == EL2 && MDCR_EL2.PMEE != '11' then
        return TRUE;
    elseif target == el && (PMEGR_EL1.KPME == '0' || pm == '1') then
        return TRUE;

    return FALSE;

```

### Library pseudocode for aarch64/debug/ebep/PMUIinterruptEnabled

```

// PMUIinterruptEnabled()
// =====
// Return TRUE if the PMU interrupt request (PMUIIRQ) is enabled, FALSE

boolean PMUIinterruptEnabled()
    if !IsFeatureImplemented(FEAT_EBEP) then

```

```

        return TRUE;

boolean enabled;

if HaveEL(EL3) && MDCR_EL3.PMEE != '01' then
    enabled = MDCR_EL3.PMEE == '00';

elsif EL2Enabled() && MDCR_EL2.PMEE != '01' then
    enabled = MDCR_EL2.PMEE == '00';

else
    bits(2) pmee_el1 = PMECR_EL1.PMEE;
    if pmee_el1 == '01' then                                // Reserved value
        Constraint c;
        (c, pmee_el1) = ConstrainUnpredictableBits(Unpredictable RE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then pmee_el1 = '10';
        // Otherwise the value returned by ConstrainUnpredictableBi
        // a non-reserved value
    enabled = pmee_el1 == '00';

return enabled;

```

### **Library pseudocode for aarch64/debug/ebep/TakePMUException**

```

// TakePMUException()
// =====
// Takes a PMU exception.

TakePMUException(bits(2) target_el)
    ExceptionRecord except = ExceptionSyndrome(Exception_PMU);
    bit synchronous = if IsFeatureImplemented(FEAT_SEBEP) then PSTATE.P
    except.syndrome = Zeros(24) : synchronous;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

### **Library pseudocode for aarch64/debug/ebep/inst\_addr\_executed**

```
bits(64) inst_addr_executed;
```

### **Library pseudocode for aarch64/debug/ebep/sync\_counter\_overflowed**

```
boolean sync_counter_overflowed;
```

### **Library pseudocode for aarch64/debug/enables/ AArch64.GenerateDebugExceptions**

```

// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D)

```

### **Library pseudocode for aarch64/debug/enables/ AArch64.GenerateDebugExceptionsFrom**

```

// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState ss)
    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS_Secure || IsSecure(HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')));
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        enabled = MDCR_EL3.SDD == '0';
        if from_el == EL0 && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;

```

### **Library pseudocode for aarch64/debug/ite/AArch64.TRCIT**

```

// AArch64.TRCIT()
// =====
// Determines whether an Instrumentation trace packet should
// be generated and then generates an instrumentation trace packet
// containing the value of the register passed as an argument

AArch64.TRCIT(bits(64) Xt)
    ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);

```

### **Library pseudocode for aarch64/debug/ite/TraceInstrumentation**

```

// TraceInstrumentation()
// =====
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

```

```
TraceInstrumentation(bits(64) Xt);
```

## Library pseudocode for aarch64/debug/pmu/AArch64.ClearEventCounters

```
// AArch64.ClearEventCounters()
// =====
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_EL0[idx] = Zeros(64);
```

## Library pseudocode for aarch64/debug/pmu/ AArch64.GetNumEventCountersAccessible

```
// AArch64.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current EL level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!IsFeatureImplemented(FEAT_HPMN0) &&
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable_PMUEVENTS));
    else
        n = total_counters;

    return n;
```

## Library pseudocode for aarch64/debug/pmu/AArch64.IncrementCycleCounter

```
// AArch64.IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

AArch64.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return;
    bit d = PMCR_EL0.D;      // Check divide-by-64
    bit lc = PMCR_EL0.LC;
    boolean lc_enabled;
    (lc_enabled, -) = PMUEExceptionEnabled();
    lc = if lc_enabled then '1' else lc;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles() then return;

    integer old_value = UInt(PMCCNTR_EL0);
    integer new_value = old_value + 1;
```

```

PMCCNTR_EL0 = new_value<63:0>;
constant integer ovflw = if HaveAArch32\(\) && lc == '1' then 64 else
if old_value<64:ovflw> != new_value<64:ovflw> then
    PMOVSET_EL0.C = '1';
    PMOVCLR_EL0.C = '1';

return;

```

## Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```

// AArch64.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    bit lp;
    if IsFeatureImplemented(FEAT_PMUv3p5) then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        boolean pmuexception_enabled;
        (pmuexception_enabled, -) = PMUExceptionEnabled\(\);
        if pmuexception_enabled then
            lp = '1';
        else
            lp = if PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.HLP;
    else
        lp = '0';
    PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
    constant integer ovflw = if lp == '1' then 64 else 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSET_EL0<idx> = '1';
        PMOVCLR_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if (idx<0> == '0' && idx + 1 < GetNumEventCounters\(\) &&
            (!IsFeatureImplemented(FEAT_PMUv3p5) || lp == '0')) then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
    if (IsFeatureImplemented(FEAT_SEBEP) &&
        IsSupportingPMUSynchronousMode(PMEVTPER_EL0[idx].evtCount) &&
        PMINTENSET_EL1[idx] == '1' && PMOVCLR_EL0[idx] == '1' &&
        SyncCounterOverflowed = TRUE;

    return;

```

## Library pseudocode for aarch64/debug/pmu/AArch64.PMUCycle

```

// AArch64.PMUCycle()
// =====
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the

```

```

// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;

    PMUEvent (PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if (!IsFeatureImplemented(FEAT_SEBEP) || PMEVTYPEPER_EL0[idx]
                !IsSupportingPMUSynchronousMode (PMEVTYPEPER_EL0[idx] .
                CountPMUEvents(idx)) then
                integer accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter (idx, accumulated);
                PMUEventAccumulator[idx] = 0;
            AArch64.IncrementCycleCounter();
            CheckForPMUOverflow();

```

### Library pseudocode for aarch64/debug/pmu/AArch64.PMUSwIncrement

```

// AArch64.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC_ELO.

AArch64.PMUSwIncrement (bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent (PMU_EVENT_SW_INCR, 1, idx);

```

### Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```

// CollectContextIDR1()
// =====

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';

```

### Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```

// CollectContextIDR2()
// =====

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';

```

## Library pseudocode for aarch64/debug/statisticalprofiling/ CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled())
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;

    bits(2) PCT_e11;
    if !IsFeatureImplemented(FEAT_ECV) then
        PCT_e11 = '0':PMSCR_EL1.PCT<0>; // PCT<0>
    else
        PCT_e11 = PMSCR_EL1.PCT;
        if PCT_e11 == '10' then
            // Reserved value
            (-, PCT_e11) = ConstrainUnpredictableBits(Unpredictable_PMS);
    if EL2Enabled() then
        bits(2) PCT_e12;
        if !IsFeatureImplemented(FEAT_ECV) then
            PCT_e12 = '0':PMSCR_EL2.PCT<0>; // PCT<0>;
        else
            PCT_e12 = PMSCR_EL2.PCT;
            if PCT_e12 == '10' then
                // Reserved value
                (-, PCT_e12) = ConstrainUnpredictableBits(Unpredictable_PMS);
    case PCT_e12 of
        when '00'
            return if IsInHost() then TimeStamp_Physical else TimeStamp_Uncertain;
        when '01'
            if owning_el == EL2 then return TimeStamp_Physical;
        when '11'
            assert IsFeatureImplemented(FEAT_ECV); // FEAT_ECV
            if owning_el == EL1 && PCT_e11 == '00' then
                return if IsInHost() then TimeStamp_Physical else TimeStamp_Uncertain;
            else
                return TimeStamp_OffsetPhysical;
        otherwise
            Unreachable();
```

```

    case PCT_el1 of
        when '00' return if IsInHost() then TimeStamp_Physical else Time
        when '01' return TimeStamp_Physical;
        when '11'
            assert IsFeatureImplemented(FEAT_ECV);
            return TimeStamp_OffsetPhysical;
    otherwise Unreachable();

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/OpType**

```

// OpType
// =====
// Types of operation filtered by SPECollectRecord().

enumeration OpType {
    OpType_Load,           // Any memory-read operation other than atomics
    OpType_Store,          // Any memory-write operation, including atomic
    OpType_LoadAtomic,     // Atomics with return, compare-and-swap and sw
    OpType_Branch,         // Software write to the PC
    OpType_Other           // Any other class of operation
};

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ ProfilingBufferEnabled**

```

// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
    if !IsFeatureImplemented(FEAT_SPE) then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    bits(2) state_bits;
    if IsFeatureImplemented(FEAT_RME) then
        state_bits = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
    else
        state_bits = '0' : SCR_EL3.NS;

    boolean state_match;
    case owning_ss of
        when SS_Secure   state_match = state_bits == '00';
        when SS_NonSecure state_match = state_bits == '01';
        when SS_Realm    state_match = state_bits == '11';
    return (!ELUsingAArch32(owning_el) && state_match &&
            PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ ProfilingBufferOwner**

```

// ProfilingBufferOwner()
// =====

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

```

```

if HaveEL(EL3) then
    bits(3) state_bits;
    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3.<NSPBE,NSPB>;
        if (state_bits IN {'10x'} ||
            (!IsFeatureImplemented(FEAT_SEL2) && state_bits IN {'00x'}) ||
            // Reserved value
            (-, state_bits) = ConstrainUnpredictableBits(UnpredictableBits));
    else
        state_bits = '0' : MDCR_EL3.NSPB;

    case state_bits of
        when '00x' owning_ss = SS_Secure;
        when '01x' owning_ss = SS_NonSecure;
        when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;

    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
    else
        owning_el = EL1;

    return (owning_ss, owning_el);

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ ProfilingSynchronizationBarrier**

```

// ProfilingSynchronizationBarrier()
// =====
// Barrier to ensure that all existing profiling data has been formatted
// addresses have been translated such that writes to the profiling buffer
// A following DSB completes when writes to the profiling buffer have completed
ProfilingSynchronizationBarrier();

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEAddByteToRecord**

```

// SPEAddByteToRecord()
// =====
// Add one byte to a record and increase size property appropriately.

SPEAddByteToRecord(bits(8) b)
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[SPERecordSize] = b;
    SPERecordSize = SPERecordSize + 1;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEAddPacketToRecord**

```

// SPEAddPacketToRecord()
// =====
// Add passed header and payload data to the record.

```

```

// Payload must be a multiple of 8.

SPEAddPacketToRecord(bits(2) header_hi, bits(4) header_lo,
                     bits(N) payload)
    assert N MOD 8 == 0;
    bits(2) sz;
    case N of
        when 8 sz = '00';
        when 16 sz = '01';
        when 32 sz = '10';
        when 64 sz = '11';
        otherwise Unreachable\(\);
    end;

    bits(8) header = header_hi:sz:header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1
        SPEAddByteToRecord(payload<i*8+7:i*8>);

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEBranch

```

// SPEBranch()
// =====
// Called on every branch if SPE is present. Maintains previous branch
// and branch related SPE functionality.

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional,
          boolean is_isb = FALSE;
          SPEBranch(target, branch_type, conditional, taken_flag, is_isb);

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional,
          boolean is_isb)
    // If the PE implements branch prediction, data about (mis)prediction
    // through the PMU events.

    boolean collect_prev_br;
    boolean collect_prev_br_eret = boolean IMPLEMENTATION_DEFINED "SPE"
    boolean collect_prev_br_exception = boolean IMPLEMENTATION_DEFINED
    boolean collect_prev_br_isb = boolean IMPLEMENTATION_DEFINED "SPE p"
    case branch_type of
        when BranchType EXCEPTION
            collect_prev_br = collect_prev_br_exception;
        when BranchType ERET
            collect_prev_br = collect_prev_br_eret;
        otherwise
            collect_prev_br = !is_isb || collect_prev_br_isb;

    // Implements previous branch target functionality
    if (taken_flag && !IsZero(PMSIDR_EL1.PBT) && StatisticalProfilingEnab
        collect_prev_br) then

        if SPESampleInFlight then
            // Save the target address for it to be added to record.
            bits(64) previous_target = SPESamplePreviousBranchAddress;
            SPESampleAddress[SPEAddrPosPrevBranchTarget]<63:0> = previous_target;
            boolean previous_branch_valid = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = previous_branch_valid;
            SPESamplePreviousBranchAddress<55:0> = target<55:0>;

```

```

bit ns;
bit nse;
case CurrentSecurityState() of
    when SS_Secure
        ns = '0';
        nse = '0';
    when SS_NonSecure
        ns = '1';
        nse = '0';
    when SS_Realm
        ns = '1';
        nse = '1';
    otherwise Unreachable();
SPESamplePreviousBranchAddress<63> = ns;
SPESamplePreviousBranchAddress<60> = nse;
SPESamplePreviousBranchAddress<62:61> = PSTATE.EL;
SPESamplePreviousBranchAddressValid = TRUE;

if !StatisticalProfilingEnabled() then
    if taken_flag then
        // Invalidate previous branch address, if profiling is disabled
        // or prohibited.
        SPESamplePreviousBranchAddressValid = FALSE;
    return;

if SPESampleInFlight then
    is_direct = branch_type IN {BranchType_DIR, BranchType_DIRCALL};
    SPESampleClass = '10';
    SPESampleSubclass<1> = if is_direct then '0' else '1';
    SPESampleSubclass<0> = if conditional then '1' else '0';
    SPESampleOpType = OpType_Branch;

    // Save the target address.
    if taken_flag then
        SPESampleAddress[SPEAddrPosBranchTarget]<55:0> = target<55:>

        bit ns;
        bit nse;
        case CurrentSecurityState() of
            when SS_Secure
                ns = '0';
                nse = '0';
            when SS_NonSecure
                ns = '1';
                nse = '0';
            when SS_Realm
                ns = '1';
                nse = '1';
            otherwise Unreachable();
SPESampleAddress[SPEAddrPosBranchTarget]<63> = ns;
SPESampleAddress[SPEAddrPosBranchTarget]<60> = nse;
SPESampleAddress[SPEAddrPosBranchTarget]<62:61> = PSTATE.EL;
SPESampleAddressValid[SPEAddrPosBranchTarget] = TRUE;

SPESampleEvents<6> = if !taken_flag then '1' else '0';

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferFilled

```
// SPEBufferFilled()
// =====
// Deal with a full buffer event.

SPEBufferFilled()
    if IsZero(PMBSR_EL1.S) then
        PMBSR_EL1.S = '1';                                // Assert PMBIRQ
        PMBSR_EL1.EC = '000000';                          // Other buffer mana...
        PMBSR_EL1.MSS = ZeroExtend('000001', 16);      // Set buffer full e...
    PMUEvent(PMU_EVENT_SAMPLE_WRAP);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferIsFull

```
// SPEBufferIsFull()
// =====
// Return true if another full size sample record would not fit in the
// profiling buffer.

boolean SPEBufferIsFull()
    integer write_pointer_limit = UInt(PMLIMITR_EL1.LIMIT:Zeros(12));
    integer current_write_pointer = UInt(PMBPTR_EL1);
    integer record_max_size = 1<<UInt(PMSIDR_EL1.MaxValue);
    return current_write_pointer > (write_pointer_limit - record_max_si...
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPECollectRecord

```
// SPECollectRecord()
// =====
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType
    assert StatisticalProfilingEnabled());

    bits(64) mask = 0xAA<63:0>;                                // Bits
    bits(64) e;
    bits(64) m;
    if IsFeatureImplemented(FEAT_SVE) then mask<18:17> = '11'; // Prec...
    if IsFeatureImplemented(FEAT_TME) then mask<16> = '1';
    if IsFeatureImplemented(FEAT_SPEv1p1) then mask<11> = '1'; // Aliq...
    if IsFeatureImplemented(FEAT_SPEv1p2) then mask<6> = '1'; // Not ...
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask<10:8,4,2> = '1111';
    else
        bits(5) impdef_mask;
        impdef_mask = bits(5) IMPLEMENTATION_DEFINED "SPE mask 10:8,4,2";
        mask<10:8,4,2> = impdef_mask;

    mask<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE mask 63:48";
    mask<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE mask 31:24";
    mask<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE mask 15:12";

    e = events AND mask;
```

```

boolean is_rejected_nevent = FALSE;
boolean is_nevt;
// Filtering by inverse event
if IsFeatureImplemented(FEAT_SPEv1p2) then
    m = PMSNEVFR_EL1 AND mask;
    is_nevt = IsZero(e AND m);
    if PMSFCR_EL1.FnE == '1' then
        // Inverse filtering by event is enabled
        if !IsZero(m) then
            // Not UNPREDICTABLE case
            is_rejected_nevent = !is_nevt;
        else
            is_rejected_nevent = ConstrainUnpredictableBool(Unpredi
else
    is_nevt = TRUE; // not implemented

boolean is_rejected_event = FALSE;

// Filtering by event
m = PMSEVFR_EL1 AND mask;
boolean is_evt = IsZero(NOT(e) AND m);
if PMSFCR_EL1.FE == '1' then
    // Filtering by event is enabled
    if !IsZero(m) then
        // Not UNPREDICTABLE case
        is_rejected_event = !is_evt;
    else
        is_rejected_event = ConstrainUnpredictableBool(Unpredictabl

if (IsFeatureImplemented(FEAT_SPEv1p2) && PMSFCR_EL1.<FnE,FE> == '1'
    !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
    // UNPREDICTABLE case due to combination of filter and inverse f
    is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_E
    is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BA

if is_evt && is_nevt then
    PMUEvent(PMU_EVENT_SAMPLE_FEED_EVENT);

boolean is_op_br = FALSE;
boolean is_op_ld = FALSE;
boolean is_op_st = FALSE;

is_op_br = (opType == OpType_Branch);
is_op_ld = (opType IN {OpType_Load, OpType_LoadAtomic});
is_op_st = (opType IN {OpType_Store, OpType_LoadAtomic});

if is_op_br then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR);
if is_op_ld then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD);
if is_op_st then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST);

boolean is_op = ((is_op_br && PMSFCR_EL1.B == '1') ||
                  (is_op_ld && PMSFCR_EL1.LD == '1') ||
                  (is_op_st && PMSFCR_EL1.ST == '1'));

if is_op then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP);

// Filter by type
boolean is_rejected_type = FALSE;
if PMSFCR_EL1.FT == '1' then
    // Filtering by type is enabled

```

```

if !IsZero(PMSFCR_EL1.<B, LD, ST>) then
    // Not an UNPREDICTABLE case
    is_rejected_type = !is_op;
else
    is_rejected_type = ConstrainUnpredictableBool(Unpredictable);

// Filter by latency
boolean is_rejected_latency = FALSE;
boolean is_lat = (total_latency < UInt(PMSLATFR_EL1.MINLAT));
if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT);

if PMSFCR_EL1.FL == '1' then
    // Filtering by latency is enabled
    if !IsZero(PMSLATFR_EL1.MINLAT) then
        // Not an UNPREDICTABLE case
        is_rejected_latency = !is_lat;
    else
        is_rejected_latency = ConstrainUnpredictableBool(Unpredictable);

boolean is_rejected_data_source;
// Filtering by Data Source
if (IsFeatureImplemented(FEAT_SPE_FDS) && PMSFCR_EL1.FDS == '1' &&
    is_op_ld && SPESampleDataSourceValid) then
    bits(16) data_source = SPESampleDataSource;
    integer index = UInt(data_source<5:0>);
    is_rejected_data_source = PMSDSFR_EL1<index> == '0';
else
    is_rejected_data_source = FALSE;

boolean return_value;
return_value = !(is_rejected_nevent || is_rejected_event ||
                 is_rejected_type || is_rejected_latency);

if return_value then
    PMUEvent(PMU_EVENT_SAMPLE_FILTERATE);
return return_value;

```

### Library pseudocode for aarch64/debug/statisticalprofiling/ SPEConstructRecord

```

// SPEConstructRecord()
// =====
// Create new record and populate it with packets using sample storage
// This is an example implementation, packets may appear in
// any order as long as the record ends with an End or Timestamp packet

SPEConstructRecord()
// Empty the record.
SPEEmptyRecord();

// Add contextEL1 if available
if SPESampleContextEL1Valid then
    SPEAddPacketToRecord('01', '0100', SPESampleContextEL1);

// Add contextEL2 if available
if SPESampleContextEL2Valid then
    SPEAddPacketToRecord('01', '0101', SPESampleContextEL2);

```

```

// Add valid counters
for counter_index = 0 to (SPEMaxCounters - 1)
    if SPESampleCounterValid[counter_index] then
        if counter_index >= 8 then
            // Need extended format
            SPEAddByteToRecord('001000':counter_index<4:3>);
        // Check for overflow
        boolean large_counters = boolean IMPLEMENTATION_DEFINED "SP"
        if SPESampleCounter[counter_index] > 0xFFFF && large_counters
            SPESampleCounter[counter_index] = 0xFFFF;
        elsif SPESampleCounter[counter_index] > 0xFFF then
            SPESampleCounter[counter_index] = 0FFF;

        // Add byte0 for short format (byte1 for extended format)
        SPEAddPacketToRecord('10', '1':counter_index<2:0>,
                               SPESampleCounter[counter_index]<15:0>);

// Add valid addresses
if IsFeatureImplemented(FEAT_SPEv1p2) then
    // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
    // previous branch packet is present.
    boolean include_prev_br = boolean IMPLEMENTATION_DEFINED "SPE o"
    if SPESampleOpType != OpType_Branch && !include_prev_br then
        SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = FALSE;

// Data Virtual address should not be collected if this was an NV2
// Profiling is disabled at EL2.
if !StatisticalProfilingEnabled(EL2) && SPESampleInstIsNV2 then
    SPESampleAddressValid[SPEAddrPosDataVirtual] = FALSE;

for address_index = 0 to (SPEMaxAddrs - 1)
    if SPESampleAddressValid[address_index] then
        if address_index >= 8 then
            // Need extended format
            SPEAddByteToRecord('001000':address_index<4:3>);
        // Add byte0 for short format (byte1 for extended format)
        SPEAddPacketToRecord('10', '0':address_index<2:0>,
                               SPESampleAddress[address_index]);

// Add Data Source
if SPESampleDataSourceValid then
    constant integer ds_payload_size = SPEGetDataSourcePayloadSize();
    SPEAddPacketToRecord('01', '0011', SPESampleDataSource<8*ds_payload_size>);

// Add operation details
SPEAddPacketToRecord('01', '10':SPESampleClass, SPESampleSubclass);

// Add events
// Get size of payload in bytes.
constant integer events_payload_size = SPEGetEventsPayloadSize();
SPEAddPacketToRecord('01', '0010', SPESampleEvents<8*events_payload_size>);

// Add Timestamp to end the record if one is available.
// Otherwise end with an End packet.
if SPESampleTimestampValid then
    SPEAddPacketToRecord('01', '0001', SPESampleTimestamp);
else
    SPEAddByteToRecord('00000001');

// Add padding

```

```

        while SPERecordSize MOD (1<<UInt(PMBIDR_EL1.Align)) != 0 do
            SPEAddByteToRecord(Zeros(8));
            SPEWriteToBuffer();

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPECycle

```

// SPECycle()
// =====
// Function called at the end of every cycle. Responsible for asserting
// and advancing counters.

SPECycle()
    if !IsFeatureImplemented(FEAT_SPE) then
        return;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[i] then
                SPESampleCounter[i] = SPESampleCounter[i] + 1;

    // Assert PMBIRQ if appropriate.
    SetInterruptRequestLevel(InterruptID_PMBIRO,
        if PMBSR_EL1.S == '1' then Signal_High else

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEEmptyRecord

```

// SPEEmptyRecord()
// =====
// Reset record data.

SPEEmptyRecord()
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1)
        SPERecordData[i] = Zeros(8);

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEEvent

```

// SPEEvent()
// =====
// Called by PMUEvent if a sample is in flight.
// Sets appropriate bit in SPESampleStorage.events.

SPEEvent(bits(16) pmuevent)
    case pmuevent of
        when PMU_EVENT_DSNP_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<23> = '1';
        when PMU_EVENT_L1D_LFB_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<22> = '1';
        when PMU_EVENT_L2D_LFB_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<22> = '1';
        when PMU_EVENT_L3D_LFB_HIT_RD

```

```

        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<22> = '1';
when PMU_EVENT_LL_LFB_HIT_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<22> = '1';
when PMU_EVENT_L1D_CACHE_HITM_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<21> = '1';
when PMU_EVENT_L2D_CACHE_HITM_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<21> = '1';
when PMU_EVENT_L3D_CACHE_HITM_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<21> = '1';
when PMU_EVENT_LL_CACHE_HITM_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<21> = '1';
when PMU_EVENT_L2D_CACHE_LMISS_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<20> = '1';
when PMU_EVENT_L2D_CACHE_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<19> = '1';
when PMU_EVENT_SVE_PRED_EMPTY_SPEC
        if IsFeatureImplemented(FEAT_SPEv1p1) then
            SPESampleEvents<18> = '1';
when PMU_EVENT_SVE_PRED_PARTIAL_SPEC
        if IsFeatureImplemented(FEAT_SPEv1p1) then
            SPESampleEvents<17> = '1';
when PMU_EVENT_LDST_ALIGN_LAT
        if IsFeatureImplemented(FEAT_SPEv1p1) then
            SPESampleEvents<11> = '1';
when PMU_EVENT_REMOTE_ACCESS           SPESampleEvents<10> = '1'
when PMU_EVENT_LL_CACHE_MISS          SPESampleEvents<9> = '1'
when PMU_EVENT_LL_CACHE              SPESampleEvents<8> = '1'
when PMU_EVENT_BR_MIS_PRED           SPESampleEvents<7> = '1'
when PMU_EVENT_BR_MIS_PRED_RETIRED   SPESampleEvents<7> = '1'
when PMU_EVENT_DTLB_WALK             SPESampleEvents<5> = '1'
when PMU_EVENT_L1D_TLB               SPESampleEvents<4> = '1'
when PMU_EVENT_L1D_CACHE_REFILL
        if !IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<3> = '1';
when PMU_EVENT_L1D_CACHE_LMISS_RD
        if IsFeatureImplemented(FEAT_SPEv1p4) then
            SPESampleEvents<3> = '1';
when PMU_EVENT_L1D_CACHE             SPESampleEvents<2> = '1'
when PMU_EVENT_INST_RETIRIED         SPESampleEvents<1> = '1'
when PMU_EVENT_EXC_TAKEN            SPESampleEvents<0> = '1'
otherwise return;
return;

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEFreezeOnEvent

```

// SPEFreezeOnEvent()
// =====
// Returns TRUE if PMU event counter idx should be frozen due to an SPE
boolean SPEFreezeOnEvent(integer idx)

```

```

assert 0 <= idx;
if !IsFeatureImplemented(FEAT_SPEv1p2) || !IsFeatureImplemented(FEAT_SPE_DPFZS)
if PMBSR_EL1.S != '1' || PMBLIMITR_EL1.[E,PMFZ] != '11' then return FALSE;

if idx == CYCLE_COUNTER_ID && !IsFeatureImplemented(FEAT_SPE_DPFZS)
    // FZS does not affect the cycle counter when FEAT_SPE_DPFZS is
    return FALSE;

if PMUCounterIsHyp(idx) then
    return MDCR_EL2.HPMFZS == '1';
else
    return PMCR_EL0.FZS == '1';

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEGetDataSourcePayloadSize**

```

// SPEGetDataSourcePayloadSize()
// =====
// Returns the size of the Data Source payload in bytes.

integer SPEGetDataSourcePayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Data Source packet payload size";

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEGetEventsPayloadSize**

```

// SPEGetEventsPayloadSize()
// =====
// Returns the size in bytes of the Events packet payload as an integer.

integer SPEGetEventsPayloadSize()
    integer size = integer IMPLEMENTATION_DEFINED "SPE Events packet payload size";
    return size;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEGetRandomBoolean**

```

// SPEGetRandomBoolean()
// =====
// Returns a random or pseudo-random boolean value.

boolean SPEGetRandomBoolean();

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEGetRandomInterval**

```

// SPEGetRandomInterval()
// =====
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

bits(8) SPEGetRandomInterval();

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEISB

```
// SPEISB()
// ======
// Called by ISB instruction, correctly calls SPEBranch to save previous state
SPEISB()
    bits(64) address = PC64 + 4;
    BranchType branch_type = BranchType_DIR;
    boolean branch_conditional = FALSE;
    boolean taken = FALSE;
    boolean is_isb = TRUE;

    SPEBranch(address, branch_type, branch_conditional, taken, is_isb);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxAddrs

```
constant integer SPEMaxAddrs = 32;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxCounters

```
constant integer SPEMaxCounters = 32;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxRecordSize

```
constant integer SPEMaxRecordSize = 64;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEPostExecution

```
constant integer SPEAddrPosPCVirtual = 0;
constant integer SPEAddrPosBranchTarget = 1;
constant integer SPEAddrPosDataVirtual = 2;
constant integer SPEAddrPosDataPhysical = 3;
constant integer SPEAddrPosPrevBranchTarget = 4;
constant integer SPECounterPosTotalLatency = 0;
constant integer SPECounterPosIssueLatency = 1;
constant integer SPECounterPosTranslationLatency = 2;
boolean SPESampleInFlight = FALSE;
bits(32) SPESampleContextEL1;
boolean SPESampleContextEL1Valid;
bits(32) SPESampleContextEL2;
boolean SPESampleContextEL2Valid;
boolean SPESampleInstIsNV2 = FALSE;
bits(64) SPESamplePreviousBranchAddress;
boolean SPESamplePreviousBranchAddressValid;
bits(16) SPESampleDataSource;
boolean SPESampleDataSourceValid;
OpType SPESampleOpType;
bits(2) SPESampleClass;
bits(8) SPESampleSubclass;
boolean SPESampleSubclassValid;
bits(64) SPESampleTimestamp;
```

```

boolean SPESampleTimestampValid;
bits(64) SPESampleEvents;

// SPEPostExecution()
// =====
// Called after every executed instruction.

SPEPostExecution()
    if SPESampleInFlight then
        SPESampleInFlight = FALSE;
        PMUEvent(PMU_EVENT_SAMPLE_FEED);

        // Stop any pending counters
        for counter_index = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[counter_index] then
                SPEStopCounter(counter_index);

        boolean discard = FALSE;
        if IsFeatureImplemented(FEAT_SPEv1p2) then
            discard = PMBLIMITR_EL1.FM == '10';
        if SPECollectRecord(SPESampleEvents,
            SPESampleCounter[SPECounterPosTotalLatency]
            SPESampleOpType) && !discard then
            SPEConstructRecord();
            if SPEBufferIsFull() then
                SPEBufferFilled();

        SPEResetSampleStorage();

    // Counter storage
    array [0..SPEMaxCounters-1] of integer SPESampleCounter;
    array [0..SPEMaxCounters-1] of boolean SPESampleCounterValid;
    array [0..SPEMaxCounters-1] of boolean SPESampleCounterPending;

    // Address storage
    array [0..SPEMaxAddrs-1] of bits(64) SPESampleAddress;
    array [0..SPEMaxAddrs-1] of boolean SPESampleAddressValid;

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEPreExecution

```

// SPEPreExecution()
// =====
// Called prior to execution, for all instructions.

SPEPreExecution()
    if StatisticalProfilingEnabled() then
        PMUEvent(PMU_EVENT_SAMPLE_POP);
        if SPEToCollectSample() then
            if !SPESampleInFlight then
                SPESampleInFlight = TRUE;

            // Start total latency and issue latency counters for S
            SPEStartCounter(SPECounterPosTotalLatency);
            SPEStartCounter(SPECounterPosIssueLatency);

```

```

        SPESampleAddContext() ;

        SPESampleAddAddressPCVirtual() ;

        // Timestamp may be collected at any point in the sample
        // Collecting prior to execution is one possible choice
        // This choice is IMPLEMENTATION_DEFINED.
        SPESampleAddTimeStamp() ;
    else
        PMUEvent(PMU_EVENT_SAMPLE_COLLISION) ;
        PMBSR_EL1.COLL = '1' ;

        // Many operations are type other and not conditional, can
        // and overhead by having this as the default and not calling
        // if conditional == FALSE
        SPESampleAddOpOther(FALSE) ;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEResetSampleCounter**

```

// SPEResetSampleCounter()
// =====
// Reset PMSICR_EL1.Counter

SPEResetSampleCounter()
    PMSICR_EL1.COUNT<31:8> = PMSIRR_EL1.INTERVAL;
    if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '0' then
        PMSICR_EL1.COUNT<7:0> = SPEGetRandomInterval() ;
    else
        PMSICR_EL1.COUNT<7:0> = Zeros(8) ;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPEResetSampleStorage**

```

integer SPERecordSize;

// SPEResetSampleStorage()
// =====
// Reset all variables inside sample storage.

SPEResetSampleStorage()
    // Context values
    SPESampleContextEL1 = Zeros(32);
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2 = Zeros(32);
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1)
        SPESampleCounter[i] = 0;
        SPESampleCounterValid[i] = FALSE;
        SPESampleCounterPending[i] = FALSE;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1)
        SPESampleAddressValid[i] = FALSE;
        SPESampleAddress[i] = Zeros(64);

```

```

// Data source values
SPESampleDataSource = Zeros(16);
SPESampleDataSourceValid = FALSE;

// Operation values
SPESampleClass = Zeros(2);
SPESampleSubclass = Zeros(8);
SPESampleSubclassValid = FALSE;

// Timestamp values
SPESampleTimestamp = Zeros(64);
SPESampleTimestampValid = FALSE;

// Event values
SPESampleEvents<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE EVENTS";
SPESampleEvents<47:32> = Zeros(16);
SPESampleEvents<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE EVENTS";
SPESampleEvents<23:16> = Zeros(8);
SPESampleEvents<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE EVENTS";
SPESampleEvents<11:0> = Zeros(12);

SPESampleInstIsNV2 = FALSE;

array [0..SPEMaxRecordSize-1] of bits(8) SPERecordData;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleAddAddressPCVirtual**

```

// SPESampleAddAddressPCVirtual()
// =====
// Save the current PC address to sample storage.

SPESampleAddAddressPCVirtual()
    bits(64) this_address = ThisInstrAddr(64);
    SPESampleAddress[SPEAddrPosPCVirtual]<55:0> = this_address<55:0>;

    bit ns;
    bit nse;
    case CurrentSecurityState() of
        when SS_Secure
            ns = '0';
            nse = '0';
        when SS_NonSecure
            ns = '1';
            nse = '0';
        when SS_Realm
            ns = '1';
            nse = '1';
        otherwise Unreachable();

    bits(2) el = PSTATE.EL;
    SPESampleAddress[SPEAddrPosPCVirtual]<63:56> = ns:el:nse:Zeros(4);
    SPESampleAddressValid[SPEAddrPosPCVirtual] = TRUE;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleAddContext**

```

// SPESampleAddContext()
// =====
// Save contexts to sample storage if appropriate.

SPESampleAddContext()
    if CollectContextIDR1\(\) then
        SPESampleContextEL1 = CONTEXTIDR_EL1<31:0>;
        SPESampleContextEL1Valid = TRUE;
    if CollectContextIDR2\(\) then
        SPESampleContextEL2 = CONTEXTIDR_EL2<31:0>;
        SPESampleContextEL2Valid = TRUE;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleAddOpOther**

```

// SPESampleAddOpOther()
// =====
// Add other operation to sample storage.

SPESampleAddOpOther(boolean conditional, boolean taken)
    SPESampleEvents<6> = if conditional && !taken then '1' else '0';
    SPESampleAddOpOther(conditional);

SPESampleAddOpOther(boolean conditional)
    SPESampleClass = '00';
    SPESampleSubclass<0> = if conditional then '1' else '0';
    SPESampleOpType = OpType\_Other;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleAddOpSVELoadStore**

```

// SPESampleAddOpSVELoadStore()
// =====
// Sets the subclass of the operation type packet to Load/Store for SVE

SPESampleAddOpSVELoadStore(boolean is_gather_scatter, bits(3) evl, boolean
                           is_load)
    bit sg = if is_gather_scatter then '1' else '0';
    bit pred = if predicated then '1' else '0';
    bit ldst = if is_load then '0' else '1';
    SPESampleClass = '01';
    SPESampleSubclass<7:0> = sg:evl:'1':pred:'0':ldst;
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = if is_load then OpType\_Load else OpType\_Store;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleAddOpSVEOther**

```

// SPESampleAddOpSVEOther()
// =====
// Sets the subclass of the operation type packet to Other for SVE operations

SPESampleAddOpSVEOther(bits(3) evl, boolean predicated, boolean floating_point)
    bit pred = if predicated then '1' else '0';
    bit fp = if floating_point then '1' else '0';

```

```

SPESampleClass = '00';
SPESampleSubclass<7:0> = '0':evl:'1':pred:fp:'0';
SPESampleSubclassValid = TRUE;
SPESampleOpType = OpType\_Other;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleAddTimeStamp**

```

// SPESampleAddTimeStamp()
// =====
// Save the appropriate type of timestamp to sample storage.

SPESampleAddTimeStamp()
    TimeStamp timestamp = CollectTimeStamp\(\);
    case timestamp of
        when TimeStamp\_None
            SPESampleTimestampValid = FALSE;
        otherwise
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimestamp(timestamp);

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleExtendedLoadStore**

```

// SPESampleExtendedLoadStore()
// =====
// Sets the subclass of the operation type packet for
// extended load/store operations.

SPESampleExtendedLoadStore(bit ar, bit excl, bit at, boolean is_load)
    SPESampleClass = '01';
    bit ldst = if is_load then '0' else '1';
    SPESampleSubclass = '000':ar:excl:at:'1':ldst;

    SPESampleSubclassValid = TRUE;

    if is_load then
        if at == '1' then
            SPESampleOpType = OpType\_LoadAtomic;
        else
            SPESampleOpType = OpType\_Load;
    else
        SPESampleOpType = OpType\_Store;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleGeneralPurposeLoadStore**

```

// SPESampleGeneralPurposeLoadStore()
// =====
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

SPESampleGeneralPurposeLoadStore()
    SPESampleClass = '01';

```

```
SPESampleSubclass<7:1> = Zeros(7);  
SPESampleSubclassValid = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleLoadStore

```
// SPESampleLoadStore()  
// =====  
// Called if a sample is in flight when writing or reading memory,  
// indicating that the operation being sampled is in the Load, Store or  
  
SPESampleLoadStore(boolean is_load, AccessDescriptor accdesc, AddressDescriptor addrdesc)  
    // Check if this access type should be sampled.  
    if accdesc.acctype IN {AccessType SPE,  
                           AccessType IFETCH,  
                           AccessType DC,  
                           AccessType TTW,  
                           AccessType AT} then  
        return;  
  
    // MOPS instructions indicate which operation should be sampled before  
    // operation is executed. Has the instruction indicated that the load?  
    boolean sample_loads;  
    sample_loads = SPESampleSubclass<0> == '0' && SPESampleSubclassValid;  
  
    // Has the instruction indicated that the store should be sampled?  
    boolean sample_stores;  
    sample_stores = SPESampleSubclass<0> == '1' && SPESampleSubclassValid;  
  
    // No valid data has been collected, or this is operation has specified  
    // sampling.  
    if (!SPESampleSubclassValid || (sample_loads && is_load) ||  
        (sample_stores && !is_load)) then  
        // Data access virtual address  
        SPESetDataVirtualAddress(addrdesc.vaddress);  
  
        // Data access physical address  
        if CollectPhysicalAddress() then  
            SPESetDataPhysicalAddress(addrdesc, accdesc);  
  
    if !SPESampleSubclassValid then  
        // Set as unspecified load/store by default, instructions will  
        // apply to them.  
        SPESampleClass = '01';  
        SPESampleSubclassValid = TRUE;  
        SPESampleSubclass<7:1> = '0001000';  
        SPESampleSubclass<0> = if is_load then '0' else '1';  
        SPESampleOpType = if is_load then OpType Load else OpType Store  
  
        if accdesc.acctype == AccessType NV2 then  
            // NV2 register load/store  
            SPESampleSubclass<7:1> = '0011000';  
            SPESampleInstIsNV2 = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleMemcpy

```

// SPESampleMemcpy()
// =====
// Sets the subclass of the operation type packet for Memory Copy load/
// operations.

SPESampleMemcpy()
    // MemCopy does a read and a write. If one is filtered out, the other
    // If neither or both are filtered out, pick one in a (pseudo) random
    // manner.

    // Are loads allowed by filter?
    boolean loads_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.LD ==
    // Are stores allowed by filter?
    boolean stores_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.ST ==

    boolean record_load;
    if loads_pass_filter && !stores_pass_filter then
        // Only loads pass filter
        record_load = TRUE;
    elseif !loads_pass_filter && stores_pass_filter then
        // Only stores pass filter
        record_load = FALSE;
    else
        // Pick randomly between
        record_load = SPEGetRandomBoolean\(\);

    SPESampleClass = '01';
    bit ldst = if record_load then '0' else '1';
    SPESampleSubclass<7:0> = '0010000':ldst;
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = if record_load then OpType\_Load else OpType\_Store;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemSet**

```

// SPESampleMemSet()
// =====
// Sets the subclass of the operation type packet for Memory Set load/s
// operation.

SPESampleMemSet()
    SPESampleClass = '01';
    SPESampleSubclass<7:0> = '00100101';
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = OpType\_Store;

```

### **Library pseudocode for aarch64/debug/statisticalprofiling/ SPESampleSIMDFPLoadStore**

```

// SPESampleSIMDFPLoadStore()
// =====
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

SPESampleSIMDFPLoadStore()
    SPESampleClass = '01';

    SPESampleSubclass<7:1> = '0000010';
    SPESampleSubclassValid = TRUE;

```

## Library pseudocode for aarch64/debug/statisticalprofiling/ SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// =====
// Called from SampleLoadStore() to save data physical packet.

SPESetDataPhysicalAddress(AddressDescriptor addrdesc, AccessDescriptor
    bit ns;
    bit nse;
    case addrdesc.paddress.paspace of
        when PAS\_Secure
            ns = '0';
            nse = '0';
        when PAS\_NonSecure
            ns = '1';
            nse = '0';
        when PAS\_Realm
            ns = '1';
            nse = '1';
        otherwise Unreachable\(\);

    if IsFeatureImplemented(FEAT_MTE2) then
        bits(4) pat;
        if accdesc.tagchecked then
            SPESampleAddress[SPEAddrPosDataPhysical]<62> = '1'; // CH
            pat = AArch64.PhysicalTag(addrdesc.vaddress);
        else
            // CH is reset to 0 on each new packet
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED
            // between 0b0000 and the Physical Address Tag
            boolean zero_unchecked;
            zero_unchecked = boolean IMPLEMENTATION_DEFINED "SPE PAT for
            if !zero_unchecked then
                pat = AArch64.PhysicalTag(addrdesc.vaddress);
            else
                pat = Zeros(4);
            SPESampleAddress[SPEAddrPosDataPhysical]<59:56> = pat;

        bits(56) paddr = addrdesc.paddress.address;
        SPESampleAddress[SPEAddrPosDataPhysical]<56-1:0> = paddr;
        SPESampleAddress[SPEAddrPosDataPhysical]<63> = ns;
        SPESampleAddress[SPEAddrPosDataPhysical]<60> = nse;
        SPESampleAddressValid[SPEAddrPosDataPhysical] = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ SPESetDataVirtualAddress

```
// SPESetDataVirtualAddress()
// =====
// Called from SampleLoadStore() to save data virtual packet.
// Also used by exclusive load/stores to save virtual addresses if excl
// before a read/write is completed.

SPESetDataVirtualAddress(bits(64) vaddress)
    bit tbi;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
```

```

boolean non_tbi_is_zeros;
non_tbi_is_zeros = boolean IMPLEMENTATION_DEFINED "SPE non-tbi tag
if tbi == '1' || !non_tbi_is_zeros then
    SPESampleAddress[SPEAddrPosDataVirtual]<63:0> = vaddress<63:0>;
else
    SPESampleAddress[SPEAddrPosDataVirtual]<63:56> = zeros(8);
    SPESampleAddress[SPEAddrPosDataVirtual]<55:0> = vaddress<55:0>;
SPESampleAddressValid[SPEAddrPosDataVirtual] = TRUE;

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEStartCounter

```

// SPEStartCounter()
// =====
// Enables incrementing of the counter at the passed index when SPECycl
SPEStartCounter(integer counter_index)
    assert counter_index < SPEMaxCounters;
    SPESampleCounterPending[counter_index] = TRUE;

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEStopCounter

```

// SPEStopCounter()
// =====
// Disables incrementing of the counter at the passed index when SPECycl
SPEStopCounter(integer counter_index)
    SPESampleCounterValid[counter_index] = TRUE;
    SPESampleCounterPending[counter_index] = FALSE;

```

### Library pseudocode for aarch64/debug/statisticalprofiling/ SPEToCollectSample

```

// SPEToCollectSample()
// =====
// Returns TRUE if the instruction which is about to be executed should
// be sampled. Returns FALSE otherwise.

boolean SPEToCollectSample()
    if IsZero(PMSICR_EL1.COUNT) then
        SPEResetSampleCounter();
    else
        PMSICR_EL1.COUNT = PMSICR_EL1.COUNT - 1;
        if IsZero(PMSICR_EL1.COUNT) then
            if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '1' then
                PMSICR_EL1.ECOUNT = SPEGetRandomInterval();
            else
                return TRUE;
        if UInt(PMSICR_EL1.ECOUNT) != 0 then
            PMSICR_EL1.ECOUNT = PMSICR_EL1.ECOUNT - 1;
            if IsZero(PMSICR_EL1.ECOUNT) then
                return TRUE;
    return FALSE;

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEWriteToBuffer

```
// SPEWriteToBuffer()
// =====
// Write the active record to the Profiling Buffer.

SPEWriteToBuffer()
    assert ProfilingBufferEnabled\(\);

    // Check alignment
    constant integer align = UInt(PMBIDR_EL1.Align);
    boolean aligned = IsZero(PMBPTR_EL1.PTR<align-1:0>);
    boolean ttw_fault_as_external_abort;
    ttw_fault_as_external_abort = boolean IMPLEMENTATION_DEFINED "SPE T

    FaultRecord fault;
    PhysMemRetStatus memstatus;
    AddressDescriptor addrdesc;
    AccessDescriptor accdesc;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner\(\);
    accdesc = CreateAccDescSPE(owning_ss, owning_el);

    bits(64) start_vaddr = PMBPTR_EL1<63:0>;
    for i = 0 to SPERecordSize - 1
        // If a previous write did not cause an issue
        if PMBSR_EL1.S == '0' then
            (memstatus, addrdesc) = DebugMemWrite(PMBPTR_EL1<63:0>, accdesc,
                SPERecordData[i]);

        fault = addrdesc.fault;

        boolean ttw_fault;
        ttw_fault = fault.statuscode IN {Fault\_SyncExternalOnWalk,
            Fault\_TTWExternalAbort};

        if IsFault(fault.statuscode) && !(ttw_fault && ttw_fault_as_external_abort)
            DebugWriteFault(PMBPTR_EL1<63:0>, fault);
        elseif IsFault(memstatus) || (ttw_fault && ttw_fault_as_external_abort)
            DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);

        // Move pointer if no Buffer Management Event has been caused
        if IsZero(PMBSR_EL1.S) then
            PMBPTR_EL1 = PMBPTR_EL1 + 1;

    return;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

boolean StatisticalProfilingEnabled()
    return StatisticalProfilingEnabled(PSTATE.EL);
```

```

// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the specified EL, F
boolean StatisticalProfilingEnabled(bits(2) el)
    if !IsFeatureImplemented(FEAT_SPE) || UsingAArch32\(\) || !ProfilingBu
        return FALSE;

    tge_set = EL2Enabled\(\) && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner\(\);
    if (UInt(owning_el) < UInt(el) || (tge_set && owning_el == EL1) ||
        owning_ss != SecurityStateAtEL(el)) then
        return FALSE;
    bit spe_bit;
    case el of
        when EL3 Unreachable\(\);
        when EL2 spe_bit = PMSCR_EL2.E2SPE;
        when EL1 spe_bit = PMSCR_EL1.E1SPE;
        when EL0 spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSC
    return spe_bit == '1';

```

### Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```

// TimeStamp
// =====

enumeration TimeStamp {
    TimeStamp_None,                      // No timestamp
    TimeStamp_CoreSight,                  // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,                  // Physical counter value with no offset
    TimeStamp_OffsetPhysical,            // Physical counter value minus CNTPOFF
    TimeStamp_Virtual };                // Physical counter value minus CNTVOFF

```

### Library pseudocode for aarch64/debug/takeexceptiondbg/ AArch64.TakeExceptionInDebugState

```

// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord ex
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(tar
    assert target_el != EL3 || EDSCR.SDD == '0';
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR\_EL[target_el].IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' &&
            // SCTLR_EL[].IESB and/or SCR_EL3.NMEA (if applicable) might be
            if !ConstrainUnpredictableBool(Unpredictable\_IESBinDebug) then
                sync_errors = FALSE;
    else
        sync_errors = FALSE;

    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then

```

```

TMFailure cause;
case except.exceptype of
    when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
    when Exception_Breakpoint           cause = TMFailure_DBG;
    when Exception_Watchpoint          cause = TMFailure_DBG;
    when Exception_SoftwareStep        cause = TMFailure_DBG;
    otherwise                          cause = TMFailure_ERR;
    FailTransaction(cause, FALSE);

boolean brbe_source_allowed = FALSE;
bits(64) brbe_source_address = Zeros(64);
if IsFeatureImplemented(FEAT_BRBE) then
    brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
    brbe_source_address = bits(64) UNKNOWN;

if !IsFeatureImplemented(FEAT_ExS) || SCLTR_EL[target_el].EIS == '1'
    SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
    ResetSVEstate();
else
    MaybeZeroSVEUppers(target_el);

AArch64.ReportException(except, target_el);

if IsFeatureImplemented(FEAT_GCS) then
    PSTATE.EXLOCK = '0'; // Effective value of GCSCR_ELx.EXLOCKEN is

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR_Elx[] = bits(64) UNKNOWN;
ELR_Elx[] = bits(64) UNKNOWN;

// PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state
PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
PSTATE.IL = '0';
if from_32 then                                     // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0';                                // PSTATE.J is RES0
if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
    (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCLTR_Elx[].SPAN == '0') then
    PSTATE.PAN = '1';
if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = bit UNKNOWN;
if IsFeatureImplemented(FEAT_SEBEP) then
    PSTATE.PPEND = '0';
    ShouldSetPPEND = FALSE;

DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;

```

```
EDSCR.ERR = '1';
UpdateEDSCRFields\(\); // Update EDSCR process

if sync_errors then
    SynchronizeErrors\(\);

EndOfInstruction\(\);
```

**Library pseudocode for aarch64/debug/watchpoint/  
AArch64.WatchpointByteMatch**

```

// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)
    constant integer dbgtop = DebugAddrTop();
    constant integer cmpbottom = if DBGWVR_EL1[n]<2> == '1' then 2 else
        integer bottom = cmpbottom;
    constant integer select = UInt(vaddress<cmpbottom-1:0>);
    byte_select_match = (DBGWCR_EL1[n].BAS<select> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is a nonzero value and DBGWCR_EL1[n].BAS is
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WP);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable\_WP);
            bottom = 3; // For the whole document

    // If the address mask is set to a reserved value, the behavior is
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable\_WP);
        assert c IN {Constraint\_DISABLED, Constraint\_NONE, Constraint\_UNPREDICTABLE};
        case c of
            when Constraint\_DISABLED return FALSE; // Disabled
            when Constraint\_NONE mask = 0; // No masking
            otherwise the value returned by ConstrainUnpredictableInteger

    // When FEAT_LVA3 is not implemented, if the DBGWVR_EL1[n].RESS field
    // sign extension of the MSB of DBGWVR_EL1[n].VA, it is UNPREDICTABLE
    // appear to be included in the match.
    boolean unpredictable_ress = (dbgtop < 55 && !IsOnes(DBGWVR_EL1[n]<55:dbgtop>)
        !IsZero(DBGWVR_EL1[n]<63:dbgtop>) &&
        ConstrainUnpredictableBool(Unpredictable\_WP));
    constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
    constant integer cmplsb = if mask > bottom then mask else bottom;
    constant integer bottombit = bottom;
    boolean WVR_match = (vaddress<cmpmsb:cmplsb> == DBGWVR_EL1[n]<cmpmsb:cmplsb>);
    if mask > bottom then
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is
        // UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<cmpbottom-1:bottombit>)
            WVR_match = ConstrainUnpredictableBool(Unpredictable\_WP);

    return (WVR_match && byte_select_match);

```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```

// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer s
    AccessDescriptor accdesc)
    assert !ELUsingAArch32(S1TranslationRegime());

```

```

assert n < NumWatchpointsImplemented();

boolean enabled      = IsWatchpointEnabled(n);
linked = DBGWCR_EL1[n].WT == '1';
isbreakpt = FALSE;
lbnx = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGWCR_EL1[n].I
linked_n = UInt(lbnx : DBGWCR_EL1[n].LBN);
ssce = if IsFeatureImplemented(FEAT_RME) then DBGWCR_EL1[n].SSCE el
state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EI
                           linked, linked_n, isbreakpt, PC64

boolean ls_match;
case DBGWCR_EL1[n].LSC<1:0> of
    when '00' ls_match = FALSE;
    when '01' ls_match = accdesc.read;
    when '10' ls_match = accdesc.write || accdesc.acctype == AccessType
    when '11' ls_match = TRUE;

boolean value_match = FALSE;
for byte = 0 to size - 1
    value_match = value_match || AArch64.WatchpointByteMatch(n, vac

return value_match && state_match && ls_match && enabled;

```

## Library pseudocode for aarch64/debug/watchpoint/IsWatchpointEnabled

```

// IsWatchpointEnabled()
// =====
// Returns TRUE if the effective value of DBGWCR_EL1[n].E is '1', and FA

boolean IsWatchpointEnabled(integer n)
    if (n > 15 &&
        (!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPE
        (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0'))) th
    return FALSE;
return DBGWCR_EL1[n].E == '1';

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```

// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

if IsDebugException(fault) then
    if fault.accessdesc.acctype == AccessType_IFETCH then
        if UsingAArch32() && fault.debugmoe == DebugException_Vecto
            AArch64.VectorCatchException(fault);
        else
            AArch64.BreakpointException(fault);
    else
        AArch64.WatchpointException(vaddress, fault);
elseif fault.gpcf.gpf != GPCF_None && ReportAsGPCException(fault) th
    TakeGPCException(vaddress, fault);
elseif fault.accessdesc.acctype == AccessType_IFETCH then
    AArch64.InstructionAbort(vaddress, fault);

```

```

    else
        AArch64.DataAbort(vaddress, fault);

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```

// AArch64.AbortSyndrome()
// =====
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord
                                         bits(64) vaddress, bits(2) target_el)
{
    except = ExceptionSyndrome(exceptype);

    if (!IsFeatureImplemented(FEAT_PFAR) ||
        !IsExternalSyncAbort(fault) ||
        (EL2Enabled() && HCR_EL2.VM == '1' && target_el == EL1)) then
        except.pavalid = FALSE;
    else
        except.pavalid = boolean IMPLEMENTATION_DEFINED "PFAR_ELx is valid";

    (except.syndrome, except.syndrome2) = AArch64.FaultSyndrome(exceptype,
                                                               vaddress);

    if fault.statuscode == Fault\_TagCheck then
        if IsFeatureImplemented(FEAT_MTE4) then
            except.vaddress = ZeroExtend(vaddress, 64);
        else
            except.vaddress = bits(4) UNKNOWN : vaddress<59:0>;
    else
        except.vaddress = ZeroExtend(vaddress, 64);

    if IPAVValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS\_NonSecure then '1'
                    else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    return except;
}

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```

// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()
{
    bits(64) pc = ThisInstrAddr(64);

    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
}

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```

// AArch64.DataAbort()
// =====

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = AArch64.SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         (IsFeatureImplemented(FEAT_RME) && fault.gpcf. ||
                          HCR_EL2.GPF == '1') ||
                         (IsFeatureImplemented(FEAT_NV2) &&
                          fault.accessdesc.acctype == AccessType_NV2) ||
                         IsSecondStage(fault)));
        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;

    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el)
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    ExceptionRecord except;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2
        except = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        except = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

```

// AArch64.EffectiveTCF()
// =====
// Indicate if a Tag Check Fault should cause a synchronous exception,
// be asynchronously accumulated, or have no effect on the PE.

TCFType AArch64.EffectiveTCF(bits(2) el, boolean read)
    bits(2) tcf;

    Regime regime = TranslationRegime(el);

    case regime of
        when Regime_EL3 tcf = SCTRLR_EL3.TCF;
        when Regime_EL2 tcf = SCTRLR_EL2.TCF;
        when Regime_EL20 tcf = if el == EL0 then SCTRLR_EL2.TCF0 else SCTRLR_EL2.TCF1;
        when Regime_EL10 tcf = if el == EL0 then SCTRLR_EL1.TCF0 else SCTRLR_EL1.TCF1;
        otherwise Unreachable();

    if tcf == '11' then // Reserved value

```

```

if !IsFeatureImplemented(FEAT_MTE_ASYMFAULT) then
    (-,tcf) = ConstrainUnpredictableBits(Unpredictable\_RESTCF)

case tcf of
    when '00' // Tag Check Faults have no effect on the PE
        return TCFType Ignore;
    when '01' // Tag Check Faults cause a synchronous exception
        return TCFType Sync;
    when '10'
        if IsFeatureImplemented(FEAT_MTE_ASYNC) then
            // If asynchronous faults are implemented,
            // Tag Check Faults are asynchronously accumulated
            return TCFType Async;
        else
            // Otherwise, Tag Check Faults have no effect on the PE
            return TCFType Ignore;
    when '11'
        if IsFeatureImplemented(FEAT_MTE_ASYMFAULT) then
            // Tag Check Faults cause a synchronous exception on re
            // a read/write access, and are asynchronously accumula
            if read then
                return TCFType Sync;
            else
                return TCFType Async;
        else
            // Otherwise, Tag Check Faults have no effect on the PE
            return TCFType Ignore;
    otherwise
        Unreachable();

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```

// AArch64.InstructionAbort()
// =====

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if IsFeatureImplemented(FEAT_DoubleFault) then assert fault.statusc

    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = AArch64.SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         (IsFeatureImplemented(FEAT_RME) && fault.gpcf.
                            HCR_EL2.GPF == '1') ||
                         IsSecondStage(fault)));
        if PSTATE.EL == EL3 then
            target_el = EL3;
        elseif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;

```

```

    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el)
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    ExceptionRecord except = AArch64.AbortSyndrome(Exception_Instruction
                                                vaddress, target_el)
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

### Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```

// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PCalignment);
    except.vaddress = ThisInstrAddr(64);
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elseif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

### Library pseudocode for aarch64/exceptions/aborts/ AArch64.PhysicalSErrorTarget

```

// AArch64.PhysicalSErrorTarget()
// =====
// Returns a tuple of whether SError exception can be taken and, if so,
// (boolean, bits(2)) AArch64.PhysicalSErrorTarget()
//   boolean route_to_el3;
//   boolean route_to_el2;

// The exception is explicitly routed to EL3.
if PSTATE.EL != EL3 then
    route_to_el3 = (HaveEL(EL3) && EffectiveEA() == '1');
else
    route_to_el3 = FALSE;

// The exception is explicitly routed to EL2.
if !route_to_el3 && EL2Enabled() && PSTATE.EL == EL1 then
    route_to_el2 = (HCR_EL2.AMO == '1');
elseif !route_to_el3 && EL2Enabled() && PSTATE.EL == EL0 then
    route_to_el2 = (!IsInHost() && HCR_EL2.<TGE,AMO> != '00');
else
    route_to_el2 = FALSE;

// The exception is "masked".
boolean masked;

```

```

case PSTATE.EL of
    when EL3
        masked = (EffectiveEA() == '0' || PSTATE.A == '1');
    when EL2
        masked = (!route_to_el3 &&
                   (HCR_El2.<TGE,AMO> == '00' || PSTATE.A == '1'));
    when EL1, EL0
        masked = (!route_to_el3 && !route_to_el2 && PSTATE.A == '1');

// When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the masked exception is taken at EL3
if IsFeatureImplemented(FEAT_DoubleFault2) then
    bit nmea_bit;
    case PSTATE.EL of
        when EL3
            nmea_bit = SCR_El3.NMEA;
        when EL2
            nmea_bit = if IsSCLR2EL2Enabled() then SCLR2_El2.NMEA;
        when EL1
            nmea_bit = if IsSCLR2EL1Enabled() then SCLR2_El1.NMEA;
        when EL0
            if IsInHost() then
                nmea_bit = if IsSCLR2EL2Enabled() then SCLR2_El2.NMEA;
            else
                nmea_bit = if IsSCLR2EL1Enabled() then SCLR2_El1.NMEA;
    masked = masked && (nmea_bit == '0');

elseif IsFeatureImplemented(FEAT_DoubleFault) && PSTATE.EL == EL3 then
    bit nmea_bit = SCR_El3.NMEA AND EffectiveEA();
    masked = masked && (nmea_bit == '0');

boolean route_masked_to_el3;
boolean route_masked_to_el2;

if IsFeatureImplemented(FEAT_DoubleFault2) then
    // The masked exception is routed to EL2.
    route_masked_to_el2 = (EL2Enabled() && !route_to_el3 &&
                           IsHCRXEL2Enabled() && HCRX_El2.TMEA == '1' ||
                           ((PSTATE.EL == EL1 && (PSTATE.A == '1' || masked)) ||
                            (PSTATE.EL == EL0 && masked && !IsInHost()));

    // The masked exception is routed to EL3.
    route_masked_to_el3 = (HaveEL(EL3) && SCR_El3.TMEA == '1' &&
                           !(route_to_el2 || route_masked_to_el2) &&
                           ((PSTATE.EL IN {EL2, EL1} &&
                             (PSTATE.A == '1' || masked)) ||
                            (PSTATE.EL == EL0 && masked)));
else
    route_masked_to_el2 = FALSE;
    route_masked_to_el3 = FALSE;

// The exception is taken at EL3.
take_in_el3 = PSTATE.EL == EL3 && !masked;

// The exception is taken at EL2 or in the Host EL0.
take_in_el2_0 = ((PSTATE.EL == EL2 || IsInHost()) &&
                 !(route_to_el3 || route_masked_to_el3) && !masked);

// The exception is taken at EL1 or in the non-Host EL0.
take_in_el1_0 = ((PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) ||
                  !(route_to_el2 || route_masked_to_el2)) && !masked);

```

```

        !(route_to_el3 || route_masked_to_el3) && !masked)

bits(2) target_el;
if take_in_el3 || route_to_el3 || route_masked_to_el3 then
    masked = FALSE; target_el = EL3;
elsif take_in_el2_0 || route_to_el2 || route_masked_to_el2 then
    masked = FALSE; target_el = EL2;
elsif take_in_el1_0 then
    masked = FALSE; target_el = EL1;
else
    masked = TRUE; target_el = bits(2) UNKNOWN;

return (masked, target_el);

```

### **Library pseudocode for aarch64/exceptions/aborts/ AArch64.RaiseTagCheckFault**

```

// AArch64.RaiseTagCheckFault()
// =====
// Raise a Tag Check Fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, FaultRecord fault)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;

except = AArch64.AbortSyndrome(Exception_DataAbort, fault, va, targ
AArch64.TakeException(target_el, except, preferred_exception_return);

```

### **Library pseudocode for aarch64/exceptions/aborts/ AArch64.ReportTagCheckFault**

```

// AArch64.ReportTagCheckFault()
// =====
// Records a Tag Check Fault exception into the appropriate TFSR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    case el of
        when EL3 assert ttbr == '0'; TFSR_EL3.TF0      = '1';
        when EL2 if ttbr == '0' then TFSR_EL2.TF0      = '1'; else TFSR_EL2.TF0 = '0';
        when EL1 if ttbr == '0' then TFSR_EL1.TF0      = '1'; else TFSR_EL1.TF0 = '0';
        when EL0 if ttbr == '0' then TFSR_E0_EL1.TF0   = '1'; else TFSR_E0_EL1.TF0 = '0';

```

### **Library pseudocode for aarch64/exceptions/aborts/ AArch64.RouteToSErrorOffset**

```

// AArch64.RouteToSErrorOffset()
// =====
// Returns TRUE if synchronous External abort exceptions are taken to t
// appropriate Serror vector offset, and FALSE otherwise.

```

```

boolean AArch64.RouteToSErrorOffset(bits(2) target_el)
    if !IsFeatureImplemented(FEAT_DoubleFault) then return FALSE;

    bit ease_bit;
    case target_el of
        when EL3
            ease_bit = SCR_EL3.EASE;
        when EL2
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL2Enabled\(\)
                ease_bit = SCTLR2_EL2.EASE;
            else
                ease_bit = '0';
        when EL1
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL1Enabled\(\)
                ease_bit = SCTLR2_EL1.EASE;
            else
                ease_bit = '0';

    return (ease_bit == '1');

```

### Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```

// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elseif EL2Enabled\(\) && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

### Library pseudocode for aarch64/exceptions/aborts/ AArch64.SyncExternalAbortTarget

```

// AArch64.SyncExternalAbortTarget()
// =====
// Returns the target Exception level for a Synchronous External
// Data or Instruction Abort.

bits(2) AArch64.SyncExternalAbortTarget(FaultRecord fault)
    boolean route_to_el3;

    // The exception is explicitly routed to EL3
    if PSTATE.EL != EL3 then
        route_to_el3 = (HaveEL(EL3) && EffectiveEA() == '1');
    else
        route_to_el3 = FALSE;

```

```

// The exception is explicitly routed to EL2
bit tea_bit = (if IsFeatureImplemented(FEAT_RAS) && EL2Enabled() th

boolean route_to_el2;
if !route_to_el3 && EL2Enabled() && PSTATE.EL == EL1 then
    route_to_el2 = (tea_bit == '1' ||
                    fault.accessdesc.acctype == AccessType_NV2 ||
                    IsSecondStage(fault));

elseif !route_to_el3 && EL2Enabled() && PSTATE.EL == EL0 then
    route_to_el2 = (!IsInHost() && (HCR_EL2.TGE == '1' || tea_bit ==
IsSecondStage(fault)));
else
    route_to_el2 = FALSE;

boolean route_masked_to_el3;
boolean route_masked_to_el2;

if IsFeatureImplemented(FEAT_DoubleFault2) then
    // The masked exception is routed to EL2
    route_masked_to_el2 = (EL2Enabled() && !route_to_el3 &&
                           (PSTATE.EL == EL1 && PSTATE.A == '1') &&
                           IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1');

    // The masked exception is routed to EL3
    route_masked_to_el3 = (HaveEL(EL3) &&
                           !(route_to_el2 || route_masked_to_el2) &&
                           (PSTATE.EL IN {EL2, EL1} && PSTATE.A ==
                           SCR_EL3.TMEA == '1');
else
    route_masked_to_el2 = FALSE;
    route_masked_to_el3 = FALSE;

// The exception is taken at EL3
take_in_el3 = PSTATE.EL == EL3;

// The exception is taken at EL2 or in the Host EL0
take_in_el2_0 = ((PSTATE.EL == EL2 || IsInHost()) &&
                  !(route_to_el3 || route_masked_to_el3));

// The exception is taken at EL1 or in the non-Host EL0
take_in_el1_0 = ((PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) &&
                  !(route_to_el2 || route_masked_to_el2) &&
                  !(route_to_el3 || route_masked_to_el3));

bits(2) target_el;
if take_in_el3 || route_to_el3 || route_masked_to_el3 then
    target_el = EL3;
elseif take_in_el2_0 || route_to_el2 || route_masked_to_el2 then
    target_el = EL2;
elseif take_in_el1_0 then
    target_el = EL1;
else
    assert(FALSE);

return target_el;

```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

```

// AArch64.TagCheckFault()
// =====
// Handle a Tag Check Fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccessDescriptor accdesc)
    TCFType tcftype = AArch64.EffectiveTCF(accdesc.el, accdesc.read);

    case tcftype of
        when TCFType_Sync
            FaultRecord fault = NoFault();
            fault.accessdesc = accdesc;
            fault.write = accdesc.write;
            fault.statuscode = Fault_TagCheck;
            AArch64.RaiseTagCheckFault(vaddress, fault);
        when TCFType_Async
            AArch64.ReportTagCheckFault(accdesc.el, vaddress<55>);
        when TCFType_Ignore
            return;
        otherwise
            Unreachable();

```

### Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```

// BranchTargetException()
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_BranchTarget);
    except.syndrome<1:0> = PSTATE.BTYPE;
    except.syndrome<24:2> = Zeros(23); // RES0

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elseif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

### Library pseudocode for aarch64/exceptions/aborts/TCFType

```

// TCFType
// =====

enumeration TCFType { TCFType_Sync, TCFType_Async, TCFType_Ignore };

```

### Library pseudocode for aarch64/exceptions/aborts/TakeGPCEException

```

// TakeGPCEException()
// =====
// Report Granule Protection Exception faults

```

```

TakeGPCException(bits(64) vaddress, FaultRecord fault)
    assert IsFeatureImplemented(FEAT_RME);
    assert IsFeatureImplemented(FEAT_LSE);
    assert IsFeatureImplemented(FEAT_HAFDBS);
    assert IsFeatureImplemented(FEAT_DoubleFault);

ExceptionRecord except;

except.exceptiontype = Exception\_GPC;
except.vaddress = ZeroExtend(vaddress, 64);
except.paddress = fault.paddress;
except.pavalid = TRUE;

if IPAVlid(fault) then
    except.ipavalid = TRUE;
    except.NS = if fault.ipaddress.paspace == PAS\_NonSecure
        except.ipaddress = fault.ipaddress.address;
else
    except.ipavalid = FALSE;

if fault.accessdesc.acctype == AccessType\_GCS then
    except.syndrome2<8> = '1'; //GCS

// Populate the fields grouped in ISS
except.syndrome<24:22> = Zeros(3); // RES0
except.syndrome<21> = if fault.gpcfs2walk then '1' else '0'; //
if fault.accessdesc.acctype == AccessType\_IFETCH then
    except.syndrome<20> = '1'; // InD
else
    except.syndrome<20> = '0'; // InD
except.syndrome<19:14> = EncodeGPCSC(fault.gpcf); // GPCSC
if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType\_DC
    except.syndrome<13> = '1'; // VNCR
else
    except.syndrome<13> = '0'; // VNCR
except.syndrome<12:11> = '00'; // RES0
except.syndrome<10:9> = '00'; // RES0

if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_WN}
    except.syndrome<8> = '1'; // CM
else
    except.syndrome<8> = '0'; // CM

except.syndrome<7> = if fault.s2fs1walk then '1' else '0'; // S1PTW

if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_WN}
    except.syndrome<6> = '1'; // WnR
elseif fault.statuscode IN {Fault\_HWUpdateAccessFlag, Fault\_ExclusiveAccess}
    except.syndrome<6> = bit UNKNOWN; // WnR
elseif fault.accessdesc.atomicop && IsExternalAbort(fault) then
    except.syndrome<6> = bit UNKNOWN; // WnR
else
    except.syndrome<6> = if fault.write then '1' else '0'; // WnR

except.syndrome<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

bits(64) preferred_exception_return = ThisInstrAddr(64);
bits(2) target_el = EL3;

integer vect_offset;

```

```

if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el)
    vect_offset = 0x180;
else
    vect_offset = 0x0;

AArch64.TakeException(target_el, except, preferred_exception_return);

```

### **Library pseudocode for aarch64/exceptions/async/ AArch64.TakePhysicalFIQException**

```

// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalFIQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x100;
except = ExceptionSyndrome(Exception FIQ);

if route_to_el3 then
    AArch64.TakeException(EL3, except, preferred_exception_return,
elseif PSTATE.EL == EL2 || route_to_el2 then
    assert PSTATE.EL != EL3;
    AArch64.TakeException(EL2, except, preferred_exception_return,
else
    assert PSTATE.EL IN {EL0, EL1};
    AArch64.TakeException(EL1, except, preferred_exception_return,

```

### **Library pseudocode for aarch64/exceptions/async/ AArch64.TakePhysicalIRQException**

```

// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x80;

except = ExceptionSyndrome(Exception IRQ);

if route_to_el3 then
    AArch64.TakeException(EL3, except, preferred_exception_return,
elseif PSTATE.EL == EL2 || route_to_el2 then
    assert PSTATE.EL != EL3;
    AArch64.TakeException(EL2, except, preferred_exception_return,
else
    assert PSTATE.EL IN {EL0, EL1};
    AArch64.TakeException(EL1, except, preferred_exception_return,

```

## Library pseudocode for aarch64/exceptions/async/ AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakePhysicalSErrorException(boolean implicit_esb)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO
                     bits(64) preferred_exception_return = ThisInstrAddr(64));
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elseif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    except = ExceptionSyndrome(Exception_SError);
    bits(25) syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);
    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    except.syndrome = syndrome;
    AArch64.TakeException(target_el, except, preferred_exception_return);
```

## Library pseudocode for aarch64/exceptions/async/ AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enable

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    except = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/async/ AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enable
```

```

bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x80;

except = ExceptionSyndrome(Exception IRQ);

AArch64.TakeException(EL1, except, preferred_exception_return, vect)

```

### **Library pseudocode for aarch64/exceptions/async/ AArch64.TakeVirtualSErrorException**

```

// AArch64.TakeVirtualSErrorException()
// =====

AArch64.TakeVirtualSErrorException()

assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError

bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x180;
except = ExceptionSyndrome(Exception SError);

if IsFeatureImplemented(FEAT_RAS) then
    except.syndrome<24> = VSESR_EL2.IDS;
    except.syndrome<23:0> = VSESR_EL2.ISS;
else
    bits(25) syndrome = bits(25) IMPLEMENTATION_DEFINED "Virtual SE
    impdef_syndrome = syndrome<24> == '1';
    if impdef_syndrome then except.syndrome = syndrome;

ClearPendingVirtualSError();
AArch64.TakeException(EL1, except, preferred_exception_return, vect)

```

### **Library pseudocode for aarch64/exceptions/debug/ AArch64.BreakpointException**

```

// AArch64.BreakpointException()
// =====

AArch64.BreakpointException(FaultRecord fault)
assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr(64);
bits(2) target_el;
vect_offset = 0x0;
target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1

vaddress = bits(64) UNKNOWN;
except = AArch64.AbortSyndrome(Exception Breakpoint, fault, vaddress);
AArch64.TakeException(target_el, except, preferred_exception_return)

```

## Library pseudocode for aarch64/exceptions/debug/ AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    except.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(EL1, except, preferred_exception_return,
```

## Library pseudocode for aarch64/exceptions/debug/ AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// =====

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        except.syndrome<24> = '0';
    else
        except.syndrome<24> = '1';
        except.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
        except.syndrome<5:0> = '100010'; // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(EL1, except, preferred_exception_return,
```

## Library pseudocode for aarch64/exceptions/debug/ AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// =====
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called w
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddr

AArch64.TakeException(EL2, except, preferred_exception_return, vect
```

## Library pseudocode for aarch64/exceptions/debug/ AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled()) &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1

ExceptionRecord except;
if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == Ac
    except = AArch64.AbortSyndrome(Exception NV2Watchpoint, fault,
else
    except = AArch64.AbortSyndrome(Exception Watchpoint, fault, vac
AArch64.TakeException(target_el, except, preferred_exception_return)
```

## Library pseudocode for aarch64/exceptions/exceptions/ AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception_Uncategorized                      ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap                            ec = 0x01;
        when Exception_CP15RTTTrap                      ec = 0x03; assert from_32;
```

```

when Exception_CP15RRTTrap           ec = 0x04; assert from_32;
when Exception_CP14RRTTrap          ec = 0x05; assert from_32;
when Exception_CP14DTTrap          ec = 0x06; assert from_32;
when Exception_AdvSIMDFPAccessTrap ec = 0x07;
when Exception_FPIDTrap           ec = 0x08;
when Exception_PACTrap            ec = 0x09;
when Exception_LDST64BTrap         ec = 0x0A;
when Exception_TSTARTAccessTrap   ec = 0x1B;
when Exception_GPC                ec = 0x1E;
when Exception_CP14RRTTrap         ec = 0x0C; assert from_32;
when Exception_BranchTarget       ec = 0x0D;
when Exception_IllegalState        ec = 0x0E; il_is_valid = F;
when Exception_SupervisorCall      ec = 0x11;
when Exception_HypervisorCall     ec = 0x12;
when Exception_MonitorCall        ec = 0x13;
when Exception_SystemRegisterTrap ec = 0x18; assert !from_32;
when Exception_SystemRegister128Trap ec = 0x14; assert !from_32;
when Exception_SVEAccessTrap       ec = 0x19; assert !from_32;
when Exception_ERetTrap           ec = 0x1A; assert !from_32;
when Exception_PACFail            ec = 0x1C; assert !from_32;
when Exception_SMEAcessTrap       ec = 0x1D; assert !from_32;
when Exception_InstructionAbort   ec = 0x20; il_is_valid = F;
when Exception_PCALignment        ec = 0x22; il_is_valid = F;
when Exception_DataAbort          ec = 0x24;
when Exception_NV2DataAbort       ec = 0x25;
when Exception_SPALignment        ec = 0x26; il_is_valid = F;
when Exception_MemCpyMemSet       ec = 0x27;
when Exception_GCSFail            ec = 0x2D; assert !from_32;
when Exception_FPTrappedException ec = 0x28;
when ExceptionSError             ec = 0x2F; il_is_valid = F;
when Exception_Breakpoint         ec = 0x30; il_is_valid = F;
when Exception_SoftwareStep       ec = 0x32; il_is_valid = F;
when Exception_Watchpoint         ec = 0x34; il_is_valid = F;
when Exception_NV2Watchpoint      ec = 0x35; il_is_valid = F;
when Exception_SoftwareBreakpoint ec = 0x38;
when Exception_VectorCatch        ec = 0x3A; il_is_valid = F;
when Exception_PMU                ec = 0x3D;
otherwise                           Unreachable();

if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
    ec = ec + 1;

if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
    ec = ec + 4;
bit il;
if il_is_valid then
    il = if ThisInstrLength() == 32 then '1' else '0';
else
    il = '1';
assert from_32 || il == '1';                                // AArch64 instructions always have length 32 or 64

return (ec,il);

```

## Library pseudocode for aarch64/exceptions/exceptions/ AArch64.ReportException

```

// AArch64.ReportException()
// =====

```

```

// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord except, bits(2) target_el)

Exception exceptype = except.exceptype;

(ec,il) = AArch64.ExceptionClass(exceptype, target_el);
iss = except.syndrome;
iss2 = except.syndrome2;

// IL is not valid for Data Abort exceptions without valid instruction
if ec IN {0x24,0x25} && iss<24> == '0' then
    il = '1';

ESR_EL[target_el] = (Zeros(8) : // <63:56>
                        iss2      : // <55:32>
                        ec<5:0>   : // <31:26>
                        il        : // <25>
                        iss);       // <24:0>

if exceptype IN {
    Exception_InstructionAbort,
    Exception_PCAAlignment,
    Exception_DataAbort,
    Exception_NV2DataAbort,
    Exception_NV2Watchpoint,
    Exception_GPC,
    Exception_Watchpoint
} then
    FAR_EL[target_el] = except.vaddress;
else
    FAR_EL[target_el] = bits(64) UNKNOWN;

if except.ipavvalid then
    HPFAR_EL2<47:4> = except.ipaddress<55:12>;
    if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure
        HPFAR_EL2.NS = except.NS;
    else
        HPFAR_EL2.NS = '0';
elseif target_el == EL2 then
    HPFAR_EL2<47:4> = bits(44) UNKNOWN;

if except.pavvalid then
    bits(64) faultaddr = ZeroExtend(except.paddress.address, 64);
    if IsFeatureImplemented(FEAT_RME) then
        case except.paddress.paspace of
            when PAS_Secure      faultaddr<63:62> = '00';
            when PAS_NonSecure   faultaddr<63:62> = '10';
            when PAS_Root       faultaddr<63:62> = '01';
            when PAS_Realm      faultaddr<63:62> = '11';
        if exceptype == Exception_GPC then
            faultaddr<11:0> = Zeros(12);
        else
            faultaddr<63> = if except.paddress.paspace == PAS_NonSecure
                PFAR_EL[target_el] = faultaddr;
            elseif (IsFeatureImplemented(FEAT_PFAR) ||
                    (IsFeatureImplemented(FEAT_RME) && target_el == EL3)) then
                PFAR_EL[target_el] = bits(64) UNKNOWN;
    return;

```

## Library pseudocode for aarch64/exceptions/exceptions/ AArch64.ResetControlRegisters

```
// AArch64.ResetControlRegisters()
// =====
// Resets System registers and memory-mapped control registers that have
// reset values to those values.

AArch64.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64\(\);

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL\(EL3\) then
        PSTATE.EL = EL3;
    elseif HaveEL\(EL2\) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers
    // and other system components
AArch64.ResetControlRegisters\(cold\_reset\);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';                                // Select stack pointer
    PSTATE.<D,A,I,F> = '1111';                    // All asynchronous exceptions
    PSTATE.SS = '0';                                // Clear software step bit
    PSTATE.DIT = '0';                               // PSTATE.DIT is reset to 0 when
    PSTATE.IL = '0';                                // Clear Illegal Execution state

    if IsFeatureImplemented(FEAT_TME) then TSTATE.depth = 0;      // None

    // All registers, bits and fields not reset by the above pseudocode
    // below are UNKNOWN bitstrings after reset. In particular, the return
    // address and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined
AArch64.ResetGeneralRegisters\(\);
AArch64.ResetSIMDFPRegisters\(\);
AArch64.ResetSpecialRegisters\(\);
ResetExternalDebugRegisters\(cold\_reset\);

    bits(64) rv;                                     // IMPLEMENTATION DEFINED reset value

    if HaveEL\(EL3\) then
        rv = RVBAR_EL3;
    elseif HaveEL\(EL2\) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;
```

```

// The reset vector must be correctly aligned
constant integer pamax = AArch64.PAMax();
assert IsZero(rv<63:pamax>) && IsZero(rv<1:0>);

boolean branch_conditional = FALSE;
EDPRSR.R = '0'; // Leaving Reset State.
BranchTo(rv, BranchType_RESET, branch_conditional);

```

### Library pseudocode for aarch64/exceptions/ieeeefp/ AArch64.FPTrappedException

```

// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exception)
    except = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV"
            except.syndrome<23> = '1'; // TFV
        else
            except.syndrome<23> = '0'; // TFV
    else
        except.syndrome<23> = '1'; // TFV
    except.syndrome<10:8> = bits(3) UNKNOWN;
    if except.syndrome<23> == '1' then
        except.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,
    else
        except.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(EL1, except, preferred_exception_return,

```

### Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```

// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_HypervisorCall);

```

```

except.syndrome<15:0> = immediate;

if PSTATE.EL == EL3 then
    AArch64.TakeException(EL3, except, preferred_exception_return,
else
    AArch64.TakeException(EL2, except, preferred_exception_return,

```

### **Library pseudocode for aarch64/exceptions/syscalls/ AArch64.CallSecureMonitor**

```

// AArch64.CallSecureMonitor()
// =====

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    HSAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, except, preferred_exception_return, vect_

```

### **Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor**

```

// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == 1;
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_re
    elseif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(EL1, except, preferred_exception_return,

```

### **Library pseudocode for aarch64/exceptions/takeexception/ AArch64.TakeException**

```

// AArch64.TakeException()
// =====

```

```

// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in);
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) > UInt(PSTATE.EL);
if Halted() then
    AArch64.TakeExceptionInDebugState(target_el, exception_in);
    return;
ExceptionRecord except = exception_in;
boolean sync_errors;
boolean iesb_req;
if IsFeatureImplemented(FEAT_IESB) then
    sync_errors = SCTLR_EL[target_el].IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) then
        sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' &&
if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
    iesb_req = FALSE;
    sync_errors = FALSE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
    TMFailure cause;
    case except.exception_type of
        when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
        when Exception_Breakpoint cause = TMFailure_DBG;
        when Exception_Watchpoint cause = TMFailure_DBG;
        when Exception_SoftwareStep cause = TMFailure_DBG;
        otherwise cause = TMFailure_ERR;
    FailTransaction(cause, FALSE);

boolean brbe_source_allowed = FALSE;
bits(64) brbe_source_address = Zeros(64);
if IsFeatureImplemented(FEAT_BRBE) then
    brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
    brbe_source_address = preferred_exception_return;

if !IsFeatureImplemented(FEAT_ExS) || SCTLR_EL[target_el].EIS == '1'
    SynchronizeContext();

// If coming from AArch32 state, the top parts of the X[] registers
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
    ResetSVEState();
else
    MaybeZeroSVEUppers(target_el);

integer vect_offset = vect_offset_in;
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    elseif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);

```

```

    else
        lower_32 = ELUsingAAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState, 64);

    if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
        if (IsFeatureImplemented(FEAT_NV2) &&
            (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> ==
            spsr<3:2> = '10';
        else
            if IsFeatureImplemented(FEAT_NV) && HCR_EL2.<NV,NV1> == '10'
                spsr<3:2> = '10';

    if IsFeatureImplemented(FEAT_BTI) && !UsingAAArch32() then
        boolean zero_btype;
        // SPSR_ELx[].BTYPEn is only guaranteed valid for these exception types
        if except.exceptiontype IN {ExceptionSError, ExceptionIRQ, ExceptionFIQ,
                                    ExceptionSoftwareStep, ExceptionPCALI,
                                    ExceptionInstructionAbort, ExceptionExternalAbort,
                                    ExceptionVectorCatch, ExceptionSoftwareError,
                                    ExceptionIllegalState, ExceptionBranchError}
            zero_btype = FALSE;
        else
            zero_btype = ConstrainUnpredictableBool(Unpredictable_ZERO);
        if zero_btype then spsr<11:10> = '00';

    if (IsFeatureImplemented(FEAT_NV2) &&
        except.exceptiontype == ExceptionNV2DataAbort && target_el == EL3)
        // External aborts are configured to be taken to EL3
        except.exceptiontype = ExceptionDataAbort;
    if !(except.exceptiontype IN {ExceptionIRQ, ExceptionFIQ}) then
        AArch64.ReportException(except, target_el);

    if IsFeatureImplemented(FEAT_BRBE) then
        bits(64) brbe_target_address = VBAR_EL[target_el]<63:11>:vect_offset;
        BRBEEception(except, brbe_source_allowed, brbe_source_address,
                      brbe_target_address, target_el,
                      except.trappedsyscallinst);

    if IsFeatureImplemented(FEAT_GCS) then
        if PSTATE.EL == target_el then
            if GetCurrentEXLOCKEN() then
                PSTATE.EXLOCK = '1';
            else
                PSTATE.EXLOCK = '0';
        else
            PSTATE.EXLOCK = '0';

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR_ELx[] = spsr;
    ELR_ELx[] = preferred_exception_return;

    PSTATE.SS = '0';

```

```

        if IsFeatureImplemented(FEAT_NMI) && !ELUsingAArch32(target_el) then
            PSTATE.ALLINT = NOT SCTLR_ELx[].SPINTMASK;
        PSTATE.<D,A,I,F> = '1111';
        PSTATE.IL = '0';
        if from_32 then                                // Coming from AArch32
            PSTATE.IT = '00000000';
            PSTATE.T = '0';                            // PSTATE.J is RES0
        if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
            (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
            SCTLR_ELx[].SPAN == '0') then
                PSTATE.PAN = '1';
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
        if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR_ELx[].D;
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
        if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = '1';
        if IsFeatureImplemented(FEAT_SEBEP) then
            PSTATE.PPEND = '0';
            ShouldSetPPEND = FALSE;

        boolean branch_conditional = FALSE;
BranchTo(VBAR_ELx[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION,
CheckExceptionCatch(TRUE);                                // Check for debug even

        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();

```

## Library pseudocode for aarch64/exceptions/traps/ AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AARCH32 System register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >=
        bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

## Library pseudocode for aarch64/exceptions/traps/ AArch64.AArch32SystemAccessTrapSyndrome

```

// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC
// VMSR instructions, other than traps that are due to HCPT or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr,

```

```

ExceptionRecord except;

case ec of
    when 0x0    except = ExceptionSyndrome(Exception_Uncategorized);
    when 0x3    except = ExceptionSyndrome(Exception_CP15RTTrap);
    when 0x4    except = ExceptionSyndrome(Exception_CP15RRTTrap);
    when 0x5    except = ExceptionSyndrome(Exception_CP14RTTrap);
    when 0x6    except = ExceptionSyndrome(Exception_CP14DTTrap);
    when 0x7    except = ExceptionSyndrome(Exception_AdvSIMDFPAccess);
    when 0x8    except = ExceptionSyndrome(Exception_FPIDTrap);
    when 0xC    except = ExceptionSyndrome(Exception_CP14RRTTrap);
    otherwise   except = Unreachable();
endcase

bits(20) iss = Zeros(20);

if except.exceptype == Exception_Uncategorized then
    return except;
elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                           Exception_CP15RTTrap} then
    // Trapped MRC/MCR, VMRS on FPSID
    if except.exceptype != Exception_FPIDTrap then      // When trap
        iss<19:17> = instr<7:5>;                      // opc2
        iss<16:14> = instr<23:21>;                     // opc1
        iss<13:10> = instr<19:16>;                     // CRn
        iss<4:1>   = instr<3:0>;                      // CRm
    else
        iss<19:17> = '000';
        iss<16:14> = '111';
        iss<13:10> = instr<19:16>;                   // reg
        iss<4:1>   = '0000';

    if instr<20> == '1' && instr<15:12> == '1111' then // MRC, R
        iss<9:5> = '11111';
    elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, F
        iss<9:5> = bits(5) UNKNOWN;
    else
        iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    endif
elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccess,
                           Exception_CP15RRTTrap} then
    // Trapped MRRC/MCRR, VMRS/VMSR
    iss<19:16> = instr<7:4>;                      // opc1
    if instr<19:16> == '1111' then      // Rt2==15
        iss<14:10> = bits(5) UNKNOWN;
    else
        iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;
    endif
    if instr<15:12> == '1111' then      // Rt==15
        iss<9:5> = bits(5) UNKNOWN;
    else
        iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;                      // CRm
    endif
elsif except.exceptype == Exception_CP14DTTrap then
    // Trapped LDC/STC
    iss<19:12> = instr<7:0>;                      // imm8
    iss<4>     = instr<23>;                        // U
    iss<2:1>   = instr<24,21>;                     // P,W
    if instr<19:16> == '1111' then      // Rn==15, LDC(Literal address)
        iss<9:5> = bits(5) UNKNOWN;
        iss<3>   = '1';
    endif
    iss<0>     = instr<20>;                        // Direction
endif

```

```

except.syndrome<24:20> = ConditionSyndrome();
except.syndrome<19:0> = iss;

return except;

```

### Library pseudocode for aarch64/exceptions/traps/ AArch64.AdvSIMDFPAccessTrap

```

// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE ==

    if route_to_el2 then
        except = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        except.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, except, preferred_exception_re

    return;

```

### Library pseudocode for aarch64/exceptions/traps/ AArch64.CheckCP15InstrCoarseTraps

```

// AArch64.CheckCP15InstrCoarseTraps()
// =====
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCLTR_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CR
    trapped_encoding = ((CRn == 9 && CRm IN {0,1,2, 5,6,7,8 }) ||
                        (CRn == 10 && CRm IN {0,1, 4, 8 }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCLTR_EL1.TIDCP.
    if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && !IsInH
        !ELUsingAArch32(EL1) && SCLTR_EL1.TIDCP == '1' && trapped_en
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCLTR_EL2.TIDCP.
        if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && Is
            SCLTR_EL2.TIDCP == '1' && trapped_encoding) then
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);

```

```

major = if nreg == 1 then CRn else CRm;
// Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn>
// and MRC and MCR disabled by HCR_EL2.TIDCP.
if (!IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1'
    (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)
    if (PSTATE.EL == EL0 &&
        boolean IMPLEMENTATION_DEFINED "UNDEF unallocated C
        UNDEFINED;
AArch64.AArch32SystemAccessTrap(EL2, 0x3);

```

### **Library pseudocode for aarch64/exceptions/traps/ AArch64.CheckFPAdvSIMDEnabled**

```

// AArch64.CheckFPAdvSIMDEnabled()
// =====
AArch64.CheckFPAdvSIMDEnabled()
AArch64.CheckFPEEnabled();
// Check for illegal use of Advanced
// SIMD in Streaming SVE Mode
if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64
    SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);

```

### **Library pseudocode for aarch64/exceptions/traps/ AArch64.CheckFPAdvSIMDTrap**

```

// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if HaveEL(EL3) && CPTR_EL3.TFP == '1' && EL3SDDUndefPriority() then
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE ==
                when '11' disabled = FALSE;
                if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

AArch64.CheckFPAdvSIMDTrap();                                // Also check against C
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETA, ERETAB when either HCR
    // HFGITR_EL2.ERET is set, is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled()) &&
                    ((IsFeatureImplemented(FEAT_NV) && HCR_EL2.NV == '1'
                      || IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3)
                         || HFGITR_EL2.ERET == '1')));

    if route_to_el2 then
        ExceptionRecord except;
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then                                         // ERET
            except.syndrome<1> = '0';
            except.syndrome<0> = '0';                                // RES0
        else
            except.syndrome<1> = '1';
            if pac_uses_key_a then                                     // ERETA
                except.syndrome<0> = '0';
            else // ERETAB
                except.syndrome<0> = '1';
        AArch64.TakeException(EL2, except, preferred_exception_return,
```

## Library pseudocode for aarch64/exceptions/traps/ AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction
```

```

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (! (PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
            if IsFeatureImplemented(FEAT_NV) && HCR_EL2.NV == '1' && HC
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC
if route_to_el2 then
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;
    except = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome<15:0> = imm;
    except.trappedsyscallinst = TRUE;
    AArch64.TakeException(EL2, except, preferred_exception_return,

```

### Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```

// AArch64.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if IsFeatureImplemented(FEAT_FGT) then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                           EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                           (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3))

        elseif PSTATE.EL == EL1 then
            route_to_el2 = (EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
                           (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return,

```

### Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```

// AArch64.CheckForWFxTrap()
// =====
// Check for trap on WFE or WFI instruction

```

```

AArch64.CheckForWFxTrap(bits(2) target_el, WFxType wfxtYPE)
    assert HaveEL(target_el);

    boolean is_wfe = wfxtYPE IN {WFxType_WFE, WFxType_WFET};
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR_ELx [].nTWE else SCTLR_ELx [].nTWE);
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

    if trap then
        AArch64.WFxTrap(wfxtYPE, target_el);

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```

// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if
// EL0 or EL1 is active.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE;

        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        except = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, except, preferred_exception_return);
        elseif route_to_el2 then
            AArch64.TakeException(EL2, except, preferred_exception_return);
        else
            AArch64.TakeException(EL1, except, preferred_exception_return);

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```

// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, except, preferred_exception_return,
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

```

// AArch64.SystemAccessTrap()
// =====
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >=
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

    except = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

## Library pseudocode for aarch64/exceptions/traps/ AArch64.SystemAccessTrapSyndrome

```

// AArch64.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, int ec)
    ExceptionRecord except;
    bits(32) instr = instr_in;
    case ec of
        when 0x0                                // Trapped access due to unknown
            except = ExceptionSyndrome(Exception\_Uncategorized);
        when 0x7                                // Trapped access to SVE, Advanced SIMD, FPA
            except = ExceptionSyndrome(Exception\_AdvSIMDFPAccessTrap);
            except.syndrome<24:20> = ConditionSyndrome();
        when 0x14                               // Trapped access to 128-bit System registers
                                                // 128-bit System instruction.
            except = ExceptionSyndrome(Exception\_SystemRegister128Trap);
            instr = ThisInstr();
            except.syndrome<21:20> = instr<20:19>; // Op0
            except.syndrome<19:17> = instr<7:5>;   // Op2
            except.syndrome<16:14> = instr<18:16>; // Op1
            except.syndrome<13:10> = instr<15:12>; // CRn
            except.syndrome<9:6>   = instr<4:1>;   // Rt
            except.syndrome<4:1>   = instr<11:8>; // CRm
            except.syndrome<0>    = instr<21>;     // Direction
        when 0x18                               // Trapped access to System registers
            except = ExceptionSyndrome(Exception\_SystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome<21:20> = instr<20:19>; // Op0
            except.syndrome<19:17> = instr<7:5>;   // Op2
            except.syndrome<16:14> = instr<18:16>; // Op1
            except.syndrome<13:10> = instr<15:12>; // CRn
            except.syndrome<9:5>   = instr<4:0>;   // Rt
            except.syndrome<4:1>   = instr<11:8>; // CRm
            except.syndrome<0>    = instr<21>;     // Direction
        when 0x19                               // Trapped access to SVE System registers
            except = ExceptionSyndrome(Exception\_SVEAccessTrap);
        when 0x1D                               // Trapped access to SME System registers
            except = ExceptionSyndrome(Exception\_SMEAccessTrap);
        otherwise
            Unreachable();

    return except;

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.Undefined

```
// AArch64.Undefined()
// =====

AArch64.Undefined()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1'
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(EL1, except, preferred_exception_return,
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====

AArch64.WFxTrap(WFxType wfxttype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_WFxTrap);
    except.syndrome<24:20> = ConditionSyndrome();

    case wfxttype of
        when WFxType_WFI
            except.syndrome<1:0> = '00';
        when WFxType_WFE
            except.syndrome<1:0> = '01';
        when WFxType_WFIT
            except.syndrome<1:0> = '10';
            except.syndrome<2> = '1'; // Register field is valid
            except.syndrome<9:5> = ThisInstr()<4:0>;
        when WFxType_WFET
            except.syndrome<1:0> = '11';
            except.syndrome<2> = '1'; // Register field is valid
            except.syndrome<9:5> = ThisInstr()<4:0>;

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(target_el, except, preferred_exception_return,
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```

// CheckFPAdvSIMDEnabled()
// =====
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled()
    AArch64.CheckFPAdvSIMDEnabled\(\);

```

### Library pseudocode for aarch64/exceptions/traps/CheckFPEnabled64

```

// CheckFPEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPEnabled64()
    AArch64.CheckFPEnabled\(\);

```

### Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```

// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('10', 25); // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCLTR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_El2.TGE == '1' then EL2
            else
                trap = SCLTR_EL2.EnALS == '0';
                target_el = EL2;
        else
            target_el = EL1;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !ISHCRXEL2Enabled() || HCRX_El2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);

```

### Library pseudocode for aarch64/exceptions/traps/CheckST64BV0Enabled

```

// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('1', 25); // 0x1
    bits(2) target_el;

```

```

if (PSTATE.EL != EL3 && HaveEL(EL3) &&
    SCR_EL3.EnAS0 == '0' && EL3SDDUndefPriority()) then
    UNDEFINED;

if PSTATE.EL == EL0 then
    if !IsInHost() then
        trap = SCLTR_EL1.EnAS0 == '0';
        target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2
    else
        trap = SCLTR_EL2.EnAS0 == '0';
        target_el = EL2;

if (!trap && EL2Enabled() &&
    ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
    trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
    target_el = EL2;

if !trap && PSTATE.EL != EL3 then
    trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
    target_el = EL3;

if trap then
    if target_el == EL3 && EL3SDDUndef() then
        UNDEFINED;
    else
        LDST64BTrap(target_el, iss);

```

### Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```

// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = Zeros(25);
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCLTR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2
        else
            trap = SCLTR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);

```

### Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

```

// LDST64BTrap()
// =====

```

```

// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_LDST64BTrap);
    except.syndrome = iss;
    AArch64.TakeException(target_el, except, preferred_exception_return);

    return;

```

## Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```

// WFETrapDelay()
// =====
// Returns TRUE when delay in trap to WFE is enabled with value to amount
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTRLR_EL1.TWEDEn == '1';
                delay         = 1 << (UInt(SCTRLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTRLR_EL2.TWEDEn == '1';
                delay         = 1 << (UInt(SCTRLR_EL2.TWEDEL) + 8);
        when EL2
            assert EL2Enabled();
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay         = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay         = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

    return (delay_enabled, delay);

```

## Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```

// WaitForEventUntilDelay()
// =====
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);

```

## Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```

// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address
// exceptions taken to an Exception level using AArch64.

```

```

(bits(25), bits(24)) AArch64.FaultSyndrome(Exception exceptype,
                                             bits(64) vaddress)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros(25);
    bits(24) iss2 = Zeros(24);

    boolean d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort,
                                    Exception_Watchpoint, Exception_NV2Watchpoint}
    if IsFeatureImplemented(FEAT_RAS) && fault.statuscode == Fault_SyncErrorState
        errstate = AArch64.PEErrorState(fault);
        iss<12:11> = AArch64.EncodeSyncErrorSyndrome(errstate); // SETS

    if d_side then
        if fault.accessdesc.acctype == AccessType_GCS then
            iss2<8> = '1';
        if exceptype == Exception_Watchpoint then
            iss<23:0> = WatchpointRelatedSyndrome(fault, vaddress);
        if IsFeatureImplemented(FEAT_LS64) && fault.accessdesc.ls64 then
            if (fault.statuscode IN {Fault_AccessFlag, Fault_Translation,
                                    (iss2, iss<24:14>)) = LS64InstructionSyndrome();
        elsif (IsSecondStage(fault) && !fault.s2fs1walk &&
               (!IsExternalSyncAbort(fault) ||
                (!IsFeatureImplemented(FEAT_RAS) && fault.accessdesc.access
                 boolean IMPLEMENTATION_DEFINED "ISV on second stage translation"
                 iss<24:14> = LSErrorCode());
        if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype =
            iss<13> = '1'; // Fault is generated by use of VNCR_EL2

        if (IsFeatureImplemented(FEAT_LS64) &&
            fault.statuscode IN {Fault_AccessFlag, Fault_Translation,
            iss<12:11> = GetLoadStoreType());
        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_I,
            iss<8> = '1';

            if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_I,
                iss<6> = '1';
            elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_ExclusiveAccessFlag,
                iss<6> = bit UNKNOWN;
            elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
                iss<6> = bit UNKNOWN;
            else
                iss<6> = if fault.write then '1' else '0';

            if fault.statuscode == Fault_Permission then
                iss2<5> = if fault.dirtybit then '1' else '0';
                iss2<6> = if fault.overlay then '1' else '0';
                if iss<24> == '0' then
                    iss<21> = if fault.toplevel then '1' else '0';
                    iss2<7> = if fault.assuredonly then '1' else '0';
                    iss2<9> = if fault.tagaccess then '1' else '0';
                    iss2<10> = if fault.s1tagnotdata then '1' else '0';

            else
                if (fault.accessdesc.acctype == AccessType_IFETCH &&
                    fault.statuscode == Fault_Permission) then
                    iss2<5> = if fault.dirtybit then '1' else '0';

```

```

        iss<21> = if fault.toplevel then '1' else '0';
        iss2<7> = if fault.assuredonly then '1' else '0';
        iss2<6> = if fault.overlay then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

        return (iss, iss2);
    
```

### **Library pseudocode for aarch64/functions/aborts/EncodeGPCSC**

```

// EncodeGPCSC()
// =====
// Function that gives the GPCSC code for types of GPT Fault

bits(6) EncodeGPCSC(GPCFRecord gpcf)
    assert gpcf.level IN {0,1};

    case gpcf.gpf of
        when GPCF\_AddressSize return '00000':gpcf.level<0>;
        when GPCF\_Walk         return '00010':gpcf.level<0>;
        when GPCF\_Fail        return '00110':gpcf.level<0>;
        when GPCF\_EABT        return '01010':gpcf.level<0>;
    
```

### **Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome**

```

// LS64InstructionSyndrome()
// =====
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.

(bits(24), bits(11)) LS64InstructionSyndrome();
    
```

### **Library pseudocode for aarch64/functions/aborts/WatchpointFARNotPrecise**

```

// WatchpointFARNotPrecise()
// =====
// Returns TRUE If the lowest watchpointed address that is higher than
// recorded in EDWAR might not have been accessed by the instruction,
// UNPREDICTABLE condition of watchpoint matching a range of addresses
// rounded down and upper address rounded up to nearest 16 byte multiple
// FALSE otherwise.

boolean WatchpointFARNotPrecise(FaultRecord fault);
    
```

### **Library pseudocode for aarch64/functions/at/AArch64.AT**

```

// AArch64.AT()
// =====
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in,
    
```

```

TranslationStage stage = stage_in;
bits(2) el = el_in;
bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NSE();
if (IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3 &&
    effective_nse_ns == '10' && el != EL3) then
    UNDEFINED;
// For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and re
// the EL2&0 translation regime is used.
if EL2Enabled() && HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage
    el = EL2;
if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() the
    stage = TranslationStage_1;

SecurityState ss = SecurityStateAtEL(el);

accdesc = CreateAccDescAT(ss, el, ataccess);
aligned = TRUE;

FaultRecord fault = NoFault(accdesc);
Regime regime;
if stage == TranslationStage_12 then
    regime = Regime_EL10;
else
    regime = TranslationRegime(el);

AddressDescriptor addrdesc;
if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(EL0))
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, address,
                                                    accdesc);
    else
        (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime,
                                                       accdesc);
else
    (fault, addrdesc) = AArch64.S1Translate(fault, regime, address,
                                             accdesc);

if stage == TranslationStage_12 && fault.statuscode == Fault_None then
    boolean s1aarch64;
    if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled()
        addrdesc.vaddress = ZeroExtend(address, 64);
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned);
    elseif regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, s1aarch64);

is_ATS1Ex = stage != TranslationStage_12;
if fault.statuscode != Fault_None then
    addrdesc = CreateFaultyAddressDescriptor(address, fault);
    // Take an exception on:
    // * A Synchronous External abort occurs on translation table w
    // * A stage 2 fault occurs on a stage 1 walk
    // * A GPC Exception (FEAT_RME)
    // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.
    if (IsExternalAbort(fault) ||
        (PSTATE.EL == EL1 && fault.s2fs1walk) ||
        (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_ReportAsGPCEException(fault) ||
         (EL2Enabled() && HCR_EL2.GPF == '1' && PSTATE.EL == EL1) ||
         is_ATS1Ex)
        )) then

```

```

    PAR_EL1 = bits(128) UNKNOWN;
    AArch64.Abort(address, addrdesc.fault);

    AArch64.EncodePAR(regime, is_ATS1Ex, addrdesc);
    return;

```

## Library pseudocode for aarch64/functions/at/AArch64.EncodePAR

```

// AArch64.EncodePAR()
// =====
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, boolean is_ATS1Ex, AddressDescriptor addrdesc)
{
    PAR_EL1 = Zeros(128);
    paspace = addrdesc.paddress.paspace;

    if AArch64.isPARFormatD128(regime, is_ATS1Ex) then
        PAR_EL1.D128 = '1';
    else
        PAR_EL1.D128 = '0';

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        if IsFeatureImplemented(FEAT_RME) then
            if regime == Regime_EL3 then
                case paspace of
                    when PAS_Secure      PAR_EL1.<NSE,NS> = '00';
                    when PAS_NonSecure   PAR_EL1.<NSE,NS> = '01';
                    when PAS_Root        PAR_EL1.<NSE,NS> = '10';
                    when PAS_Realm       PAR_EL1.<NSE,NS> = '11';

                elseif SecurityStateForRegime(regime) == SS_Secure then
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';

                elseif SecurityStateForRegime(regime) == SS_Realm then
                    if regime == Regime_EL10 && is_ATS1Ex then
                        PAR_EL1.NSE = bit UNKNOWN;
                        PAR_EL1.NS = bit UNKNOWN;
                    else
                        PAR_EL1.NSE = bit UNKNOWN;
                        PAR_EL1.NS = if paspace == PAS_Realm then '0' else '1';

                    else
                        PAR_EL1.NSE = bit UNKNOWN;
                        PAR_EL1.NS = bit UNKNOWN;
                else
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS = bit UNKNOWN;
            else
                PAR_EL1<11> = '1'; // RES1
                if SecurityStateForRegime(regime) == SS_Secure then
                    PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
                else
                    PAR_EL1.NS = bit UNKNOWN;
                PAR_EL1.SH = ReportedPARShareability(PAREncodeShareability(addrdesc));
                if PAR_EL1.D128 == '1' then
                    PAR_EL1<119:76> = addrdesc.paddress.address<55:12>;
                else
                    PAR_EL1<55:12> = addrdesc.paddress.address<55:12>;
                PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattr));
            end;
        end;
    end;
}

```

```

        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
else
    PAR_EL1.F      = '1';
    PAR_EL1.DirtyBit   = if addrdesc.fault.dirtybit then '1' else '0';
    PAR_EL1.Overlay    = if addrdesc.fault.overlay then '1' else '0';
    PAR_EL1.TopLevel    = if addrdesc.fault.toplevel then '1' else '0';
    PAR_EL1.AssuredOnly = if addrdesc.fault.assuredonly then '1' else '0';
    PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
    PAR_EL1.PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
    PAR_EL1.S      = if addrdesc.fault.secondstage then '1' else '0';
    PAR_EL1<11> = '1'; // RES1
    PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR"
return;

```

### Library pseudocode for aarch64/functions/at/[AArch64.PARFaultStatus](#)

```

// AArch64.PARFaultStatus()
// =====
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    return fst;

```

### Library pseudocode for aarch64/functions/at/[AArch64.isPARFormatD128](#)

```

// AArch64.isPARFormatD128()
// =====
// Check if last stage of translation uses VMSAv9-128.
// Last stage of translation is stage 2 if enabled, else it is stage 1.

boolean AArch64.isPARFormatD128(Regime regime, boolean is_ATS1Ex)
    boolean isPARFormatD128;
    // Regime_EL2 does not support VMSAv9-128
    if regime == Regime_EL2 || !IsFeatureImplemented(FEAT_D128) then
        isPARFormatD128 = FALSE;
    else
        isPARFormatD128 = FALSE;
        case regime of
            when Regime_EL3
                isPARFormatD128 = TCR_EL3.D128 == '1';
            when Regime_EL20
                isPARFormatD128 = TCR2_EL2.D128 == '1';
            when Regime_EL10
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2.<VM,DC> == '00'
                    isPARFormatD128 = TCR2_EL1.D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2.D128 == '1';

    return isPARFormatD128;

```

### **Library pseudocode for aarch64/functions/at/GetPAR\_EL1\_D128**

```

// GetPAR_EL1_D128()
// =====
// Query the PAR_EL1.D128 field

bit GetPAR_EL1_D128()
    bit D128;

    D128 = PAR_EL1.D128;
    return D128;

```

### **Library pseudocode for aarch64/functions/at/GetPAR\_EL1\_F**

```

// GetPAR_EL1_F()
// =====
// Query the PAR_EL1.F field.

bit GetPAR_EL1_F()
    bit F;

    F = PAR_EL1.F;
    return F;

```

### **Library pseudocode for aarch64/functions/barrierop/MemBarrierOp**

```

// MemBarrierOp
// =====
// Memory barrier instruction types.

```

```

enumeration MemBarrierOp { MemBarrierOp_DSB           // Data Synchronization
                           , MemBarrierOp_DMB           // Data Memory Barrier
                           , MemBarrierOp_ISB           // Instruction Synchronization Barrier
                           , MemBarrierOp_SSSB          // Speculative Synchronization Barrier
                           , MemBarrierOp_PSSBB         // Speculative Synchronization Barrier
                           , MemBarrierOp_SB            // Speculation Barrier
};

```

### Library pseudocode for aarch64/functions/bfxpreferred/BFXPreferred

```

// BFXPreferred()
// =====
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;

```

### Library pseudocode for aarch64/functions/bitmasks/AltDecodeBitMasks

```

// AltDecodeBitMasks()
// =====
// Alternative but logically equivalent implementation of DecodeBitMask
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                       boolean immediate, integer M)

bits(64) tmask, wmask;
bits(6) tmask_and, wmask_and;
bits(6) tmask_or, wmask_or;
bits(6) levels;

// Compute log2 of element size
// 2^len must be in range [2, M]
len = HighestSetBit(immN:NOT(imms));
if len < 1 then UNDEFINED;

```

```

assert M >= (1 << len);

// Determine s, r and s - r parameters
levels = ZeroExtend(Ones(len), 6);

// For logical immediates an all-ones value of s is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then
    UNDEFINED;

s = UInt(imms AND levels);
r = UInt(immr AND levels);
diff = s - r;      // 6-bit subtract with borrow

// Compute "top mask"
tmask_and = diff<5:0> OR NOT(levels);
tmask_or = diff<5:0> AND levels;

tmask = Ones(64);
tmask = ((tmask
            AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
            OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
// optimization of first step:
// tmask = Replicate(tmask_and<0> : '1', 32);
tmask = ((tmask
            AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
            OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
tmask = ((tmask
            AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
            OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
tmask = ((tmask
            AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
            OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
tmask = ((tmask
            AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
            OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
tmask = ((tmask
            AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
            OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
            AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
            OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
            AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
            OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
            AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
            OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
            AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
            OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask

```

```

        AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
        OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2)
wmask = ((wmask
        AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
        OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1))

if diff<6> != '0' then // borrow from s - r
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<M-1:0>);

```

## Library pseudocode for aarch64/functions/bitmasks/DecodeBitMasks

```

// DecodeBitMasks()
// =====
// Decode AArch64 bitfield and logical immediate masks which use a similar
(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)
bits(M) tmask, wmask;
bits(6) levels;

// Compute log2 of element size
// 2^len must be in range [2, M]
constant integer len = HighestSetBit(immN:NOT(imms));
if len < 1 then UNDEFINED;
assert M >= (1 << len);

// Determine s, r and s - r parameters
levels = ZeroExtend(Ones(len), 6);

// For logical immediates an all-ones value of s is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then
    UNDEFINED;

s = UInt(imms AND levels);
r = UInt(immr AND levels);
constant integer diff = s - r;      // 6-bit subtract with borrow

constant integer esize = 1 << len;
d = UInt(diff<len-1:0>);
welem = ZeroExtend(Ones(s + 1), esize);
telem = ZeroExtend(Ones(d + 1), esize);
wmask = Replicate(ROR(welem, r), M DIV esize);
tmask = Replicate(telem, M DIV esize);
return (wmask, tmask);

```

## Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```

// AArch64.DataMemZero()
// =====
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor

```

```

AccessDescriptor accdesc = accdesc_in;

// If the instruction targets tags as a payload, confer with system
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagaccess then
    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdes

// If the instruction encoding permits tag checking, confer with system
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(vaddress, accde

boolean aligned = TRUE;
AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress,

if IsFault(memaddrdesc) then
    if IsDebugException(memaddrdesc.fault) then
        AArch64.Abort(vaddress, memaddrdesc.fault);
    else
        AArch64.Abort(regval, memaddrdesc.fault);

if IsFeatureImplemented(FEAT_TME) then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddr
        FailTransaction(TMFailure_IMP, FALSE);

for i = 0 to size-1
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(vaddress);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            if (boolean IMPLEMENTATION_DEFINED
                "DC_ZVA tag fault reported with lowest faulting address"
                AArch64.TagCheckFault(vaddress, accdesc);
            else
                AArch64.TagCheckFault(regval, accdesc);
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1
    return;

```

## Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```

// AArch64.TagMemZero()
// =====
// Write Zero to tag memory.

AArch64.TagMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

AccessDescriptor accdesc = accdesc_in;

integer count = size >> LOG2_TAG_GRANULE;
bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
boolean aligned = IsAligned(vaddress, TAG_GRANULE);

// Stores of allocation tags must be aligned
if !aligned then

```

```

AArch64.Abort(vaddress, AlignmentFault(accdesc));

if IsFeatureImplemented(FEAT_MTE2) then
    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdes

memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned,

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    if IsDebugException(memaddrdesc.fault) then
        AArch64.Abort(vaddress, memaddrdesc.fault);
    else
        AArch64.Abort(regval, memaddrdesc.fault);

if !accdesc.tagaccess || memaddrdesc.memattrs.tags != MemTag_Allocat
    return;

for i = 0 to count-1
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc

    memaddrdesc.paddress.address = memaddrdesc.paddress.address + I

    return;

```

## Library pseudocode for aarch64/functions/compareop/CompareOp

```

// CompareOp
// ======
// Vector compare instruction types.

enumeration CompareOp { CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                        CompareOp_LE, CompareOp_LT };

```

## Library pseudocode for aarch64/functions/countop/CountOp

```

// CountOp
// ======
// Bit counting instruction types.

enumeration CountOp { CountOp_CLZ, CountOp_CLS, CountOp_CNT };

```

## Library pseudocode for aarch64/functions/d128/IsD128Enabled

```

// IsD128Enabled()
// =====
// Returns true if 128-bit page descriptor is enabled

boolean IsD128Enabled(bits(2) el)
    boolean d128enabled;
    if IsFeatureImplemented(FEAT_D128) then
        case el of
            when EL0
                if !ELIsInHost(EL0) then

```

```

        d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 ==
    else
        d128enabled = IsTCR2EL2Enabled() && TCR2_EL2.D128 ==
    when EL1
        d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1'
    when EL2
        d128enabled = IsTCR2EL2Enabled() && HCR_EL2.E2H == '1'
    when EL3
        d128enabled = TCR_EL3.D128 == '1';
else
    d128enabled = FALSE;

return d128enabled;

```

## Library pseudocode for aarch64/functions/dc/AArch64.DC

```

// AArch64.DC()
// =====
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope = opscope_in,
CacheRecord cache;

cache.acctype = AccessType_DC;
cache.cachetype = cachetype;
cache.cacheop = cacheop;
cache.opscope = opscope;

if opscope == CacheOpScope_SetWay then
    ss = SecurityStateAtEL(PSTATE.EL);
    cache.cpas = CPASAtSecurityState(ss);
    cache.shareability = Shareability_NSH;
    (cache.setnum, cache.waynum, cache.level) = DecodeSW(regval, ca
    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Ena
        (HCR_EL2.SWIO == '1' || HCR_EL2.<DC, VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;

if EL2Enabled() && !IsInHost() then
    if PSTATE.EL IN {EL0, EL1} then
        cache.is_vmid_valid = TRUE;
        cache.vmid = VMID[];
    else
        cache.is_vmid_valid = FALSE;
else
    cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid = ASID[];
else
    cache.is_asid_valid = FALSE;

if (opscope == CacheOpScope_PoDP &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not support PoDP"

```

```

        opscope = CacheOpScope_PoP;
if (opscope == CacheOpScope_PoP) &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not support PoP";
    opscope = CacheOpScope_PoC;
vaddress = regval;

size = 0;           // by default no watchpoint address
if cacheop == CacheOp_Invalidate then
    size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Word Size";
    assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
    vaddress = Align(regval, size);

if DCInstNeedsTranslation(opscope) then
    cache.vaddress = vaddress;
    boolean aligned = TRUE;
    AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress);
    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    cache.translated = TRUE;
    cache.paddress = memaddrdesc.paddress;
    cache.cpas = CPASATPAS(memaddrdesc.paddress.paspace);
    if opscope IN {CacheOpScope_PoC, CacheOpScope_PoP, CacheOpScope_PoE}
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability_NSH;
elseif opscope == CacheOpScope_PoE then
    cache.translated = TRUE;
    cache.shareability = Shareability_OSH;
    cache.paddress.address = regval<55:0>;
    cache.paddress.paspace = DecodePASpace(regval<62>, regval<63>);
    cache.cpas = CPASATPAS(cache.paddress.paspace);

// If a Reserved encoding is selected, the instruction is permitted
if cache.paddress.paspace != PAS_Reserved then
    EndOfInstruction();

if boolean IMPLEMENTATION_DEFINED "Apply granule protection checks"
    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = CreateAccDescDC(cache);
    memaddrdesc.paddress = cache.paddress;
    memaddrdesc.fault.gpcf = GranuleProtectionCheck(memaddrdesc);

    if memaddrdesc.fault.gpcf != GPCF_None then
        memaddrdesc.fault.statuscode = Fault_GPCFOOnOutput;
        memaddrdesc.fault.paddress = memaddrdesc.paddress;
        AArch64.Abort(bits(64) UNKNOWN, memaddrdesc.fault);
elseif opscope == CacheOpScope_PoPA then
    cache.translated = TRUE;
    cache.shareability = Shareability_OSH;
    cache.paddress.address = regval<55:0>;
    cache.paddress.paspace = DecodePASpace(regval<62>, regval<63>);
    cache.cpas = CPASATPAS(cache.paddress.paspace);
else
    cache.vaddress = vaddress;
    cache.translated = FALSE;
    cache.shareability = Shareability_UNKNOWN;
    cache.paddress = FullAddress_UNKNOWN;

```

```

if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled
    HCR_EL2.<DC,VM> != '00') then
    cache.cacheop = CacheOp_CleanInvalidate;

// If Secure state is not implemented, but RME is, the instruction
if cache.translated && cache.cpas == CPAS_Secure && !HaveSecureStat
    return;

CACHE_OP(cache);
return;

```

## Library pseudocode for aarch64/functions/dc/AArch64.MemZero

```

// AArch64.MemZero()
// =====

AArch64.MemZero(bits(64) regval, CacheType cachetype)
    integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    assert size <= MAX_ZERO_BLOCK_SIZE;
    if IsFeatureImplemented(FEAT_MTE2) then
        assert size >= TAG_GRANULE;

    bits(64) vaddress = Align(regval, size);

    boolean tagaccess = cachetype IN {CacheType_Tag, CacheType_Data_Tac};
    boolean tagchecked = cachetype == CacheType_Data;
    AccessDescriptor accdesc = CreateAccDescDCZero(tagaccess, tagchecked);

    if cachetype IN {CacheType_Tag, CacheType_Data_Tag} then
        AArch64.TagMemZero(regval, vaddress, accdesc, size);

    if cachetype IN {CacheType_Data, CacheType_Data_Tag} then
        AArch64.DataMemZero(regval, vaddress, accdesc, size);
    return;

```

## Library pseudocode for aarch64/functions/dc/MemZero

```
constant integer MAX_ZERO_BLOCK_SIZE = 2048;
```

## Library pseudocode for aarch64/functions/eret/AArch64.ExceptionReturn

```

// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if IsFeatureImplemented(FEATIESB) then
        sync_errors = SCLTR_ELx[].IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' &&
        if sync_errors then

```

```

    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    boolean brbe_source_allowed = FALSE;
    bits(64) brbe_source_address = Zeros(64);
    if IsFeatureImplemented(FEAT_BRBE) then
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = PC64;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_ELx[].EOS == '1' then
        SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal E
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elseif UsingAArch32() then                                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                                     // T32
        else
            new_pc<1:0> = '00';                                    // A32
    else
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEEExceptionReturn(new_pc, source_el,
                               brbe_source_allowed, brbe_source_address);

    if UsingAArch32() then
        if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then Rese
            // 32 most significant bits are ignored.
            boolean branch_conditional = FALSE;
            BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
        else
            BranchToAddr(new_pc, BranchType_ERET);
    
```

CheckExceptionCatch(FALSE); // Check for debug event c

## Library pseudocode for aarch64/functions/exclusive/ AArch64.ExclusiveMonitorsPass

```

// AArch64.ExclusiveMonitorsPass()
// =====
// Return TRUE if the Exclusives monitors for the current PE include all
// associated with the virtual address region of size bytes starting at
// The immediately following memory write must be to the same addresses

```

```

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts
// before or after the check on the local Exclusives monitor. As a result
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size, AArch64.ACCDESC accdesc)
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size)
        AArch64.Abort(address, AlignmentFault(accdesc));

    if !AArch64.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;

```

### **Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA**

```

// AArch64.IsExclusiveVA()
// =====
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

```

### **Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA**

```

// AArch64.MarkExclusiveVA()
// =====
// Optionally record an exclusive access to the virtual address region
// starting at address for processorid.

AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);

```

## Library pseudocode for aarch64/functions/exclusive/ AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// =====
// Sets the Exclusives monitors for the current PE to record the address
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel,
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size)
        AArch64.Abort(address, AlignmentFault(accdesc));

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch64/functions/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

## Library pseudocode for aarch64/functions/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg, N];
```

```

boolean unsigned;
integer len;

case exttype of
    when ExtendType_SXTB unsigned = FALSE; len = 8;
    when ExtendType_SXTH unsigned = FALSE; len = 16;
    when ExtendType_SXTW unsigned = FALSE; len = 32;
    when ExtendType_SXTX unsigned = FALSE; len = 64;
    when ExtendType_UXTB unsigned = TRUE; len = 8;
    when ExtendType_UXTH unsigned = TRUE; len = 16;
    when ExtendType_UXTW unsigned = TRUE; len = 32;
    when ExtendType_UXTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

constant integer nbits = Min(len, N - shift);
return Extend(val<nbits-1:0> : Zeros(shift), N, unsigned);

```

### **Library pseudocode for aarch64/functions/extendreg/ExtendType**

```

// ExtendType
// =====
// AArch64 register extend and shift.

enumeration ExtendType {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW,
                        ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW}

```

### **Library pseudocode for aarch64/functions/fpconvop/FPConvOp**

```

// FPConvOp
// =====
// Floating-point convert/move instruction types.

enumeration FPConvOp {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                      FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
                     , FPConvOp_CVT_FtoI_JS
};

```

### **Library pseudocode for aarch64/functions/fpmaxminop/FPMaxMinOp**

```

// FPMaxMinOp
// =====
// Floating-point min/max instruction types.

enumeration FPMaxMinOp {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                        FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};

```

### **Library pseudocode for aarch64/functions/fpunaryop/FPUnaryOp**

```

// FPUnaryOp
// =====
// Floating-point unary instruction types.

enumeration FPUnaryOp {FPUnaryOp_ABS, FPUnaryOp_MOV,
FPUnaryOp_NEG, FPUnaryOp_SQRT};

```

## Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```

// FPRSqrtStepFused()
// =====

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    boolean fpexc = !altp;                                // Generate no float
    if altp then fpcr.<FIZ,FZ> = '11';                  // Flush denormal i
    if altp then fpcr.RMode = '00';                      // Use RNE rounding

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPPprocessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;

```

## Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```

// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)

```

```

assert N IN {16, 32, 64};
bits(N) op1 = op1_in;
bits(N) result;
boolean done;
FPCRTType fpcr = FPCR[];
op1 = FPNeg(op1);

boolean altpf = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
boolean fpexc = !altpf; // Generate no float exception
if altpf then fpcr.<FIZ,FZ> = '11'; // Flush denormal instead of signaling
if altpf then fpcr.RMode = '00'; // Use RNE rounding mode

(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
FPRounding rounding = FPRoundingMode(fpcr);

if !done then
    inf1 = (type1 == FPTYPE_INFINITY);
    inf2 = (type2 == FPTYPE_INFINITY);
    zero1 = (type1 == FPTYPE_ZERO);
    zero2 = (type2 == FPTYPE_ZERO);

    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo('0', N);
    elsif inf1 || inf2 then
        result = FPInfinity(sign1 EOR sign2, N);
    else
        // Fully fused multiply-add
        result_value = 2.0 + (value1 * value2);
        if result_value == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign, N);
        else
            result = FPRound(result_value, fpcr, rounding, fpexc, N);
    end
end
return result;

```

## Library pseudocode for aarch64/functions/gcs/AddGCSExRecord

```

// AddGCSExRecord()
// =====
// Generates and then writes an exception record to the
// current Guarded control stack.

AddGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_STORE)

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = lr;
    Mem[ptr-16, 8, accdesc] = spsr;
    Mem[ptr-24, 8, accdesc] = elr;
    Mem[ptr-32, 8, accdesc] = Zeros(60):'1001';

```

```

// Decrement the pointer value
ptr = ptr - 32;

SetCurrentGCSPointer(ptr);
return;

```

### Library pseudocode for aarch64/functions/gcs/AddGCSRecord

```

// AddGCSRecord()
// =====
// Generates and then writes a record to the current Guarded
// control stack.

AddGCSRecord(bits(64) vaddress)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_STORE)

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPointer(ptr);
    return;

```

### Library pseudocode for aarch64/functions/gcs/CheckGCSExRecord

```

// CheckGCSExRecord()
// =====
// Validates the provided values against the top entry of the
// current Guarded control stack.

CheckGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr, GCSInstruction
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(gcsinst_type);

    // Check the ELR matches the recorded value
    bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    if recorded_elr != elr then
        GCSDataCheckException(gcsinst_type);

    // Check the SPSR matches the recorded value
    bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    if recorded_spsr != spsr then
        GCSDataCheckException(gcsinst_type);

    // Check the LR matches the recorded value
    bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];

```

```

    if recorded_lr != lr then
        GCSDataCheckException(gcsinst_type);

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;

```

## Library pseudocode for aarch64/functions/gcs/CheckGCSSTREnabled

```

// CheckGCSSTREnabled()
// =====
// Trap GCSSTR or GCSSTTR instruction if trapping is enabled.

CheckGCSSTREnabled()
    case PSTATE.EL of
        when EL0
            if GCSCRE0_EL1.STRen == '0' then
                if EL2Enabled\(\) && HCR_EL2.TGE == '1' then
                    GCSSTRTrapException(EL2);
                else
                    GCSSTRTrapException(EL1);
        when EL1
            if GCSCR_EL1.STRen == '0' then
                GCSSTRTrapException(EL1);
            elseif (EL2Enabled\(\) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')
                HFGITR_EL2.nGCSSTR_EL1 == '0') then
                GCSSTRTrapException(EL2);
        when EL2
            if GCSCR_EL2.STRen == '0' then
                GCSSTRTrapException(EL2);
        when EL3
            if GCSCR_EL3.STRen == '0' then
                GCSSTRTrapException(EL3);
    return;

```

## Library pseudocode for aarch64/functions/gcs/EXLOCKException

```

// EXLOCKException()
// =====
// Handle an EXLOCK exception condition.

EXLOCKException()
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_GCSFail);
    except.syndrome<24> = Zeros();
    except.syndrome<23:20> = '0001';
    except.syndrome<19:0> = Zeros();
    AArch64.TakeException(PSTATE.EL, except, preferred_exception_return);

```

## Library pseudocode for aarch64/functions/gcs/GCSDataCheckException

```

// GCSDataCheckException()
// =====
// Handle a Guarded Control Stack data check fault condition.

GCSDataCheckException(GCSInstruction gcsinst_type)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    boolean rn_unknown = FALSE;
    boolean is_ret = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if (EL2Enabled() && HCR_EL2.TGE == '1') then EL2
    else
        target_el = PSTATE.EL;
    except = ExceptionSyndrome(Exception\_GCSFail);
    case gcsinst_type of
        when GCSInstType\_PRET
            except.syndrome<4:0> = '00000';
            is_ret = TRUE;
        when GCSInstType\_POPM
            except.syndrome<4:0> = '00001';
        when GCSInstType\_PRETAA
            except.syndrome<4:0> = '00010';
            is_ret = TRUE;
        when GCSInstType\_PRETAB
            except.syndrome<4:0> = '00011';
            is_ret = TRUE;
        when GCSInstType\_SS1
            except.syndrome<4:0> = '00100';
        when GCSInstType\_SS2
            except.syndrome<4:0> = '00101';
            rn_unknown = TRUE;
        when GCSInstType\_POPCX
            rn_unknown = TRUE;
            except.syndrome<4:0> = '01000';
        when GCSInstType\_POPX
            except.syndrome<4:0> = '01001';
        if rn_unknown == TRUE then
            except.syndrome<9:5> = bits(5) UNKNOWN;
        elsif is_ret == TRUE then
            except.syndrome<9:5> = ThisInstr()<9:5>;
        else
            except.syndrome<9:5> = ThisInstr()<4:0>;
        except.syndrome<24:10> = Zeros();
        except.vaddress = bits(64) UNKNOWN;
        AArch64.TakeException(target_el, except, preferred_exception_return);

```

## Library pseudocode for aarch64/functions/gcs/GCSEnabled

```

// GCSEnabled()
// =====
// Returns TRUE if the Guarded control stack is enabled at
// the provided Exception level.

boolean GCSEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;

```

```

if HaveEL(EL3) && el != EL3 && SCR_EL3.GCSEn == '0' then
    return FALSE;

if (el IN {EL0, EL1} && EL2Enabled() &&
    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.GCSEn == '0')) then
    return FALSE;

return GCSPCRSelected(el);

```

### Library pseudocode for aarch64/functions/gcs/GCSInstruction

```

// GCSInstruction
// =====

enumeration GCSInstruction {
    GCSInstType_PRET,      // Procedure return without Pointer authentication
    GCSInstType_POPM,      // GCSPOPM instruction
    GCSInstType_PRETAA,    // Procedure return with Pointer authentication
    GCSInstType_PRETAB,    // Procedure return with Pointer authentication
    GCSInstType_SS1,       // GCSSS1 instruction
    GCSInstType_SS2,       // GCSSS2 instruction
    GCSInstType_POPCX,    // GCSPOPCX instruction
    GCSInstType_POPX       // GCSPOPX instruction
};

```

### Library pseudocode for aarch64/functions/gcs/GCSPCREnabled

```

// GCSPCREnabled()
// =====
// Returns TRUE if the Guarded control stack is PCR enabled
// at the provided Exception level.

boolean GCSPCREnabled(bits(2) el)
    return GCSPCRSelected(el) && GCEEnabled(el);

```

### Library pseudocode for aarch64/functions/gcs/GCSPCRSelected

```

// GCSPCRSelected()
// =====
// Returns TRUE if the Guarded control stack is PCR selected
// at the provided Exception level.

boolean GCSPCRSelected(bits(2) el)
    case el of
        when EL0 return GCSCRE0_EL1.PCRSEL == '1';
        when EL1 return GCSCR_EL1.PCRSEL == '1';
        when EL2 return GCSCR_EL2.PCRSEL == '1';
        when EL3 return GCSCR_EL3.PCRSEL == '1';
        Unreachable();
    return TRUE;

```

## Library pseudocode for aarch64/functions/gcs/GCSPOPCX

```
// GCSPOPCX()
// =====
// Called to pop and compare a Guarded control stack exception return record.

GCSPOPCX()
    bits(64) spsr = SPSR_ELx[];
    if !GCSEnabled(PSTATE.EL) then
        EndOfInstruction();
    CheckGCSExRecord(ELR_ELx[], spsr, X[30,64], GCSInstType_POPCX);
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPM

```
// GCSPOPM()
// =====
// Called to pop a Guarded control stack procedure return record.

bits(64) GCSPOPM()
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);

    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    ptr = GetCurrentGCSPointer();
    bits(64) gcs_entry = Mem[ptr, 8, accdesc];

    if gcs_entry<1:0> != '00' then
        GCSDataCheckException(GCSInstType_POPM);

    ptr = ptr + 8;
    SetCurrentGCSPointer(ptr);
    return gcs_entry;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPX

```
// GCSPOPX()
// =====
// Called to pop a Guarded control stack exception return record.

GCSPOPX()
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();

    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(GCSInstType_POPX);

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
```

```

bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];

// Increment the pointer value
ptr = ptr + 32;

SetCurrentGCSPointer(ptr);
return;

```

### Library pseudocode for aarch64/functions/gcs/GCSPUSHM

```

// GCSPUSHM()
// =====
// Called to push a Guarded control stack procedure return record.

GCSPUSHM(bits(64) value)
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    AddGCSRecord(value);
    return;

```

### Library pseudocode for aarch64/functions/gcs/GCSPUSHX

```

// GCSPUSHX()
// =====
// Called to push a Guarded control stack exception return record.

GCSPUSHX()
    bits(64) spsr = SPSR_ELx[];
    if !GCSEnabled(PSTATE.EL) then
        EndOfInstruction();
    AddGCSExRecord(ELR_ELx[], spsr, X[30,64]);
    PSTATE.EXLOCK = '0';
    return;

```

### Library pseudocode for aarch64/functions/gcs/GCSReturnValueCheckEnabled

```

// GCSReturnValueCheckEnabled()
// =====
// Returns TRUE if the Guarded control stack has return value
// checking enabled at the current Exception level.

boolean GCSReturnValueCheckEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;
    case el of
        when EL0 return GCSCRE0_EL1.RVCHKEN == '1';
        when EL1 return GCSCR_EL1.RVCHKEN == '1';
        when EL2 return GCSCR_EL2.RVCHKEN == '1';
        when EL3 return GCSCR_EL3.RVCHKEN == '1';

```

### Library pseudocode for aarch64/functions/gcs/GCSSS1

```

// GCSSS1()
// ======
// Operational pseudocode for GCSSS1 instruction.

GCSSS1(bits(64) incoming_pointer)
    bits(64) outgoing_pointer, cmpoperand, operand, data;
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    AccessDescriptor accdesc = CreateAccDescGCSS1(PSTATE.EL);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid cap entry is expected
    cmpoperand = incoming_pointer[63:12]:'000000000001';
    // In-progress cap entry should be stored if the comparison is success
    operand      = outgoing_pointer[63:3]:'101';

    data = MemAtomic(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer[63:3]:'000');
    else
        GCSDataCheckException(GCSInstType_SS1);
    return;

```

## Library pseudocode for aarch64/functions/gcs/GCSSS2

```

// GCSSS2()
// ======
// Operational pseudocode for GCSSS2 instruction.

bits(64) GCSSS2()
    bits(64) outgoing_pointer, incoming_pointer, outgoing_value;
    AccessDescriptor accdesc_ld = CreateAccDescGCS(PSTATE.EL, MemOp LOA);
    AccessDescriptor accdesc_st = CreateAccDescGCS(PSTATE.EL, MemOp STO);
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem[incoming_pointer, 8, accdesc_ld];

    if outgoing_value[2:0] == '101' then //in_progress token
        outgoing_pointer[63:3] = outgoing_value[63:3] - 1;
        outgoing_pointer[2:0] = '000';
        outgoing_value = outgoing_pointer[63:12]:'000000000001';
        Mem[outgoing_pointer, 8, accdesc_st] = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType_SS2);
    return outgoing_pointer;

```

## Library pseudocode for aarch64/functions/gcs/GCSSTRTrapException

```

// GCSSTRTrapException()
// ======
// Handle a trap on GCSSTR instruction condition.

GCSSTRTrapException(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_GCSFail);

```

```
except.syndrome<24> = Zeros();
except.syndrome<23:20> = '0010';
except.syndrome<19:15> = Zeros();
except.syndrome<14:10> = ThisInstr()<9:5>;
except.syndrome<9:5> = ThisInstr()<4:0>;
except.syndrome<4:0> = Zeros();
AArch64.TakeException(target_el, except, preferred_exception_return);
```

### Library pseudocode for aarch64/functions/gcs/GCSSynchronizationBarrier

```
// GCSSynchronizationBarrier()
// =====
// Barrier instruction that synchronizes Guarded Control Stack
// accesses in relation to other load and store accesses

GCSSynchronizationBarrier();
```

### Library pseudocode for aarch64/functions/gcs/GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// =====

boolean GetCurrentEXLOCKEN()
    if Halted() || Restarting() then
        return FALSE;

    case PSTATE.EL of
        when EL0
            Unreachable();
        when EL1
            return GCSCR_EL1.EXLOCKEN == '1';
        when EL2
            return GCSCR_EL2.EXLOCKEN == '1';
        when EL3
            return GCSCR_EL3.EXLOCKEN == '1';
```

### Library pseudocode for aarch64/functions/gcs/GetCurrentGCSPointer

```

// GetCurrentGCSPointer()
// =====
// Returns the value of the current Guarded control stack
// pointer register.

bits(64) GetCurrentGCSPointer()
    bits(64) ptr;

    case PSTATE.EL of
        when EL0
            ptr = GCSPR_EL0.PTR:'000';
        when EL1
            ptr = GCSPR_EL1.PTR:'000';
        when EL2
            ptr = GCSPR_EL2.PTR:'000';
        when EL3
            ptr = GCSPR_EL3.PTR:'000';
    return ptr;

```

### Library pseudocode for aarch64/functions/gcs/LoadCheckGCSRecord

```

// LoadCheckGCSRecord()
// =====
// Validates the provided address against the top entry of the
// current Guarded control stack.

bits(64) LoadCheckGCSRecord(bits(64) vaddress, GCSIInstruction gcsinst_t)
    bits(64) ptr;
    bits(64) recorded_va;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);

    ptr = GetCurrentGCSPointer();
    recorded_va = Mem[ptr, 8, accdesc];
    if GCSRReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress)
        GCSDataCheckException(gcsinst_type);

    return recorded_va;

```

### Library pseudocode for aarch64/functions/gcs/SetCurrentGCSPointer

```

// SetCurrentGCSPointer()
// =====
// Writes a value to the current Guarded control stack pointer register

SetCurrentGCSPointer(bits(64) ptr)
    case PSTATE.EL of
        when EL0
            GCSPR_EL0.PTR = ptr<63:3>;
        when EL1
            GCSPR_EL1.PTR = ptr<63:3>;
        when EL2
            GCSPR_EL2.PTR = ptr<63:3>;
        when EL3
            GCSPR_EL3.PTR = ptr<63:3>;
    return;

```

## Library pseudocode for aarch64/functions/ic/AArch64.IC

```
// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
CacheRecord cache;

cache.acctype    = AccessType_IC;
cache.cachetype = CacheType_Instruction;
cache.cacheop   = CacheOp_Invalidate;
cache.opscope   = opscope;

if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
    ss = SecurityStateAtEL(PSTATE.EL);
    cache.cpas = CPASAtSecurityState(ss);
    if (opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_EL2
        && EL2Enabled() && HCR_EL2.FB == '1')) then
        cache.shareability = Shareability_ISH;
    else
        cache.shareability = Shareability_NSH;
    cache.regval = regval;
    CACHE_OP(cache);
else
    assert opscope == CacheOpScope_PoU;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];;
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];;
    else
        cache.is_asid_valid = FALSE;

    bits(64) vaddress = regval;
    boolean need_translate = ICInstNeedsTranslation(opscope);

    cache.vaddress      = regval;
    cache.shareability = Shareability_NSH;
    cache.translated    = need_translate;

    if !need_translate then
        cache.paddress = FullAddress UNKNOWN;
```

```

    CACHE_OP(cache);
    return;

AccessDescriptor accdesc = CreateAccDescIC(cache);
boolean aligned = TRUE;
integer size = 0;
AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddr);

if IsFault(memaddrdesc) then
    AArch64.Abort(regval, memaddrdesc.fault);

cache.cpas      = CPASATPAS(memaddrdesc.paddress.paspace);
cache.paddress = memaddrdesc.paddress;
CACHE_OP(cache);
return;

```

### **Library pseudocode for aarch64/functions/immediateop/ImmediateOp**

```

// ImmediateOp
// =====
// Vector logical immediate instruction types.

enumeration ImmediateOp { ImmediateOp_MOVI, ImmediateOp_MVNI,
                           ImmediateOp_ORR, ImmediateOp_BIC};

```

### **Library pseudocode for aarch64/functions/logicalop/LogicalOp**

```

// LogicalOp
// =====
// Logical instruction types.

enumeration LogicalOp { LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

```

### **Library pseudocode for aarch64/functions/mec/AArch64.S1AMECFault**

```

// AArch64.S1AMECFault()
// =====
// Returns TRUE if a Translation fault should occur for Realm EL2 and R
// stage 1 translated addresses to Realm PA space.

boolean AArch64.S1AMECFault(S1TTWParams walkparams, PASpace paspace, Re
                           bits(N) descriptor)
    assert N IN {64,128};
    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103

    return (walkparams.<emec, amec> == '10' &&
            regime IN {Regime EL2, Regime EL20} &&
            paspace == PAS Realm &&
            descriptor_amec == '1');

```

### **Library pseudocode for aarch64/functions/mec/ AArch64.S1DisabledOutputMECID**

```

// AArch64.S1DisabledOutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is disabled

bits(16) AArch64.S1DisabledOutputMECID(S1TTWParams walkparams, Regime regime)
    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if !(regime IN {Regime_EL2, Regime_EL20, Regime_EL10}) then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    if regime == Regime_EL10 then
        return VMECID_P_EL2.MECID;
    else
        return MECID_P0_EL2.MECID;

```

## Library pseudocode for aarch64/functions/mec/AArch64.S1OutputMECID

```

// AArch64.S1OutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is enabled

bits(16) AArch64.S1OutputMECID(S1TTWParams walkparams, Regime regime, PASpace paspace, bits(N) descriptor)
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103>;
    case regime of
        when Regime_EL3
            return MECID_RL_A_EL3.MECID;
        when Regime_EL2
            if descriptor_amec == '0' then
                return MECID_P0_EL2.MECID;
            else
                return MECID_A0_EL2.MECID;
        when Regime_EL20
            if varange == VARange_LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2.MECID;
                else
                    return MECID_A0_EL2.MECID;
            else
                if descriptor_amec == '0' then
                    return MECID_P1_EL2.MECID;
                else
                    return MECID_A1_EL2.MECID;
        when Regime_EL10
            return VMECID_P_EL2.MECID;

```

## Library pseudocode for aarch64/functions/mec/AArch64.S2OutputMECID

```
// AArch64.S2OutputMECID()
// =====
// Returns the output MECID for stage 2 address translation.

bits(16) AArch64.S2OutputMECID(S2TTWParams walkparams, PASpace paspace,
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT\_MECID;

    if paspace != PAS\_Realm then
        return DEFAULT\_MECID;

    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103>;
    if descriptor_amec == '0' then
        return VMECID_P_EL2.MECID;
    else
        return VMECID_A_EL2.MECID;
```

## Library pseudocode for aarch64/functions/mec/AArch64.TTWalkMECID

```
// AArch64.TTWalkMECID()
// =====
// Returns the associated MECID for the translation table walk of the current
// translation regime and Security state.

bits(16) AArch64.TTWalkMECID(bit emec, Regime regime, SecurityState ss)
    if emec == '0' then
        return DEFAULT\_MECID;

    if ss != SS\_Realm then
        return DEFAULT\_MECID;

    case regime of
        when Regime\_EL2
            return MECID_P0_EL2.MECID;
        when Regime\_EL20
            if TCR_EL2.A1 == '0' then
                return MECID_P1_EL2.MECID;
            else
                return MECID_P0_EL2.MECID;
        // This applies to stage 1 and stage 2 translation table walks
        // Realm EL1&0, but the stage 2 translation for a stage 1 walk
        // might later override the MECID according to AMEC configuration
        when Regime\_EL10
            return VMECID_P_EL2.MECID;
        otherwise
            Unreachable();
```

## Library pseudocode for aarch64/functions/mec/DEFAULT\_MECID

```
constant bits(16) DEFAULT_MECID = Zeros(16);
```

## Library pseudocode for aarch64/functions/memory/ AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccessDescriptor accdesc)
    assert accdesc.tagchecked;

    if UsingAArch32\(\) then
        return FALSE;

    boolean is_instr = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
        EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
        (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')) then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(accdesc.el) then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if (IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && !accdesc.write &&
        StoreOnlyTagCheckingEnabled(accdesc.el)) then
        return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/memory/ AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// =====
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

## Library pseudocode for aarch64/functions/memory/ AArch64.AllocationTagCheck

```

// AArch64.AllocationTagCheck()
// =====
// Performs an Allocation Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.AllocationTagCheck(AddressDescriptor memaddrdesc, AccessDescriptor accdesc)
{
    if memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

        return ptag == readtag;
    else
        return TRUE;
}

```

### **Library pseudocode for aarch64/functions/memory/ AArch64.AllocationTagFromAddress**

```

// AArch64.AllocationTagFromAddress()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical
// bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
//           return tagged_address<59:56>;

```

### **Library pseudocode for aarch64/functions/memory/ AArch64.CanonicalTagCheck**

```

// AArch64.CanonicalTagCheck()
// =====
// Performs a Canonical Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.CanonicalTagCheck(AddressDescriptor memaddrdesc, bits(4) expected_tag)
{
    if memaddrdesc.vaddress<55> == '0' then '0000' else
        return ptag == expected_tag;
}

```

### **Library pseudocode for aarch64/functions/memory/AArch64.CheckTag**

```

// AArch64.CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc)
{
    if memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        return AArch64.AllocationTagCheck(memaddrdesc, accdesc, ptag);
    elseif memaddrdesc.memattrs.tags == MemTag_CanonicallyTagged then
        return AArch64.CanonicalTagCheck(memaddrdesc, ptag);
    else
        return TRUE;
}

```

## Library pseudocode for aarch64/functions/memory/ AArch64.IsUnprivAccessPriv

```
// AArch64.IsUnprivAccessPriv()
// =====
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

boolean AArch64.IsUnprivAccessPriv()
    boolean privileged;

    case PSTATE.EL of
        when EL0
            privileged = FALSE;
        when EL1
            privileged = EL2Enabled() && IsFeatureImplemented(FEAT_NV)
        when EL2
            privileged = !(IsFeatureImplemented(FEAT_VHE) && HCR_EL2.<EL2_PRIVILEGED>)
        when EL3
            privileged = TRUE;

    if IsFeatureImplemented(FEAT_UAO) && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;

    return privileged;
```

## Library pseudocode for aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc,
                                boolean aligned]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    AccessDescriptor accdesc = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        assert AllInAlignedQuantity(address, size, 16);
    else
        assert IsAligned(address, size);

    // If the instruction encoding permits tag checking, confer with system
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if IsFeatureImplemented(FEAT_TME) then
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc)
            FailTransaction(TMFailure_IMP, FALSE);
```

```

if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
        AArch64.TagCheckFault(address, accdesc);

if SPESampleInFlight then
    boolean is_load = TRUE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

boolean atomic;
if (memaddrdesc.memattrs.memtype == MemType_Normal &&
    memaddrdesc.memattrs.inner.attrs == MemAttr_WB &&
    memaddrdesc.memattrs.outer.attrs == MemAttr_WB) then
    atomic = TRUE;
elseif (accdesc.exclusive || accdesc.atomicop ||
        accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
    if !aligned && !ConstrainUnpredictableBool(Unpredictable_MISALI
        AArch64.Abort(address, AlignmentFault(accdesc));
    else
        atomic = TRUE;
elseif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within 16 byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is a

PhysMemRetStatus memstatus;
if atomic then
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
elseif aligned && accdesc.ispair then
    assert size IN {8, 16};
    constant integer halfsize = size DIV 2;
    bits(halfsize * 8) lowhalf, highhalf;
    (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

    value = highhalf:lowhalf;
else
    for i = 0 to size-1
        (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return value;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc,
                boolean aligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};

```

```

AccessDescriptor accdesc = accdesc_in;
if IsFeatureImplemented(FEAT_LSE2) then
    assert AllInAlignedQuantity(address, size, 16);
else
    assert IsAligned(address, size);

// If the instruction encoding permits tag checking, confer with sys
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

AddressDescriptor memaddrdesc;
memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

if IsFeatureImplemented(FEAT_TME) then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc)
        FailTransaction(TMFailure_IMP, FALSE);

if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
        AArch64.TagCheckFault(address, accdesc);

if SPESampleInFlight then
    boolean is_load = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

PhysMemRetStatus memstatus;
boolean atomic;
if (memaddrdesc.memattrs.memtype == MemType_Normal &&
    memaddrdesc.memattrs.inner.attrs == MemAttr_WB &&
    memaddrdesc.memattrs.outer.attrs == MemAttr_WB) then
    atomic = TRUE;
elseif (accdesc.exclusive || accdesc.atomicop ||
        accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
    if !aligned && !ConstrainUnpredictableBool(Unpredictable_MISALIGNED)
        AArch64.Abort(address, AlignmentFault(accdesc));
    else
        atomic = TRUE;
elseif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within 16 byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

if atomic then
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
elseif aligned && accdesc.ispair then
    assert size IN {8, 16};

```

```

constant integer halfsize = size DIV 2;
bits(halfsize*8) lowhalf, highhalf;
<highhalf, lowhalf> = value;

memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhal
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize,
memaddrdesc.paddress.address = memaddrdesc.paddress.address + h
memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highha
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize,
else
    for i = 0 to size-1
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i>
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, acc
            memaddrdesc.paddress.address = memaddrdesc.paddress.address
return;

```

## Library pseudocode for aarch64/functions/memory/AArch64.MemTag

```

// AArch64.MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in]
assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

AddressDescriptor memaddrdesc;
AccessDescriptor accdesc = accdesc_in;

boolean aligned = TRUE;

if IsFeatureImplemented(FEAT_MTE2) then
    accdesc.tagaccess = AArch64.AllocationTagAccess.IsEnabled(accdes

memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, T

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag\_Allocated
    (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return tag;
elseif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
        accdesc.tagaccess &&
        memaddrdesc.memattrs.tags == MemTag\_CanonicallyTagged) the
        return if address<55> == '0' then '0000' else '1111';
else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====

```

```

// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in] = bits(4)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

AddressDescriptor memaddrdesc;
AccessDescriptor accdesc = accdesc_in;

boolean aligned = IsAligned(address, TAG_GRANULE);

// Stores of allocation tags must be aligned
if !aligned then
    AArch64.Abort(address, AlignmentFault(accdesc));

if IsFeatureImplemented(FEAT_MTE2) then
    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc);

memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG_GRANULE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag_Allocated
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

```

## Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```

// AArch64.PhysicalTag()
// =====
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;

```

## Library pseudocode for aarch64/functions/memory/ AArch64.UnalignedAccessFaults

```

// AArch64.UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

boolean AArch64.UnalignedAccessFaults(AccessDescriptor accdesc, bits(64) address)
    if AlignmentEnforced() then
        return TRUE;
    elseif accdesc.acctype == AccessType_GCS then
        return TRUE;
    elseif accdesc.rcw then
        return TRUE;
    elseif accdesc.ls64 then
        return TRUE;
    elseif accdesc.exclusive || accdesc.atomicop then
        return !IsFeatureImplemented(FEAT_LSE2) || !AllInAlignedQuantities;
    elseif accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        return TRUE;

```

```

        return (!IsFeatureImplemented(FEAT_LSE2) ||
                (SCTLR\_ELx[]).nAA == '0' && !AllInAlignedQuantity(address));
    else
        return FALSE;
}

```

## Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```

// AddressSupportsLS64()
// =====
// Returns TRUE if the 64-byte block following the given address supports
// LD64B and ST64B instructions, and FALSE otherwise.

boolean AddressSupportsLS64(bits(56) paddress);

```

## Library pseudocode for aarch64/functions/memory/AllInAlignedQuantity

```

// AllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity,
// otherwise FALSE.

boolean AllInAlignedQuantity(bits(64) address, integer size, integer alignment)
{
    assert(size <= alignment);
    return Align((address+size)-1, alignment) == Align(address, alignment);
}

```

## Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```

// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
{
    bits(64) sp = SP[];;
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR\_ELx[]).SA0 != '0';
    else
        stack_align_check = (SCTLR\_ELx[]).SA != '0';

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAignmentFault();
}

return;

```

## Library pseudocode for aarch64/functions/memory/Mem

```

// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for
// instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccessDescriptor accde];
assert size IN {1, 2, 4, 8, 16};
constant halfsize = size DIV 2;

```

```

bits(size * 8) value;
bits(halfsize * 8) lowhalf, highhalf;
AccessDescriptor accdesc = accdesc_in;

// Check alignment on size of element accessed, not overall access
integer alignment = if accdesc.ispair then halfsize else size;
boolean aligned = IsAligned(address, alignment);

if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size)
    AArch64.Abort(address, AlignmentFault(accdesc));

if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, size)
    // If 128-bit SIMD&FP ordered access are treated as a pair of
    // 64-bit single-copy atomic accesses, then these single copy atomic
    // access can be observed in any order.
    lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
    highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
    value = highhalf:lowhalf;
elseif IsFeatureImplemented(FEAT_LSE2) && AllInAlignedQuantity(address, size)
    value = AArch64.MemSingle[address, size, accdesc, aligned];
elseif accdesc.ispair && aligned then
    accdesc.ispair = FALSE;
    if IsFeatureImplemented(FEAT_LRPCC3) && accdesc.highestaddresssize == 16
        highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
    else
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
    value = highhalf:lowhalf;
elseif aligned then
    value = AArch64.MemSingle[address, size, accdesc, aligned];
else
    assert size > 1;
    if IsFeatureImplemented(FEAT_LRPCC3) && accdesc.ispair && accdesc.highestaddresssize == 16
        // Performing memory accesses from one load or store instruction
        // crosses a boundary corresponding to the smallest translation unit
        // implementation causes CONSTRAINED UNPREDICTABLE behavior
        for i = 0 to halfsize-1
            highhalf<8*i+7:8*i> = AArch64.MemSingle[address+halfsize+i, 1, accdesc, aligned];
        for i = 0 to halfsize-1
            lowhalf<8*i+7:8*i> = AArch64.MemSingle[address + i, 1, accdesc, aligned];
        value = highhalf:lowhalf;
    else
        value<7:0> = AArch64.MemSingle[address, 1, accdesc, aligned];
        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE when
        // memory access will generate an Alignment Fault, as to get
        // byte did not, so we must be changing to a new translation
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;
        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, accdesc, aligned];

```

```

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a bi

Mem[bits(64) address, integer size, AccessDescriptor accdesc_in] = bits
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
AccessDescriptor accdesc = accdesc_in;

// Check alignment on size of element accessed, not overall access
integer alignment = if accdesc.ispair then halfsize else size;
boolean aligned = IsAligned(address, alignment);

if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size)
    AArch64.Abort(address, AlignmentFault(accdesc));

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse(value);

if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(a
    // 128-bit SIMD&FP stores are treated as a pair of 64-bit singl
    // 64-bit aligned.
    <highhalf, lowhalf> = value;
    AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhal
    AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned]
elseif IsFeatureImplemented(FEAT_LSE2) && AllInAlignedQuantity(addre
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
elseif accdesc.ispair && aligned then
    accdesc.ispair = FALSE;
    <highhalf, lowhalf> = value;
    if IsFeatureImplemented(FEAT_LRPCC3) && accdesc.highestaddressf
        AArch64.MemSingle[address+halfsize, halfsize, accdesc, alig
        AArch64.MemSingle[address, halfsize, accdesc, align
    else
        AArch64.MemSingle[address, halfsize, accdesc, align
        AArch64.MemSingle[address+halfsize, halfsize, accdesc, align
elseif aligned then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
else
    assert size > 1;
    if IsFeatureImplemented(FEAT_LRPCC3) && accdesc.ispair && accde
        // Performing memory accesses from one load or store instru
        // crosses a boundary corresponding to the smallest transla
        // implementation causes CONSTRAINED UNPREDICTABLE behavior
        <highhalf, lowhalf> = value;
        for i = 0 to halfsize-1
            // Individual byte access can be observed in any order
            AArch64.MemSingle[address+halfsize+i, 1, accdesc, align
        for i = 0 to halfsize-1
            // Individual byte access can be observed in any order,
            // of highhalf
            AArch64.MemSingle[address+i, 1, accdesc, aligned] = low
    else
        AArch64.MemSingle[address, 1, accdesc, aligned] = value<7:0>

```

```

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE when
    // memory access will generate an Alignment Fault, as to get
    // byte did not, so we must be changing to a new translation

    c = ConstrainUnpredictable\(Unpredictable\_DEVPAGE2\);
    assert c IN {Constraint\_FAULT, Constraint\_NONE};
    if c == Constraint\_NONE then aligned = TRUE;

    for i = 1 to size-1
        AArch64.MemSingle[address+i, 1, accdesc, aligned] = val
return;

```

## Library pseudocode for aarch64/functions/memory/MemAtomic

```

// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address

bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand, bits(size)
                     AccessDescriptor accdesc_in)
assert accdesc_in.atomicop;

constant integer bytes = size DIV 8;
assert bytes IN {1, 2, 4, 8, 16};

bits(size) newvalue;
bits(size) oldvalue;
AccessDescriptor accdesc = accdesc_in;
boolean aligned = IsAligned(address, bytes);

// If the instruction encoding permits tag checking, confer with sys
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes)
    AArch64.Abort(address, AlignmentFault(accdesc));

// MMU or MPU lookup
AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, a
memaddrdesc);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID());
    si

// For Store-only Tag checking, the tag check is performed on the s
if (IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked &&
    (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
     !StoreOnlyTagCheckingEnabled(accdesc.el))) then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
        accdesc.write = FALSE;           // Tag Check Fault on a read
        AArch64.TagCheckFault(address, accdesc);

```

```

// All observers in the shareability domain observe the following 1
PhysMemRetStatus memstatus;
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);

if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc)
if BigEndian(accdesc.acctype) then
    oldvalue = BigEndianReverse(oldvalue);

boolean cmpfail = FALSE;
case accdesc.modop of
    when MemAtomicOp_ADD
    when MemAtomicOp_BIC
    when MemAtomicOp_EOR
    when MemAtomicOp_ORR
    when MemAtomicOp_SMAX
    when MemAtomicOp_SMIN
    when MemAtomicOp_UMAX
    when MemAtomicOp_UMIN
    when MemAtomicOp_SWP
    when MemAtomicOp_CAS
    when MemAtomicOp_GCSSS1 newvalue = oldvalue + operand;
    newvalue = oldvalue AND NOT(operand);
    newvalue = oldvalue EOR operand;
    newvalue = oldvalue OR operand;
    newvalue = Max(SInt(oldvalue), SInt(operan
    newvalue = Min(SInt(oldvalue), SInt(operan
    newvalue = Max(UINT(oldvalue), UINT(operan
    newvalue = Min(UINT(oldvalue), UINT(operan
    newvalue = operand;
    newvalue = operand; cmpfail = cmpoperan
    newvalue = operand; cmpfail = cmpoperan

if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckin
    // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE
    // Tag check is performed.
    if accdesc.tagchecked && cmpfail then
        accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictab

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

if !cmpfail then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue)
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc)

if SPESampleInFlight then
    boolean is_load = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

// Load operations return the old (pre-operation) value
return oldvalue;

```

## Library pseudocode for aarch64/functions/memory/MemAtomicRCW

```

// MemAtomicRCW()
// =====
// Perform a single-copy-atomic access with Read-Check-Write operation

(bits(4), bits(size)) MemAtomicRCW(bits(64) address, bits(size) cmpoperan
                                         AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;

```

```

assert accdesc_in.rcw;

constant integer bytes = size DIV 8;
assert bytes IN {8, 16};

bits(4) nzcv;
bits(size) oldvalue;
bits(size) newvalue;
AccessDescriptor accdesc = accdesc_in;
boolean aligned = IsAligned(address, bytes);

// If the instruction encoding permits tag checking, confer with sys
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes)
    AArch64.Abort(address, AlignmentFault(accdesc));

// MMU or MPU lookup
AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, a);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), si);

// For Store-only Tag checking, the tag check is performed on the store
if (IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked &&
    (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
     !StoreOnlyTagCheckingEnabled(accdesc.el))) then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
        accdesc.write = FALSE;           // Tag Check Fault on a read
        AArch64.TagCheckFault(address, accdesc);

// All observers in the shareability domain observe the following 1-bit
// PhysMemRetStatus memstatus;
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);

if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);

if BigEndian(accdesc.acctype) then
    oldvalue = BigEndianReverse(oldvalue);

boolean cmpfail = FALSE;
case accdesc.modop of
    when MemAtomicOp_BIC newvalue = oldvalue AND NOT(operand);
    when MemAtomicOp_ORR newvalue = oldvalue OR operand;
    when MemAtomicOp_SWP newvalue = operand;
    when MemAtomicOp_CAS newvalue = operand; cmpfail = oldvalue != newvalue;

if cmpfail then
    nzcv = '1010'; // N = 1 indicates compare failure
else
    nzcv = RCWCheck(oldvalue, newvalue, accdesc.rcws);

```

```

if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckin
    // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE
    // Tag check is performed.
    if accdesc.tagchecked && cmpfail then
        accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    if nzcv == '0010' then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse(newvalue);

        memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);

        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

    return (nzcv, oldvalue);

```

## Library pseudocode for aarch64/functions/memory/MemLoad64B

```

// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccessDescriptor accdesc_in)
{
    bits(512) data;
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, a);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.mematrrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);
}

```

```

if !AddressSupportsLS64(memaddrdesc.paddress.address) then
    c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
    assert c IN {Constraint LIMITED ATOMICITY, Constraint FAULT};

    if c == Constraint FAULT then
        // Generate a stage 1 Data Abort reported using the DFSC co
        AArch64.Abort(address, ExclusiveFault(accdesc));
    else
        // Accesses are not single-copy atomic above the byte level
        for i = 0 to size-1
            PhysMemRetStatus memstatus;
            (memstatus, data<8*i+7:8*i>) = PhysMemRead(memaddrdesc,
                if IsFault(memstatus) then
                    HandleExternalReadAbort(memstatus, memaddrdesc, 1,

            memaddrdesc.paddress.address = memaddrdesc.paddress.adc
        else
            PhysMemRetStatus memstatus;
            (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, size, accde

    return data;

```

## Library pseudocode for aarch64/functions/memory/MemStore64B

```

// MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function
// does not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccessDescriptor accdesc)
{
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size)
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with sys
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdes

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, a

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

```

```

PhysMemRetStatus memstatus;
if !AddressSupportsLS64(memaddrdesc.paddress.address) then
    c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
    assert c IN {Constraint LIMITED ATOMICITY, Constraint FAULT};

    if c == Constraint_FAULT then
        // Generate a Data Abort reported using the DFSC code of 11
        AArch64.Abort(address, ExclusiveFault(accdesc));
    else
        // Accesses are not single-copy atomic above the byte level
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address;
else
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

return;

```

## Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```

// MemStore64BWithRet()
// =====
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccessDescriptor accdesc_in,
                           constant integer size = 64;
                           AccessDescriptor accdesc = accdesc_in;
                           boolean aligned = IsAligned(address, size);

if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size)
    AArch64.Abort(address, AlignmentFault(accdesc));

// If the instruction encoding permits tag checking, confer with system
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, a);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
    return ZeroExtend('1', 64);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    bits(4) ptag = AArch64.PhysicalTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then

```

```

        AArch64.TagCheckFault(address, accdesc);
        return ZeroExtend('1', 64);

PhysMemRetStatus memstatus;
memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc)

return memstatus.store64bstatus;

```

### **Library pseudocode for aarch64/functions/memory/ MemStore64BWithRetStatus**

```

// MemStore64BWithRetStatus()
// =====
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed
// or was not supported at this memory location.

bits(64) MemStore64BWithRetStatus();

```

### **Library pseudocode for aarch64/functions/memory/NVMem**

```

// NVMem[] - non-assignment form
// =====
// This function is the load memory access for the transformed System memory
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System
//     supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
assert offset > 0;
constant integer size = 64;
return NVMem[offset, size];

bits(N) NVMem[integer offset, integer N]
assert offset > 0;
assert N IN {64,128};
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
AccessDescriptor accdesc = CreateAccDescNV2(MemOp LOAD);
return Mem[address, N DIV 8, accdesc];

// NVMem[] - assignment form
// =====
// This function is the store memory access for the transformed System memory
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System
//     supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
assert offset > 0;

```

```

constant integer size = 64;
NVMem[offset, size] = value;
return;

NVMem[integer offset, integer N] = bits(N) value
    assert offset > 0;
    assert N IN {64,128};
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
AccessDescriptor accdesc = CreateAccDescNV2(MemOp_STORE);
Mem[address, N DIV 8, accdesc] = value;
return;

```

## Library pseudocode for aarch64/functions/memory/PhysMemTagRead

```

// PhysMemTagRead()
// =====
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the
//   memory
//
// The accdesc descriptor describes the access type: normal, exclusive,
// etc and other parameters required to access the physical memory or
// register in the event of an External abort.

(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc)

```

## Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

```

// PhysMemTagWrite()
// =====
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the
//   memory
//
// The accdesc descriptor describes the access type: normal, exclusive,
// etc and other parameters required to access the physical memory or
// register in the event of an External abort.

PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc)

```

## Library pseudocode for aarch64/functions/memory/ StoreOnlyTagCheckingEnabled

```

// StoreOnlyTagCheckingEnabled()
// =====
// Returns TRUE if loads executed at the given Exception level are Tag
// checked.

boolean StoreOnlyTagCheckingEnabled(bits(2) el)
    assert IsFeatureImplemented(FEAT_MTE_STORE_ONLY);

```

```

    bit tcso;

    case el of
        when EL0
            if !ELIsInHost(el) then
                tcso = SCTRLR_EL1.TCSO0;
            else
                tcso = SCTRLR_EL2.TCSO0;
        when EL1
            tcso = SCTRLR_EL1.TCSO;
        when EL2
            tcso = SCTRLR_EL2.TCSO;
        otherwise
            tcso = SCTRLR_EL3.TCSO;

    return tcso == '1';

```

### **Library pseudocode for aarch64/functions/mops/CPYFOptionA**

```

// CPYFOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPYF* instructions, and FALSE otherwise.

boolean CPYFOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPYF* instructions use Option

```

### **Library pseudocode for aarch64/functions/mops/CPYOptionA**

```

// CPYOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

boolean CPYOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY* instructions use Option

```

### **Library pseudocode for aarch64/functions/mops/CPYPostSizeChoice**

```

// CPYPostSizeChoice()
// =====
// Returns the size of the copy that is performed by the CPYE* instruction
// implementation given the parameters of the destination, source and size.
// Postsize is encoded as -1*size for an option A implementation if cpysize
// is negative.

bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64)

```

### **Library pseudocode for aarch64/functions/mops/CPYPreSizeChoice**

```

// CPYPreSizeChoice()
// =====
// Returns the size of the copy that is performed by the CPYP* instruction
// implementation given the parameters of the destination, source and size.

```

```
// Presize is encoded as -1*size for an option A implementation if cpysize is
bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) presize)
```

### Library pseudocode for aarch64/functions/mops/CPYSIZEChoice

```
// CPYSIZEChoice()
// =====
// Returns the size of the block this performed for an iteration of the
// parameters of the destination, source and size of the copy.

integer CPYSIZEChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) presize)
```

### Library pseudocode for aarch64/functions/mops/CheckMOPSEnabled

```
// CheckMOPSEnabled()
// =====
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

CheckMOPSEnabled()
if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
    (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
    UNDEFINED;

if (PSTATE.EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
    (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
    UNDEFINED;

if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
    UNDEFINED;
```

### Library pseudocode for aarch64/functions/mops/CheckMemcpyParams

```
// CheckMemcpyParams()
// =====
// Check if the parameters to a CPY* or CPYF* instruction are consistent
// with the PE state and well-formed.

CheckMemcpyParams(MOPSStage stage, boolean implements_option_a, bits(4)
                integer d, integer s, integer n, bits(64) toaddress,
                bits(64) cpysize)
boolean from_epilogue = stage == MOPSStage_Epilogue;
// Check if this version is consistent with the state of the call.
if MemcpyZeroSizeCheck() || SInt(cpysize) != 0 then
    boolean using_option_a = nzcv<1> == '0';
    if implements_option_a != using_option_a then
        boolean wrong_option = TRUE;
        MismatchedMemcpyException(implements_option_a, d, s, n, wrong_option,
                                    from_epilogue, options);

// Check if the parameters to this instruction are valid.
if stage == MOPSStage_Main then
    if MemcpyParametersIllformedM(toaddress, fromaddress, cpysize)
        boolean wrong_option = FALSE;
```

```

    MismatchedMemcpyException(implements_option_a, d, s, n, wrco,
                                from_epilogue, options);
else
    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, c
    if (cpysize != postsize || MemcpyParametersIllformedE(toaddress
        boolean wrong_option = FALSE;
    MismatchedMemcpyException(implements_option_a, d, s, n, wrco
                                from_epilogue, options);

```

## Library pseudocode for aarch64/functions/mops/CheckMemSetParams

```

// CheckMemSetParams()
// =====
// Check if the parameters to a SET* or SETG* instruction are consistent
// PE state and well-formed.

CheckMemSetParams(MOPSStage stage, boolean implements_option_a, bits(4)
                  integer d, integer s, integer n, bits(64) toaddress,
                  boolean is_setg)
boolean from_epilogue = stage == MOPSStage_Epilogue;

// Check if this version is consistent with the state of the call.
if MemcpyZeroSizeCheck() || SInt(setsize) != 0 then
    boolean using_option_a = nzcv<1> == '0';
    if implements_option_a != using_option_a then
        boolean wrong_option = TRUE;
    MismatchedMemSetException(implements_option_a, d, s, n, wrco
                                from_epilogue, options, is_setg);

// Check if the parameters to this instruction are valid.
if stage == MOPSStage_Main then
    if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
        boolean wrong_option = FALSE;
    MismatchedMemSetException(implements_option_a, d, s, n, wrco
                                from_epilogue, options, is_setg);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg)
    if (setsize != postsize || MemSetParametersIllformedE(toaddress
        boolean wrong_option = FALSE;
    MismatchedMemSetException(implements_option_a, d, s, n, wrco
                                from_epilogue, options, is_setg);

```

## Library pseudocode for aarch64/functions/mops/IsMemcpyForward

```

// IsMemcpyForward()
// =====
// Returns TRUE if in a memcpy of size cpysize bytes from the source address
// to destination address toaddress is done in the forward direction or
// vice versa.

boolean IsMemcpyForward(bits(64) toaddress, bits(64) fromaddress, bits(64)
                       boolean forward;

// Check for overlapping cases
if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) ||
    (UInt(fromaddress<55:0>) < UInt(ZeroExtend(toaddress<55:0>, 64))
     forward = TRUE;

```

```

        elseif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) &&
                (UInt(ZeroExtend(fromaddress<55:0>, 64) + cpysize) > UInt(
                    forward = FALSE;

                // Non-overlapping case
                else
                    forward = boolean IMPLEMENTATION_DEFINED "CPY in the forward di

                return forward;

```

### **Library pseudocode for aarch64/functions/mops/MOPSStage**

```

// MOPSStage
// =====
enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_E

```

### **Library pseudocode for aarch64/functions/mops/MaxBlockSizeCopiedBytes**

```

// MaxBlockSizeCopiedBytes()
// =====
// Returns the maximum number of bytes that can be used in a single block
integer MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a sing

```

### **Library pseudocode for aarch64/functions/mops/** **MemCpyParametersIllformedE**

```

// MemCpyParametersIllformedE()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their si
// for a CPYE* instruction for this implementation given the parameters
// source and size of the copy.

boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddr
                                bits(64) cpysize);

```

### **Library pseudocode for aarch64/functions/mops/** **MemCpyParametersIllformedM**

```

// MemCpyParametersIllformedM()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their si
// for a CPYM* instruction for this implementation given the parameters
// source and size of the copy.

boolean MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddr
                                bits(64) cpysize);

```

### **Library pseudocode for aarch64/functions/mops/MemCpyStageSize**

```

// MemCopyStageSize()
// =====
// Returns the number of bytes copied by the given stage of a CPY* or CSET* instruction.

bits(64) MemCopyStageSize(MOPSStage stage, bits(64) toaddress, bits(64)
                           bits(64) cpysize)
    bits(64) stagecpysize;
    if stage == MOPSStage_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
        assert stagecpysize<63> == cpysize<63> || IsZero(stagecpysize);

        if SInt(cpysize) > 0 then
            assert SInt(stagecpysize) <= SInt(cpysize);
        else
            assert SInt(stagecpysize) >= SInt(cpysize);
    else
        bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
        assert postsize<63> == cpysize<63> || IsZero(postsize);

        if stage == MOPSStage_Main then
            stagecpysize = cpysize - postsize;
        else
            stagecpysize = postsize;

    return stagecpysize;

```

### **Library pseudocode for aarch64/functions/mops/MemCopyZeroSizeCheck**

```

// MemCopyZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a copy of size zero.

boolean MemCopyZeroSizeCheck();

```

### **Library pseudocode for aarch64/functions/mops/ MemSetParametersIllformedE**

```

// MemSetParametersIllformedE()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their size or alignment)
// for a SETE* or SETGE* instruction for this implementation.
// parameters of the destination and size of the set.

boolean MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsized);

```

### **Library pseudocode for aarch64/functions/mops/ MemSetParametersIllformedM**

```

// MemSetParametersIllformedM()
// =====
// Returns TRUE if the inputs are not well formed (in terms of their si
// alignment) for a SETM* or SETGM* instruction for this implementation
// parameters of the destination and size of the copy.

boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsiz

```

### Library pseudocode for aarch64/functions/mops/MemSetStageSize

```

// MemSetStageSize()
// =====
// Returns the number of bytes set by the given stage of a SET* or SETG

bits(64) MemSetStageSize(MOPSStage stage, bits(64) toaddress, bits(64)
    bits(64) stagesetsize;
    if stage == MOPSStage_Prologue then
        stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
        assert stagesetsize<63> == setsize<63> || IsZero(stagesetsize);
        if is_setg then assert stagesetsize<3:0> == '0000';

        if SInt(setsize) > 0 then
            assert SInt(stagesetsize) <= SInt(setsize);
        else
            assert SInt(stagesetsize) >= SInt(setsize);
    else
        bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_se
        assert postsize<63> == setsize<63> || IsZero(postsize);
        if is_setg then assert postsize<3:0> == '0000';

        if stage == MOPSStage_Main then
            stagesetsize = setsize - postsize;
        else
            stagesetsize = postsize;

    return stagesetsize;

```

### Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```

// MemSetZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a copy of si

boolean MemSetZeroSizeCheck();

```

### Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

```

// MismatchedCpySetTargetEL()
// =====
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;

    if UInt(PSTATE.EL) > UInt(EL1) then

```

```

        target_el = PSTATE.EL;
    elseif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elseif (PSTATE.EL == EL1 && EL2Enabled() &&
IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;

```

### **Library pseudocode for aarch64/functions/mops/ MismatchedMemcpyException**

```

// MismatchedMemcpyException()
// =====
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemcpyException(boolean option_a, integer destreg, integer sr
                           boolean wrong_option, boolean from_epilogue,
                           bits(64) preferred_exception_return = ThisInstrAddr(64);
                           integer vect_offset = 0x0;
                           bits(2) target_el = MismatchedCopySetTargetEL());

    ExceptionRecord except = ExceptionSyndrome(Exception_MemCopyMemSet);
    except.syndrome<24> = '0';
    except.syndrome<23> = '0';
    except.syndrome<22:19> = options;
    except.syndrome<18> = if from_epilogue then '1' else '0';
    except.syndrome<17> = if wrong_option then '1' else '0';
    except.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    except.syndrome<14:10> = destreg<4:0>;
    except.syndrome<9:5> = srcreg<4:0>;
    except.syndrome<4:0> = sizereg<4:0>

    AArch64.TakeException(target_el, except, preferred_exception_return);

```

### **Library pseudocode for aarch64/functions/mops/ MismatchedMemSetException**

```

// MismatchedMemSetException()
// =====
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer da
                           boolean wrong_option, boolean from_epilogue,
                           boolean is_SETG)
                           bits(64) preferred_exception_return = ThisInstrAddr(64);
                           integer vect_offset = 0x0;
                           bits(2) target_el = MismatchedCopySetTargetEL());

    ExceptionRecord except = ExceptionSyndrome(Exception_MemCopyMemSet);
    except.syndrome<24> = '1';
    except.syndrome<23> = if is_SETG then '1' else '0';

```

```

// exception.syndrome<22:21> is RES0
except.syndrome<20:19> = options;
except.syndrome<18>    = if from_epilogue then '1' else '0';
except.syndrome<17>    = if wrong_option then '1' else '0';
except.syndrome<16>    = if option_a then '1' else '0';
// exception.syndrome<15> is RES0
except.syndrome<14:10> = destreg<4:0>;
except.syndrome<9:5>   = datareg<4:0>;
except.syndrome<4:0>   = sizereg<4:0>;

```

[AArch64.TakeException](#)(target\_el, except, preferred\_exception\_return)

### Library pseudocode for aarch64/functions/mops/SETGOptionA

```

// SETGOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

boolean SETGOptionA()
    return boolean IMPLEMENTATION_DEFINED "SETG* instructions use Option

```

### Library pseudocode for aarch64/functions/mops/SETOptionA

```

// SETOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

boolean SETOptionA()
    return boolean IMPLEMENTATION_DEFINED "SET* instructions use Option

```

### Library pseudocode for aarch64/functions/mops/SETPostSizeChoice

```

// SETPostSizeChoice()
// =====
// Returns the size of the set that is performed by the SETE* or SETGE*
// for this implementation, given the parameters of the destination and
// Postsize is encoded as -1*size for an option A implementation if set

bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsiz, boolean

```

### Library pseudocode for aarch64/functions/mops/SETPreSizeChoice

```

// SETPreSizeChoice()
// =====
// Returns the size of the set that is performed by the SETP* or SETGP*
// for this implementation, given the parameters of the destination and
// Presize is encoded as -1*size for an option A implementation if set

bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsiz, boolean

```

## Library pseudocode for aarch64/functions/mops/SETSizeChoice

```
// SETSizeChoice()  
// =====  
// Returns the size of the block this performed for an iteration of the  
// the parameters of the destination and size of the set. The size of t  
// is an integer multiple of alignsize.  
  
integer SETSizeChoice(bits(64) toaddress, bits(64) setsze, integer ali
```

## Library pseudocode for aarch64/functions/movewideop/MoveWideOp

```
// MoveWideOp  
// =====  
// Move wide 16-bit immediate instruction types.  
  
enumeration MoveWideOp {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

## Library pseudocode for aarch64/functions/movwpreferred/ MoveWidePreferred

```
// MoveWidePreferred()  
// =====  
//  
// Return TRUE if a bitmask immediate encoding would generate an immedi  
// value that could also be represented by a single MOVZ or MOVN instru  
// Used as a condition for the preferred MOV<-ORR alias.  
  
boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)  
    integer s = UInt(imms);  
    integer r = UInt(immr);  
    integer width = if sf == '1' then 64 else 32;  
  
    // element size must equal total immediate size  
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then  
        return FALSE;  
    if sf == '0' && !((immN:imms) IN {'00xxxxx'}) then  
        return FALSE;  
  
    // for MOVZ must contain no more than 16 ones  
    if s < 16 then  
        // ones must not span halfword boundary when rotated  
        return (-r MOD 16) <= (15 - s);  
  
    // for MOVN must contain no more than 16 zeros  
    if s >= width - 15 then  
        // zeros must not span halfword boundary when rotated  
        return (r MOD 16) <= (s - (width - 15));  
  
    return FALSE;
```

## Library pseudocode for aarch64/functions/pac/addpac/AddPAC

```

// AddPAC()
// ======
// Calculates the pointer authentication code for a 64-bit quantity and
// inserts that into pointer authentication code field of that 64-bit quantity

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
{
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean isgeneric = FALSE;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use both
    // the pointer to select between upper and lower ranges, and preserve
    // This handles the awkward case where there is apparently no correlation
    // the upper and lower address range - ie an addr of 1xxxxxxxxx0... with
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBIO == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBIO == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                // EL2 translation regime registers
                if data then
                    if TCR_EL2.TBI1 == '1' || TCR_EL2.TBIO == '1' then
                        selbit = ptr<55>;
                    else
                        selbit = ptr<63>;
                else
                    if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                        (TCR_EL2.TBIO == '1' && TCR_EL2.TBID0 == '0')) then
                        selbit = ptr<55>;
                    else
                        selbit = ptr<63>;
                else selbit = if tbi then ptr<55> else ptr<63>;
        if IsFeatureImplemented(FEAT_PAuth2) && IsFeatureImplemented(FEAT_C)
            selbit = ptr<55>;
        constant integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

        // If the VA is 56 or 55 bits and Top Byte is Ignored,
        // there are no unused bits left to insert the PAC
        if tbi && bottom_PAC_bit >= 55 then
            return ptr;
}

```

```

extfield = Replicate(selbit, 64);

// Compute the pointer authentication code for a ptr with good exte
if tbi then
    ext_ptr = (ptr<63:56> :
                extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit
elsif mtx then
    ext_ptr = (extfield<63:60> : ptr<59:56> :
                extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit
else
    ext_ptr = extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit

PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>, isgeneric);

// Check if the ptr has good extension bits and corrupt the pointer
bits(64) unusedbits_mask = Zeros(64);
unusedbits_mask<54:bottom_PAC_bit> = Ones((54-bottom_PAC_bit)+1);
if tbi then
    unusedbits_mask<63:56> = Ones(8);
elsif mtx then
    unusedbits_mask<63:60> = Ones(4);
if !IsZero(ptr AND unusedbits_mask) && ((ptr AND unusedbits_mask) !
    if IsFeatureImplemented(FEAT_EPAC) then
        PAC = 0x0000000000000000<63:0>;
    elsif !IsFeatureImplemented(FEAT_PAuth2) then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// Preserve the determination between upper and lower address at bi
// bits that are not used for the address or the tag(s).
if !IsFeatureImplemented(FEAT_PAuth2) then
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bott
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bott
        // A compliant implementation of FEAT_MTE4 also implements
        assert !mtx;
else
    if tbi then
        result = (ptr<63:56>
                    : selbit
                    (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit
                     ptr<bottom_PAC_bit-1:0>));
    elsif mtx then
        result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : selbit
                    (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit
                     ptr<bottom_PAC_bit-1:0>));
    else
        result = ((ptr<63:56> EOR PAC<63:56>
                    : selbit
                    (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit
                     ptr<bottom_PAC_bit-1:0>));
return result;

```

## Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```

// AddPACDA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer auth
// field bits with a pointer authentication code, where the pointer auth
// code is derived using a cryptographic algorithm as a combination of

```

```

// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCLTR_EL1.EnDA else SCLTR_EL2.
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCLTR_EL1.EnDA;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCLTR_EL2.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCLTR_EL3.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

if Enable == '0' then
    return x;
elsif TrapEL3 && EL3SDDUndefPriority() then
    UNDEFINED;
elsif TrapEL2 then
    TrapPACUse(EL2);
elsif TrapEL3 then
    if EL3SDDUndef() then
        UNDEFINED;
    else
        TrapPACUse(EL3);
else
    return AddPAC(x, y, APDAKey_EL1, TRUE);

```

## Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```

// AddPACDB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer auth
// field bits with a pointer authentication code, where the pointer auth
// code is derived using a cryptographic algorithm as a combination of
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

```

```

APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCLTR_EL1.EnDB else SCLTR_EL2.
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCLTR_EL1.EnDB;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCLTR_EL2.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCLTR_EL3.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elseif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse(EL2);
    elseif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);

```

## Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```

// AddPACGA()
// =====
// Returns a 64-bit value where the lower 32 bits are 0, and the upper
// a 32-bit pointer authentication code which is derived using a crypto
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;
    boolean isgeneric = TRUE;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';

```

```

when EL2
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL3
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

if TrapEL3 && EL3SDDUndefPriority() then
    UNDEFINED;
elsif TrapEL2 then
    TrapPACUse(EL2);
elsif TrapEL3 then
    if EL3SDDUndef() then
        UNDEFINED;
    else
        TrapPACUse(EL3);
else
    return ComputePAC(x, y, APIAKey_EL1<127:64>, APIAKey_EL1<63:0>,

```

## Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```

// AddPACIA()
// =====
// Returns a 64-bit value containing x, but replacing the pointer authen-
// field bits with a pointer authentication code, where the pointer authen-
// code is derived using a cryptographic algorithm as a combination of
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCTRLR_EL1.EnIA else SCTRLR_EL2.
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                   (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCTRLR_EL1.EnIA;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCTRLR_EL2.EnIA;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCTRLR_EL3.EnIA;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

if Enable == '0' then
    return x;
elsif TrapEL3 && EL3SDDUndefPriority() then

```

```

        UNDEFINED;
elsif TrapEL2 then
    TrapPACUse(EL2);
elsif TrapEL3 then
    if EL3SDDUndef() then
        UNDEFINED;
    else
        TrapPACUse(EL3);
else
    return AddPAC(x, y, APIAKey_EL1, FALSE);

```

## Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```

// AddPACIB()
// =====
// Returns a 64-bit value containing x, but replacing the pointer auth
// field bits with a pointer authentication code, where the pointer auth
// code is derived using a cryptographic algorithm as a combination of
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

        if Enable == '0' then
            return x;
        elsif TrapEL3 && EL3SDDUndefPriority() then
            UNDEFINED;
        elsif TrapEL2 then
            TrapPACUse(EL2);
        elsif TrapEL3 then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                TrapPACUse(EL3);

```

```

    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);

```

## Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```

// AArch64.PACFailException()
// =====
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == 1;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_PACFail);
    except.syndrome<1:0> = syndrome;
    except.syndrome<24:2> = Zeros(23); // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(EL1, except, preferred_exception_return,

```

## Library pseudocode for aarch64/functions/pac/auth/Auth

```

// Auth()
// =====
// Restores the upper bits of the address to be all zeros or all ones (the
// value of bit[55]) and computes and checks the pointer authentication
// check passes, then the restored address is returned. If the check fails,
// second-top and third-top bits of the extension bits in the pointer
// field are corrupted to ensure that accessing the address will give a
// fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data,
              boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    boolean isgeneric = FALSE;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left for the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return ptr;

    if tbi then
        original_ptr = (ptr<63:56> :

```

```

        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit>;
    elsif mtx then
        original_ptr = (extfield<63:60> : ptr<59:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit>;
    end;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>, isgenerating);
    // Check pointer authentication code
    if tbi then
        if !IsFeatureImplemented(FEAT_PAuth2) then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, 4)
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    elsif mtx then
        assert IsFeatureImplemented(FEAT_PAuth2);
        result = ptr;
        result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
        result<63:60> = result<63:60> EOR PAC<63:60>;
        if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
            (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
            if ((result<54:bottom_PAC_bit> != Replicate(result<55>, 55))
                (result<63:60> != Replicate(result<55>, 4))) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    else
        if !IsFeatureImplemented(FEAT_PAuth2) then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<54:bottom_PAC_bit> != original_ptr
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if result<63:bottom_PAC_bit> != Replicate(result<55>, 3)
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    end;
    return result;

```

## Library pseudocode for aarch64/functions/pac/authda/AuthDA

```

// AuthDA()
// ======
// Returns a 64-bit value containing x, but replacing the pointer auth

```

```

// field bits with the extension of the address bits. The instruction ch
// authentication code in the pointer authentication code field bits of
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCLTR_EL1.EnDA else SCLTR_EL2.
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCLTR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCLTR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCLTR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

        if Enable == '0' then
            return x;
        elseif TrapEL3 && EL3SDDUndefPriority() then
            UNDEFINED;
        elseif TrapEL2 then
            TrapPACUse(EL2);
        elseif TrapEL3 then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                TrapPACUse(EL3);
        else
            return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
    
```

## Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```

// AuthDB()
// ======
// Returns a 64-bit value containing x, but replacing the pointer auth
// field bits with the extension of the address bits. The instruction ch
// pointer authentication code in the pointer authentication code field
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

```

```

bits(128) APDBKey_EL1;

APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCLTR_EL1.EnDB else SCLTR_EL2.
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCLTR_EL1.EnDB;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCLTR_EL2.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCLTR_EL3.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);

```

## Library pseudocode for aarch64/functions/pac/authia/AuthIA

```

// AuthIA()
// ======
// Returns a 64-bit value containing x, but replacing the pointer authetication
// field bits with the extension of the address bits. The instruction changes
// authentication code in the pointer authentication code field bits of
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCLTR_EL1.EnIA else SCLTR_EL2.
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCLTR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCLTR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCLTR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

        if Enable == '0' then
            return x;
        elsif TrapEL3 && EL3SDDUndefPriority() then
            UNDEFINED;
        elsif TrapEL2 then
            TrapPACUse(EL2);
        elsif TrapEL3 then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                TrapPACUse(EL3);
        else
            return Auth(x, y, APIAKey_EL1, TRUE, '1', is_combined);

```

```

        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL1
    Enable = SCTRLR_EL1.EnIA;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL2
    Enable = SCTRLR_EL2.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
when EL3
    Enable = SCTRLR_EL3.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

if Enable == '0' then
    return x;
elsif TrapEL3 && EL3SDDUndefPriority() then
    UNDEFINED;
elsif TrapEL2 then
    TrapPACUse(EL2);
elsif TrapEL3 then
    if EL3SDDUndef() then
        UNDEFINED;
    else
        TrapPACUse(EL3);
else
    return Auth(x, y, APIBKey_EL1, FALSE, '0', is_combined);

```

## Library pseudocode for aarch64/functions/pac/authib/AuthIB

```

// AuthIB()
// ======
// Returns a 64-bit value containing x, but replacing the pointer authib
// field bits with the extension of the address bits. The instruction ch
// authentication code in the pointer authentication code field bits of
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTRLR_EL1.EnIB else SCTRLR_EL2.
            TrapEL2 = (EL2Enabled()) && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTRLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTRLR_EL2.EnIB;

```

```

        TrapEL2 = FALSE;
        TrapEL3 = HaveEL\(EL3\) && SCR_EL3.API == '0';
when EL3
    Enable = SCLTR_EL3.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elseif TrapEL3 && EL3SDDUndefPriority\(\) then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse\(EL2\);
    elseif TrapEL3 then
        if EL3SDDUndef\(\) then
            UNDEFINED;
        else
            TrapPACUse\(EL3\);
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);

```

### **Library pseudocode for aarch64/functions/pac/calcbottompacbit/ AArch64.PACEffectiveTxSZ**

```

// AArch64.PACEffectiveTxSZ()
// =====
// Compute the effective value for TxSZ used to determine the placement

bits(6) AArch64.PACEffectiveTxSZ(Regime regime, S1TTWParams walkparams)
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams..c
                                                walkparams.ds, walkp

    if AArch64.S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable\(Unpredictable RESTnSZ\) == Constraint
            if UInt(walkparams.txsz) < s1mintxsz then
                return s1mintxsz<5:0>;
            if UInt(walkparams.txsz) > s1maxtxsz then
                return s1maxtxsz<5:0>;
        elseif UInt(walkparams.txsz) < s1mintxsz then
            return s1mintxsz<5:0>;
        elseif UInt(walkparams.txsz) > s1maxtxsz then
            return s1maxtxsz<5:0>;

    return walkparams.txsz;

```

### **Library pseudocode for aarch64/functions/pac/calcbottompacbit/ CalculateBottomPACBit**

```

// CalculateBottomPACBit()
// =====

integer CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    integer bottom_PAC_bit;

```

```

regime = TranslationRegime(PSTATE.EL);
ss = CurrentSecurityState();
walkparams = AArch64.GetS1TTWParams(regime, ss, Replicate(top_bit,
bottom_PAC_bit = 64 - UInt(AArch64.PACEffectiveTxSZ(regime, walkpara

return bottom_PAC_bit;

```

### Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```

// ComputePAC()
// =====

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64)
                  boolean isgeneric)
    if UsePACIMP(isgeneric) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if UsePACQARMA3(isgeneric) then
        boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if UsePACQARMA5(isgeneric) then
        boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    Unreachable();

```

### Library pseudocode for aarch64/functions/pac/computepac/ ComputePACIMPDEF

```

// ComputePACIMPDEF()
// =====
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for
// ComputePAC()

bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0,
                           bits(64) key1, boolean isqarma3);

```

### Library pseudocode for aarch64/functions/pac/computepac/ ComputePACQARMA

```

// ComputePACQARMA()
// =====
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                           bits(64) key1, boolean isqarma3)
    bits(64) workingval;
    bits(64) runningmod;
    bits(64) roundkey;
    bits(64) modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;
    integer iterations;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    if isqarma3 then
        iterations = 2;

```

```

else // QARMA5
    iterations = 4;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
runningmod = modifier;
workingval = data EOR key0;

for i = 0 to iterations
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to iterations
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACInvSub(workingval);
    if i < iterations then
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
    runningmod = TweakInvShuffle(runningmod<63:0>);
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR RC[iterations-i];
    workingval = workingval EOR roundkey;
    workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

return workingval;

```

## **Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle**

```
// PACCellInvShuffle()  
// =====  
  
bits(64) PACCellInvShuffle(bits(64) indata)  
    bits(64) outdata;  
    outdata<3:0> = indata<15:12>;  
    outdata<7:4> = indata<27:24>;  
    outdata<11:8> = indata<51:48>;  
    outdata<15:12> = indata<39:36>;  
    outdata<19:16> = indata<59:56>;  
    outdata<23:20> = indata<47:44>;  
    outdata<27:24> = indata<7:4>;  
    outdata<31:28> = indata<19:16>;  
    outdata<35:32> = indata<35:32>;  
    outdata<39:36> = indata<55:52>;  
    outdata<43:40> = indata<31:28>;  
    outdata<47:44> = indata<11:8>;  
    outdata<51:48> = indata<23:20>;  
    outdata<55:52> = indata<3:0>;  
    outdata<59:56> = indata<43:40>;  
    outdata<63:60> = indata<63:60>;  
    return outdata;
```

## **Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle**

```
// PACCellShuffle()  
// =====  
  
bits(64) PACCellShuffle(bits(64) indata)  
    bits(64) outdata;  
    outdata<3:0> = indata<55:52>;  
    outdata<7:4> = indata<27:24>;  
    outdata<11:8> = indata<47:44>;  
    outdata<15:12> = indata<3:0>;  
    outdata<19:16> = indata<31:28>;  
    outdata<23:20> = indata<51:48>;  
    outdata<27:24> = indata<7:4>;  
    outdata<31:28> = indata<43:40>;  
    outdata<35:32> = indata<35:32>;  
    outdata<39:36> = indata<15:12>;  
    outdata<43:40> = indata<59:56>;  
    outdata<47:44> = indata<23:20>;  
    outdata<51:48> = indata<11:8>;  
    outdata<55:52> = indata<39:36>;  
    outdata<59:56> = indata<19:16>;  
    outdata<63:60> = indata<63:60>;  
    return outdata;
```

## **Library pseudocode for aarch64/functions/pac/computepac/PACInvSub**

```
// PACInvSub()  
// =====  
  
bits(64) PACInvSub(bits(64) Tinput)
```

```

// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '0101';
        when '0001' Toutput<4*i+3:4*i> = '1110';
        when '0010' Toutput<4*i+3:4*i> = '1101';
        when '0011' Toutput<4*i+3:4*i> = '1000';
        when '0100' Toutput<4*i+3:4*i> = '1010';
        when '0101' Toutput<4*i+3:4*i> = '1011';
        when '0110' Toutput<4*i+3:4*i> = '0001';
        when '0111' Toutput<4*i+3:4*i> = '1001';
        when '1000' Toutput<4*i+3:4*i> = '0010';
        when '1001' Toutput<4*i+3:4*i> = '0110';
        when '1010' Toutput<4*i+3:4*i> = '1111';
        when '1011' Toutput<4*i+3:4*i> = '0000';
        when '1100' Toutput<4*i+3:4*i> = '0100';
        when '1101' Toutput<4*i+3:4*i> = '1100';
        when '1110' Toutput<4*i+3:4*i> = '0111';
        when '1111' Toutput<4*i+3:4*i> = '0011';
    return Toutput;

```

### Library pseudocode for aarch64/functions/pac/computepac/PACMult

```

// PACMult()
// =====

bits(64) PACMult(bits(64) Sinput)
    bits(4) t0;
    bits(4) t1;
    bits(4) t2;
    bits(4) t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;

```

### Library pseudocode for aarch64/functions/pac/computepac/PACSub

```

// PACSub()
// =====

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;

```

```

for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1011';
        when '0001' Toutput<4*i+3:4*i> = '0110';
        when '0010' Toutput<4*i+3:4*i> = '1000';
        when '0011' Toutput<4*i+3:4*i> = '1111';
        when '0100' Toutput<4*i+3:4*i> = '1100';
        when '0101' Toutput<4*i+3:4*i> = '0000';
        when '0110' Toutput<4*i+3:4*i> = '1001';
        when '0111' Toutput<4*i+3:4*i> = '1110';
        when '1000' Toutput<4*i+3:4*i> = '0011';
        when '1001' Toutput<4*i+3:4*i> = '0111';
        when '1010' Toutput<4*i+3:4*i> = '0100';
        when '1011' Toutput<4*i+3:4*i> = '0101';
        when '1100' Toutput<4*i+3:4*i> = '1101';
        when '1101' Toutput<4*i+3:4*i> = '0010';
        when '1110' Toutput<4*i+3:4*i> = '0001';
        when '1111' Toutput<4*i+3:4*i> = '1010';
    return Toutput;

```

### Library pseudocode for aarch64/functions/pac/computepac/PacSub1

```

// PacSub1()
// =====

bits(64) PACSub1(bits(64) Tinput)
    // This is a 4-bit substitution from Qarma sigma1
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' Toutput<4*i+3:4*i> = '1010';
            when '0001' Toutput<4*i+3:4*i> = '1101';
            when '0010' Toutput<4*i+3:4*i> = '1110';
            when '0011' Toutput<4*i+3:4*i> = '0110';
            when '0100' Toutput<4*i+3:4*i> = '1111';
            when '0101' Toutput<4*i+3:4*i> = '0111';
            when '0110' Toutput<4*i+3:4*i> = '0011';
            when '0111' Toutput<4*i+3:4*i> = '0101';
            when '1000' Toutput<4*i+3:4*i> = '1001';
            when '1001' Toutput<4*i+3:4*i> = '1000';
            when '1010' Toutput<4*i+3:4*i> = '0000';
            when '1011' Toutput<4*i+3:4*i> = '1100';
            when '1100' Toutput<4*i+3:4*i> = '1011';
            when '1101' Toutput<4*i+3:4*i> = '0001';
            when '1110' Toutput<4*i+3:4*i> = '0010';
            when '1111' Toutput<4*i+3:4*i> = '0100';
    return Toutput;

```

### Library pseudocode for aarch64/functions/pac/computepac/RC

```

// RC[]
// ===

array bits(64) RC[0..4];

```

## Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =====

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>;incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =====

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// =====

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =====

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
```

```

    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
}

```

### Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```

// TweakShuffle()
// =====

bits(64) TweakShuffle(bits(64) indata)
{
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
}

```

### Library pseudocode for aarch64/functions/pac/computepac/UsePACIMP

```

// UsePACIMP()
// =====
// Checks whether IMPLEMENTATION DEFINED cryptographic algorithm to be
// calculation.

boolean UsePACIMP(boolean isgeneric)
    return if isgeneric then HavePACIMPGeneric\(\) else HavePACIMPAuth\(\);

```

### Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA3

```

// UsePACQARMA3()
// =====
// Checks whether QARMA3 cryptographic algorithm to be used for PAC cal

boolean UsePACQARMA3(boolean isgeneric)
    return if isgeneric then HavePACQARMA3Generic\(\) else HavePACQARMA3A;

```

## **Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA5**

```
// UsePACQARMA5()
// =====
// Checks whether QARMA5 cryptographic algorithm to be used for PAC call

boolean UsePACQARMA5(boolean isgeneric)
    return if isgeneric then HavePACQARMA5Generic\(\) else HavePACQARMA5A;
```

## **Library pseudocode for aarch64/functions/pac/pac/ConstPACField**

```
// ConstPACField()
// =====
// Returns TRUE if bit<55> can be used to determine the size of the PAC

boolean ConstPACField()
    return IsFeatureImplemented(FEAT_CONSTPACFIELD);
```

## **Library pseudocode for aarch64/functions/pac/pac/HavePACIMPAuth**

```
// HavePACIMPAuth()
// =====
// Returns TRUE if support for PAC IMP Auth is implemented, FALSE otherwise

boolean HavePACIMPAuth()
    return IsFeatureImplemented(FEAT_PACIMP);
```

## **Library pseudocode for aarch64/functions/pac/pac/HavePACIMPGeneric**

```
// HavePACIMPGeneric()
// =====
// Returns TRUE if support for PAC IMP Generic is implemented, FALSE otherwise

boolean HavePACIMPGeneric()
    return IsFeatureImplemented(FEAT_PACIMP);
```

## **Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Auth**

```
// HavePACQARMA3Auth()
// =====
// Returns TRUE if support for PAC QARMA3 Auth is implemented, FALSE otherwise

boolean HavePACQARMA3Auth()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

## **Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Generic**

```
// HavePACQARMA3Generic()
// =====
// Returns TRUE if support for PAC QARMA3 Generic is implemented, FALSE otherwise
```

```
boolean HavePACQARMA3Generic()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

### Library pseudocode for aarch64/functions/pac/pac/**HavePACQARMA5Auth**

```
// HavePACQARMA5Auth()
// =====
// Returns TRUE if support for PAC QARMA5 Auth is implemented, FALSE otherwise
boolean HavePACQARMA5Auth()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

### Library pseudocode for aarch64/functions/pac/pac/**HavePACQARMA5Generic**

```
// HavePACQARMA5Generic()
// =====
// Returns TRUE if support for PAC QARMA5 Generic is implemented, FALSE otherwise
boolean HavePACQARMA5Generic()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

### Library pseudocode for aarch64/functions/pac/pac/**PtrHasUpperAndLowerAddRanges**

```
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise
boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL);
    return HasUnprivileged(regime);
```

### Library pseudocode for aarch64/functions/pac/strip/**Strip**

```
// Strip()
// =====
// Strip() returns a 64-bit value containing A, but replacing the point
// code field bits with the extension of the address bits. This can apply
// to instructions or data, where, as the use of tagged pointers is distin
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left for the PAC
    if tbi && bottom_PAC_bit >= 55 then
```

```

        return A;

    if tbi then
        original_ptr = (A<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit>;
    elseif mtx then
        original_ptr = (extfield<63:60> : A<59:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit>;
    end;
endfunction

```

## Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```

// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= 1;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0;
    except = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, except, preferred_exception_return);

```

## Library pseudocode for aarch64/functions/predictionrestrict/ AArch64.RestrictPrediction

```

// AArch64.RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCtxt c;
    target_el      = val<25:24>;

    // If the target EL is not implemented or the instruction is executed
    // at a lower level than the specified level, the instruction is treated as
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then EndOfInstruction();

    bit ns   = val<26>;
    bit nse = val<27>;
    ss = TargetSecurityState(ns, nse);

    // If the combination of Security state and Exception level is not
    // the instruction is treated as a NOP.
    if ss == SS_Root && target_el != EL3 then EndOfInstruction();
    if !IsFeatureImplemented(FEAT_RME) && target_el == EL3 && ss != SS_Root then EndOfInstruction();

    c.security  = ss;
    c.target_el = target_el;

```

```

if EL2Enabled() then
    if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
        c.is_vmid_valid = TRUE;
        c.all_vmid      = FALSE;
        c.vmid          = VMID[];

    elseif (target_el == EL0 && !ELIsInHost(target_el)) || target_el ==
        c.is_vmid_valid = TRUE;
        c.all_vmid      = val<48> == '1';
        c.vmid          = val<47:32>;           // Only valid if val<47:32> == '1'

    else
        c.is_vmid_valid = FALSE;
else
    c.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    c.is_asid_valid = TRUE;
    c.all_asid     = FALSE;
    c.asid          = ASID[];

elseif target_el == EL0 then
    c.is_asid_valid = TRUE;
    c.all_asid     = val<16> == '1';
    c.asid          = val<15:0>;           // Only valid if val<15:0> == '1'

else
    c.is_asid_valid = FALSE;

c.restriction = restriction;
RESTRICT_PREDICTIONS(c);

```

## Library pseudocode for aarch64/functions/prefetch/Prefetch

```

// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch READ;           // PLD: prefetch for load
        when '01' hint = Prefetch EXEC;           // PLI: preload instruction
        when '10' hint = Prefetch WRITE;          // PST: prepare for store
        when '11' return;                           // unallocated hint
    target = UInt(prfop<2:1>);                 // target cache level
    stream = (prfop<0> != '0');                  // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;

```

## Library pseudocode for aarch64/functions/pstatefield/PSTATEField

```

// PSTATEField
// =====
// MSR (immediate) instruction destinations.

enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
    PSTATEField_PAN, // Armv8.1
    PSTATEField_UAO, // Armv8.2
    PSTATEField_DIT, // Armv8.4
    PSTATEField_SSBS,
    PSTATEField_TCO, // Armv8.5
    PSTATEField_SVCRSM,
    PSTATEField_SVCRZA,
    PSTATEField_SVCRSMZA,
    PSTATEField_ALLINT,
    PSTATEField_PM,
    PSTATEField_SP
};

```

### Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```

// AArch64.ESBOperation()
// =====
// Perform the AArch64 ESB operation, either for ESB executed in AArch64
// ESB in AArch32 state when SError interrupts are routed to an Exception
// AArch64

AArch64.ESBOperation()
    bits(2) target_el;
    boolean masked;

    (masked, target_el) = AArch64.PhysicalSErrorTarget\(\);

    intdis = Halted\(\) || ExternalDebugInterruptsDisabled\(target\_el\);
    masked = masked || intdis;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending\(\) then
        // This function might be called for an interworking case, and
        // the SError interrupt.
        if ELUsingAArch32\(S1TranslationRegime\(\)\) then
            bits(32) syndrome = Zeros(32);
            syndrome<31> = '1'; // A
            syndrome<15:0> = AArch32.PhysicalSErrorSyndrome\(\);
            DISR = syndrome;
        else
            implicit_esb = FALSE;
            bits(64) syndrome = Zeros(64);
            syndrome<31> = '1'; // A
            syndrome<24:0> = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = syndrome;
            ClearPendingPhysicalSError\(\); // Set ISR_EL1.A to
    return;

```

### Library pseudocode for aarch64/functions/ras/ AArch64.EncodeAsyncErrorSyndrome

```

// AArch64.EncodeAsyncErrorSyndrome()
// =====
// Return the encoding for corresponding ErrorState.

bits(3) AArch64.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState\_UC return '000';
        when ErrorState\_UEU return '001';
        when ErrorState\_UEO return '010';
        when ErrorState\_UER return '011';
        when ErrorState\_CE return '110';
        otherwise Unreachable\(\);

```

### **Library pseudocode for aarch64/functions/ras/ AArch64.EncodeSyncErrorSyndrome**

```

// AArch64.EncodeSyncErrorSyndrome()
// =====
// Return the encoding for corresponding ErrorState.

bits(2) AArch64.EncodeSyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState\_UC return '10';
        when ErrorState\_UEO return '11';
        when ErrorState\_UER return '00';
        otherwise Unreachable\(\);

```

### **Library pseudocode for aarch64/functions/ras/ AArch64.PhysicalSError Syndrome**

```

// AArch64.PhysicalSError Syndrome()
// =====
// Generate SError syndrome.

bits(25) AArch64.PhysicalSError Syndrome(boolean implicit_esb)
    bits(25) syndrome = Zeros(25);
    FaultRecord fault = GetPendingPhysicalSError\(\);
    ErrorState errorstate = AArch64.PEErrorState(fault);
    if errorstate == ErrorState\_Uncategorized then
        syndrome = Zeros(25);
    elseif errorstate == ErrorState\_IMPDEF then
        syndrome<24> = '1';
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorS
    else
        syndrome<24> = '0';
        syndrome<13> = (if implicit_esb then '1' else '0');
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate);
        syndrome<9> = fault.extflag;
        syndrome<5:0> = '010001';
    return syndrome;

```

### **Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation**

```

// AArch64.vESBOperation()
// =====

```

```

// Perform the AArch64 ESB operation for virtual SError interrupts, either
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using
// the VDISR_EL2 register.

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // If physical SError interrupts are routed to EL2, and TGE is not
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked       = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            bits(32) target = Zeros(32);
            target<31> = '1';                                // A
            target<15:14> = VDFSR<15:14>;                // AET
            target<12> = VDFSR<12>;                      // Ext
            target<9> = TTBCR.EAE;                         // LPAE
            if TTBCR.EAE == '1' then                         // Long-descriptor format
                target<5:0> = '010001';                     // STATUS
            else
                target<10,3:0> = '10110';                   // FS
            VDISR = target;
        else
            bits(64) target = Zeros(64);
            target<31> = '1';                                // A
            target<24:0> = VSESER_EL2<24:0>;
            VDISR_EL2 = target;
            HCR_EL2.VSE = '0';                            // Clear pending virtual
                                                       // SError interrupt
    return;

```

## Library pseudocode for aarch64/functions/ras/FirstRecordOfNode

```

// FirstRecordOfNode()
// =====
// Return the first record in the node that contains the record n.

integer FirstRecordOfNode(integer n)
    for q = n downto 0
        if IsFirstRecordOfNode(q) then return q;
    Unreachable();

```

## Library pseudocode for aarch64/functions/ras/ IsCommonFaultInjectionImplemented

```

// IsCommonFaultInjectionImplemented()
// =====
// Check if the Common Fault Injection Model Extension is implemented by
// error record.

boolean IsCommonFaultInjectionImplemented(integer n);

```

### **Library pseudocode for aarch64/functions/ras/IsCountableErrorsRecorded**

```
// IsCountableErrorsRecorded()  
// =====  
// Check whether Error record n records countable errors.  
  
boolean IsCountableErrorsRecorded(integer n);
```

### **Library pseudocode for aarch64/functions/ras/IsErrorAddressIncluded**

```
// IsErrorAddressIncluded()  
// =====  
// Check whether Error record n includes an address associated with an  
  
boolean IsErrorAddressIncluded(integer n);
```

### **Library pseudocode for aarch64/functions/ras/IsErrorRecordImplemented**

```
// IsErrorRecordImplemented()  
// =====  
// Is the error record n implemented  
  
boolean IsErrorRecordImplemented(integer n);
```

### **Library pseudocode for aarch64/functions/ras/IsFirstRecordOfNode**

```
// IsFirstRecordOfNode()  
// =====  
// Check if the record q is the first error record in its node.  
  
boolean IsFirstRecordOfNode(integer q);
```

### **Library pseudocode for aarch64/functions/ras/IsSPMUCounterImplemented**

```
// IsSPMUCounterImplemented()  
// =====  
// Does the System PMU s implement the counter n.  
  
boolean IsSPMUCounterImplemented(integer s, integer n);
```

### **Library pseudocode for aarch64/functions/rcw/ProtectionEnabled**

```
// ProtectionEnabled()  
// =====  
// Returns TRUE if the ProtectedBit is  
// enabled in the current Exception level.  
  
boolean ProtectionEnabled(bits(2) el)  
    assert HaveEL(el);
```

```

regime = S1TranslationRegime(el);
assert(!ELUsingAArch32(regime));
if (!ISD128Enabled(el)) then
    case regime of
        when EL1
            return IsTCR2EL1Enabled() && TCR2_EL1.PnCH == '1';
        when EL2
            return IsTCR2EL2Enabled() && TCR2_EL2.PnCH == '1';
        when EL3
            return TCR_EL3.PnCH == '1';
    else
        return TRUE;
return FALSE;

```

## Library pseudocode for aarch64/functions/rcw/RCW128\_PROTECTED\_BIT

```
constant integer RCW128_PROTECTED_BIT = 114;
```

## Library pseudocode for aarch64/functions/rcw/RCW64\_PROTECTED\_BIT

```
constant integer RCW64_PROTECTED_BIT = 52;
```

## Library pseudocode for aarch64/functions/rcw/RCWCheck

```

// RCWCheck()
// =====
// Returns nzcv based on : if the new value for RCW/RCWS instructions set
// Z is set to 1 if RCW checks fail
// C is set to 0 if RCWS checks fail

bits(4) RCWCheck(bits(N) old, bits(N) new, boolean soft)
    assert N IN {64,128};
    integer protectedbit = if N == 128 then RCW128_PROTECTED_BIT else RCW64_PROTECTED_BIT;
    boolean rcw_fail = FALSE;
    boolean rcws_fail = FALSE;
    boolean rcw_state_fail = FALSE;
    boolean rcws_state_fail = FALSE;
    boolean rcw_mask_fail = FALSE;
    boolean rcws_mask_fail = FALSE;

    //Effective RCWMask calculation
    bits(N) rcwmask = RCWMASK_EL1<N-1:0>;
    if N == 64 then
        rcwmask<49:18> = Replicate(rcwmask<17>,32);
        rcwmask<0> = '0';
    else
        rcwmask<55:17> = Replicate(rcwmask<16>,39);
        rcwmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);

    //Effective RCWSMask calculation
    bits(N) rcwsoftmask = RCWSMASK_EL1<N-1:0>;
    if N == 64 then
        rcwsoftmask<49:18> = Replicate(rcwsoftmask<17>,32);
        rcwsoftmask<0> = '0';
        if(ProtectionEnabled(PSTATE.EL)) then

```

```

        rcwsoftmask<52> = '0';
    else
        rcwsoftmask<55:17> = Replicate(rcwsoftmask<16>, 39);
        rcwsoftmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);
        rcwsoftmask<114> = '0';

//RCW Checks
//State Check
if (ProtectionEnabled(PSTATE.EL)) then
    if old<protectedbit> == '1' then
        rcw_state_fail = new<protectedbit,0> != old<protectedbit,0>;
    elseif old<protectedbit> == '0' then
        rcw_state_fail = new<protectedbit> != old<protectedbit>;
else
    rcw_state_fail = 0;

//Mask Check
if (ProtectionEnabled(PSTATE.EL)) then
    if old<protectedbit,0> == '11' then
        rcw_mask_fail = !IsZero((new EOR old) AND NOT(rcwmask));
    else
        rcw_mask_fail = 0;

//RCWS Checks
if soft then
    //State Check
    if old<0> == '1' then
        rcws_state_fail = new<0> != old<0>;
    elseif (!ProtectionEnabled(PSTATE.EL) ||
            (ProtectionEnabled(PSTATE.EL) && old<protectedbit> == '0')) then
        rcws_state_fail = new<0> != old<0>;
    //Mask Check
    if old<0> == '1' then
        rcws_mask_fail = !IsZero((new EOR old) AND NOT(rcwsoftmask));
    else
        rcws_mask_fail = 0;

rcw_fail = rcw_state_fail || rcw_mask_fail;
rcws_fail = rcws_state_fail || rcws_mask_fail;

bit n = '0';
bit z = if rcw_fail then '1' else '0';
bit c = if rcws_fail then '0' else '1';
bit v = '0';
return <n, z, c, v>;

```

## Library pseudocode for aarch64/functions/reduceop/Reduce

```

// Reduce()
// =====

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && !IsFP16();
    return Reduce(op, input, esize, altfp);

// Reduce()
// =====
// Perform the operation 'op' on pairs of elements from the input vector
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behavior.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    assert esize IN {8,16,32,64};
    bits(esize) hi;

```

```

bits(esize) lo;
bits(esize) result;

if N == esize then
    return input<esize-1:0>;

constant integer half = N DIV 2;
hi = Reduce(op, input<N-1:half>, esize, altp);
lo = Reduce(op, input<half-1:0>, esize, altp);

case op of
    when ReduceOp_FMINNUM
        result = FPMinNum(lo, hi, FPCR[]);
    when ReduceOp_FMAXNUM
        result = FPMaxNum(lo, hi, FPCR[]);
    when ReduceOp_FMIN
        result = FPMin(lo, hi, FPCR[], altp);
    when ReduceOp_FMAX
        result = FPMax(lo, hi, FPCR[], altp);
    when ReduceOp_FADD
        result = FPAdd(lo, hi, FPCR[]);
    when ReduceOp_ADD
        result = lo + hi;

return result;

```

## Library pseudocode for aarch64/functions/reduceop/ReduceOp

```

// ReduceOp
// ======
// Vector reduce instruction types.

enumeration ReduceOp { ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                        ReduceOp_FMIN, ReduceOp_FMAX,
                        ReduceOp_FADD, ReduceOp_ADD };

```

## Library pseudocode for aarch64/functions/registers/ AArch64.MaybeZeroRegisterUppers

```

// AArch64.MaybeZeroRegisterUppers()
// =====
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED U
// 32 bits of registers visible at any lower Exception level using AArch32

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32(); // Always called from AArch32 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elseif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2)
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;

```

```

        for n = first to last
            if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpre
                _R[n]<63:32> = Zeros(32);

        return;
    
```

### **Library pseudocode for aarch64/functions/registers/ AArch64.ResetGeneralRegisters**

```

// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i, 64] = bits(64) UNKNOWN;

    return;

```

### **Library pseudocode for aarch64/functions/registers/ AArch64.ResetSIMDFPRegisters**

```

// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;

```

### **Library pseudocode for aarch64/functions/registers/ AArch64.ResetSpecialRegisters**

```

// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to

```

```

if HaveAArch32EL(EL1) then
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;

```

### **Library pseudocode for aarch64/functions/registers/ AArch64.ResetSystemRegisters**

```

// AArch64.ResetSystemRegisters()
// =====
AArch64.ResetSystemRegisters(boolean cold_reset);

```

### **Library pseudocode for aarch64/functions/registers/PC64**

```

// Program counter
// ++++++++
// PC64 - non-assignment form
// =====
// Read program counter.

bits(64) PC64
    return _PC;

```

### **Library pseudocode for aarch64/functions/registers/SP**

```

// SP[] - assignment form
// =====
// Write to stack pointer from a 64-bit value.

SP[] = bits(64) value
    if PSTATE.SP == '0' then
        SP_EL0 = value;
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = value;
            when EL1  SP_EL1 = value;
            when EL2  SP_EL2 = value;
            when EL3  SP_EL3 = value;
    return;

// SP[] - non-assignment form
// =====
// Read stack pointer with slice of 64 bits.

bits(64) SP[]
    if PSTATE.SP == '0' then

```

```

        return SP_EL0;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0;
            when EL1  return SP_EL1;
            when EL2  return SP_EL2;
            when EL3  return SP_EL3;

```

### **Library pseudocode for aarch64/functions/registers/SPMCFGR\_EL1**

```

// SPMCFGR_EL1[] - non-assignment form
// =====
// Read the current configuration of System Performance monitor for
// System PMU 's'.

bits(64) SPMCFGR_EL1[integer s];

```

### **Library pseudocode for aarch64/functions/registers/SPMCGCR\_EL1**

```

// SPMCGCR_EL1[] - non-assignment form
// =====
// Read counter group 'n' configuration for System PMU 's'.

bits(64) SPMCGCR_EL1[integer s, integer n];

```

### **Library pseudocode for aarch64/functions/registers/SPMCNTENCLR\_EL0**

```

// SPMCNTENCLR_EL0[] - non-assignment form
// =====
// Read the current mapping of disabled event counters for an 's'.

bits(64) SPMCNTENCLR_EL0[integer s];

// SPMCNTENCLR_EL0[] - assignment form
// =====
// Disable event counters for System PMU 's'.

SPMCNTENCLR_EL0[integer s] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMCNTENSET\_EL0**

```

// SPMCNTENSET_EL0[] - non-assignment form
// =====
// Read the current mapping for enabled event counters of System PMU 's'

bits(64) SPMCNTENSET_EL0[integer s];

// SPMCNTENSET_EL0[] - assignment form
// =====
// Enable event counters of System PMU 's'.

SPMCNTENSET_EL0[integer s] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMCR\_EL0**

```
// SPMCR_EL0[] - non-assignment form  
// =====  
// Read the control register for System PMU 's'.  
  
bits(64) SPMCR_EL0[integer s];  
  
// SPMCR_EL0[] - assignment form  
// =====  
// Write to the control register for System PMU 's'.  
  
SPMCR_EL0[integer s] = bits(64) value;
```

### **Library pseudocode for aarch64/functions/registers/SPMDEVAFF\_EL1**

```
// SPMDEVAFF_EL1[] - non-assignment form  
// =====  
// Read the discovery information for System PMU 's'.  
  
bits(64) SPMDEVAFF_EL1[integer s];
```

### **Library pseudocode for aarch64/functions/registers/SPMDEVARCH\_EL1**

```
// SPMDEVARCH_EL1[] - non-assignment form  
// =====  
// Read the discovery information for System PMU 's'.  
  
bits(64) SPMDEVARCH_EL1[integer s];
```

### **Library pseudocode for aarch64/functions/registers/SPMEVCNTR\_EL0**

```
// SPMEVCNTR_EL0[] - non-assignment form  
// =====  
// Read a System PMU Event Counter register for counter 'n' of a given  
// System PMU 's'.  
  
bits(64) SPMEVCNTR_EL0[integer s, integer n];  
  
// SPMEVCNTR_EL0[] - assignment form  
// =====  
// Write to a System PMU Event Counter register for counter 'n' of a gi  
// System PMU 's'.  
  
SPMEVCNTR_EL0[integer s, integer n] = bits(64) value;
```

### **Library pseudocode for aarch64/functions/registers/SPMEVFILT2R\_EL0**

```
// SPMEVFILT2R_EL0[] - non-assignment form  
// =====  
// Read the additional event selection controls for  
// counter 'n' of a given System PMU 's'.
```

```

bits(64) SPMEVFILT2R_EL0[integer s, integer n];

// SPMEVFILT2R_EL0[] - assignment form
// =====
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILT2R_EL0[integer s, integer n] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMEVFILTR\_EL0**

```

// SPMEVFILTR_EL0[] - non-assignment form
// =====
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILTR_EL0[integer s, integer n];

// SPMEVFILTR_EL0[] - assignment form
// =====
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILTR_EL0[integer s, integer n] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMEVTPER\_EL0**

```

// SPMEVTPER_EL0[] - non-assignment form
// =====
// Read the current mapping of event with event counter SPMEVCNTR_EL0
// for counter 'n' of a given System PMU 's'.

bits(64) SPMEVTPER_EL0[integer s, integer n];

// SPMEVTPER_EL0[] - assignment form
// =====
// Configure which event increments the event counter SPMEVCNTR_EL0, for
// counter 'n' of a given System PMU 's'.

SPMEVTPER_EL0[integer s, integer n] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMIIDR\_EL1**

```

// SPMIIDR_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMIIDR_EL1[integer s];

```

### **Library pseudocode for aarch64/functions/registers/SPMINTENCLR\_EL1**

```

// SPMINTENCLR_EL1[] - non-assignment form
// =====
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENCLR_EL1[integer s];

// SPMINTENCLR_EL1[] - assignment form
// =====
// Disable the generation of interrupt requests on overflows of
// implemented counters of System PMU 's'.

SPMINTENCLR_EL1[integer s] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMINSET\_EL1**

```

// SPMINTENSET_EL1[] - non-assignment form
// =====
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENSET_EL1[integer s];

// SPMINTENSET_EL1[] - assignment form
// =====
// Disable the generation of interrupt requests on overflows of
// implemented counters for System PMU 's'.

SPMINSET_EL1[integer s] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMOVSCLR\_EL0**

```

// SPMOVSCLR_EL0[] - non-assignment form
// =====
// Read the overflow bit clear status of implemented counters for System

bits(64) SPMOVSCLR_EL0[integer s];

// SPMOVSCLR_EL0[] - assignment form
// =====
// Clear the overflow bit clear status of implemented counters for
// System PMU 's'.

SPMOVSCLR_EL0[integer s] = bits(64) value;

```

### **Library pseudocode for aarch64/functions/registers/SPMOVSET\_EL0**

```

// SPMOVSET_EL0[] - non-assignment form
// =====
// Read state of the overflow bit for the implemented event counters
// of System PMU 's'.

bits(64) SPMOVSET_EL0[integer s];

// SPMOVSET_EL0[] - assignment form

```

```

// =====
// Sets the state of the overflow bit for the implemented event counters
// of System PMU 's'.

SPMOVSSET_ELO[integer s] = bits(64) value;

```

### Library pseudocode for aarch64/functions/registers/SPMROOTCR\_EL3

```

// SPMROOTCR_EL3[] - non-assignment form
// =====
// Read the observability of Root and Realm events by System Performance
// Monitor for System PMU 's'.

bits(64) SPMROOTCR_EL3[integer s];

// SPMROOTCR_EL3[] - assignment form
// =====
// Configure the observability of Root and Realm events by System
// Performance Monitor for System PMU 's'.

SPMROOTCR_EL3[integer s] = bits(64) value;

```

### Library pseudocode for aarch64/functions/registers/SPMSSCR\_EL1

```

// SPMSSCR_EL1[] - non-assignment form
// =====
// Read the observability of Secure events by System Performance Monitor
// for System PMU 's'.

bits(64) SPMSSCR_EL1[integer s];

// SPMSSCR_EL1[] - assignment form
// =====
// Configure the observability of secure events by System Performance
// Monitor for System PMU 's'.

SPMSSCR_EL1[integer s] = bits(64) value;

```

### Library pseudocode for aarch64/functions/registers/V

```

// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    constant integer vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL
        if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
            _Z[n] = ZeroExtend(value, MAX\_VL);
        else
            _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);
    // V[] - non-assignment form

```

```

// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;

```

## Library pseudocode for aarch64/functions/registers/Vpart

```

// Vpart[] - non-assignment form
// =====
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the
// part 1 returns the top half of the bottom 64 bits or the top half of the
// value held in the register.

bits(width) Vpart[integer n, integer part, integer width]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n, width];
    else
        assert width IN {32,64};
        bits(128) vreg = V[n, 128];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register
// part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n, width] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n, 64];
        V[n, 128] = value<63:0> : vreg;

```

## Library pseudocode for aarch64/functions/registers/X

```

// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value
// where the size of the value is passed as an argument.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then

```

```

        _R[n] = ZeroExtend(value, 64);
        return;

// X[] - non-assignment form
// =====
// Read from general-purpose register with an explicit slice of 8, 16,
bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);

```

### Library pseudocode for aarch64/functions/shiftreg/DecodeShift

```

// DecodeShift()
// =====
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;

```

### Library pseudocode for aarch64/functions/shiftreg/ShiftReg

```

// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount, integ
    bits(N) result = X[reg, N];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;

```

### Library pseudocode for aarch64/functions/shiftreg/ShiftType

```

// ShiftType
// =====
// AArch64 register shifts.

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, S

```

### Library pseudocode for aarch64/functions/sme/CounterToPredicate

```

// CounterToPredicate()
// =====

bits(width) CounterToPredicate(bits(16) pred, integer width)
    integer count;
    integer esize;
    integer elements;
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    constant integer maxbit = HighestSetBit(CeilPow2(PL * 4)<15:0>);
    assert maxbit <= 14;
    bits(PL*4) result;
    boolean invert = pred<15> == '1';

    assert width == PL || width == PL*2 || width == PL*3 || width == PL*4;

    if IsZero(pred<3:0>) then
        return Zeros(width);

    case pred<3:0> of
        when 'xxxx'
            count = UInt(pred<maxbit:1>);
            esize = 8;
        when 'xx10'
            count = UInt(pred<maxbit:2>);
            esize = 16;
        when 'x100'
            count = UInt(pred<maxbit:3>);
            esize = 32;
        when '1000'
            count = UInt(pred<maxbit:4>);
            esize = 64;

        elements = (VL * 4) DIV esize;
        result = Zeros(PL*4);
        constant integer psize = esize DIV 8;
        for e = 0 to elements-1
            bit pbit = if e < count then '1' else '0';
            if invert then
                pbit = NOT(pbit);
            Elem[result, e, psize] = ZeroExtend(pbit, psize);

    return result<width-1:0>;

```

## Library pseudocode for aarch64/functions/sme/EncodePredCount

```

// EncodePredCount()
// =====

bits(width) EncodePredCount(integer esize, integer elements,
                            integer count_in, boolean invert_in, integer width)
    integer count = count_in;
    boolean invert = invert_in;
    constant integer PL = CurrentVL DIV 8;
    assert width == PL;
    assert esize IN {8, 16, 32, 64};
    assert count >= 0 && count <= elements;
    bits(16) pred;

```

```

if count == 0 then
    return Zeros(width);

if invert then
    count = elements - count;
elsif count == elements then
    count = 0;
    invert = TRUE;

bit inv = (if invert then '1' else '0');
case esize of
    when 8 pred = inv : count<13:0> :      '1';
    when 16 pred = inv : count<12:0> :     '10';
    when 32 pred = inv : count<11:0> :    '100';
    when 64 pred = inv : count<10:0> :   '1000';

return ZeroExtend(pred, width);

```

### **Library pseudocode for aarch64/functions/sme/HaveSME**

```

// HaveSME()
// ======
// Returns TRUE if the SME extension is implemented, FALSE otherwise.

boolean HaveSME()
    return IsFeatureImplemented(FEAT_SME);

```

### **Library pseudocode for aarch64/functions/sme/HaveSME2**

```

// HaveSME2()
// ======
// Returns TRUE if the SME2 extension is implemented, FALSE otherwise.

boolean HaveSME2()
    return IsFeatureImplemented(FEAT_SME2);

```

### **Library pseudocode for aarch64/functions/sme/HaveSME2p1**

```

// HaveSME2p1()
// ======
// Returns TRUE if the SME2.1 extension is implemented, FALSE otherwise.

boolean HaveSME2p1()
    return IsFeatureImplemented(FEAT_SME2p1);

```

### **Library pseudocode for aarch64/functions/sme/HaveSMEB16B16**

```

// HaveSMEB16B16()
// =====
// Returns TRUE if the SME2.1 non-widening BFloat16 instructions are implemented

boolean HaveSMEB16B16()
    return IsFeatureImplemented(FEAT_SVE_B16B16);

```

### **Library pseudocode for aarch64/functions/sme/**HaveSMEF16F16****

```

// HaveSMEF16F16()
// =====
// Returns TRUE if the SME2.1 half-precision instructions are implemented

boolean HaveSMEF16F16()
    return IsFeatureImplemented(FEAT_SME_F16F16);

```

### **Library pseudocode for aarch64/functions/sme/**HaveSMEF64F64****

```

// HaveSMEF64F64()
// =====
// Returns TRUE if the SMEF64F64 extension is implemented, FALSE otherwise

boolean HaveSMEF64F64()
    return IsFeatureImplemented(FEAT_SME_F64F64);

```

### **Library pseudocode for aarch64/functions/sme/**HaveSMEI16I64****

```

// HaveSMEI16I64()
// =====
// Returns TRUE if the SMEI16I64 extension is implemented, FALSE otherwise

boolean HaveSMEI16I64()
    return IsFeatureImplemented(FEAT_SME_I16I64);

```

### **Library pseudocode for aarch64/functions/sme/**Lookup****

```
bits(512) _ZT0;
```

### **Library pseudocode for aarch64/functions/sme/**PredCountTest****

```

// PredCountTest()
// =====

bits(4) PredCountTest(integer elements, integer count, boolean invert)
    bit n, z, c, v;
    z = (if count == 0 then '1' else '0');                                // none active
    if !invert then
        n = (if count != 0 then '1' else '0');                            // first active
        c = (if count == elements then '0' else '1');                      // NOT last active
    else
        n = (if count == elements then '1' else '0');                      // first active

```

```

        c = (if count != 0 then '0' else '1');           // NOT last act
        v = '0';

        return n:z:c:v;

```

### Library pseudocode for aarch64/functions/sme/System

```

// System Registers
// =====

array bits(MAX_VL) _ZA[0..255];

```

### Library pseudocode for aarch64/functions/sme/ZAhslice

```

// ZAhslice[] - non-assignment form
// =====

bits(width) ZAhslice[integer tile, integer esize, integer slice, integer width]
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// =====

ZAhslice[integer tile, integer esize, integer slice, integer width] = b
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;

```

### Library pseudocode for aarch64/functions/sme/ZAslice

```

// ZAslice[] - non-assignment form
// =====

bits(width) ZAslice[integer tile, integer esize, boolean vertical, integer width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice[] - assignment form
// =====

```

```

ZAslice[integer tile, integer esize, boolean vertical,
        integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;

```

### Library pseudocode for aarch64/functions/sme/ZAtile

```

// ZAtile[] - non-assignment form
// =====

bits(width) ZAtile[integer tile, integer esize, integer width]
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];

    return result;

// ZAtile[] - assignment form
// =====

ZAtile[integer tile, integer esize, integer width] = bits(width) value
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];

```

### Library pseudocode for aarch64/functions/sme/ZAvector

```

// ZAvector[] - non-assignment form
// =====

bits(width) ZAvector[integer index, integer width]
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    return _ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// =====

ZAvector[integer index, integer width] = bits(width) value
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
        _ZA[index] = ZeroExtend(value, MAX VL);
    else
        _ZA[index]<width-1:0> = value;

```

## Library pseudocode for aarch64/functions/sme/ZAvslice

```
// ZAvslice[] - non-assignment form
// =====

bits(width) ZAvslice[integer tile, integer esize, integer slice, integer width]
    integer slices = CurrentSVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];

    return result;

// ZAvslice[] - assignment form
// =====

ZAvslice[integer tile, integer esize, integer slice, integer width] = b
    integer slices = CurrentSVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

## Library pseudocode for aarch64/functions/sme/ZT0

```
// ZT0[] - non-assignment form
// =====

bits(width) ZT0[integer width]
    assert width == 512;
    return _ZT0<width-1:0>;

// ZT0[] - assignment form
// =====

ZT0[integer width] = bits(width) value
    assert width == 512;
    _ZT0<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() ==
        // Check if access disabled in NSACR
```

```

        if NSACR.cp10 == '0' then return FALSE;

        if el IN {EL0, EL1} then
            // Check if access disabled in CPACR
            boolean disabled;
            case CPACR.cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = el == EL0;
                when '10' disabled = ConstrainUnpredictableBool(UnpredictableBool);
                when '11' disabled = FALSE;
            if disabled then return FALSE;

        if el IN {EL0, EL1, EL2} && EL2Enabled() then
            if !ELUsingAArch32(EL2) then
                return AArch64.IsFPEEnabled(EL2);
            if HCPTR.TCP10 == '1' then return FALSE;

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then return FALSE;

        return TRUE;
    
```

## Library pseudocode for aarch64/functions/sve/AArch64.IsFPEEnabled

```

// AArch64.IsFPEEnabled()
// =====
// Returns TRUE if access to the SIMD&FP instructions or System registers
// enabled at the target exception level in AArch64 state and FALSE otherwise

boolean AArch64.IsFPEEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
}

```

## Library pseudocode for aarch64/functions/sve/ActivePredicateElement

```
// ActivePredicateElement()
// =====
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

boolean ActivePredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n> == '1';
```

## Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// =====
// Return TRUE if there is at least one active element in mask. Otherwise
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

## Library pseudocode for aarch64/functions/sve/BitDeposit

```
// BitDeposit()
// =====
// Deposit the least significant bits from DATA into result positions
// selected by nonzero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;
```

## Library pseudocode for aarch64/functions/sve/BitExtract

```
// BitExtract()
// =====
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

## Library pseudocode for aarch64/functions/sve/BitGroup

```
// BitGroup()
// =====
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

## Library pseudocode for aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// =====
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

## Library pseudocode for aarch64/functions/sve/ CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =====
// Checks for traps on SVE instructions that are not legal in streaming

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64
        SMEAccessTrap(SMEExceptionType\_Streaming, PSTATE.EL);
```

## Library pseudocode for aarch64/functions/sve/CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that access SW
```

```

// registers.

CheckOriginalSVEEnabled()
    assert IsFeatureImplemented(FEAT_SVE) ;
    boolean disabled;

    if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTR_EL3.TFP == '1') &&
        EL3SDDUndefPriority()) then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == ...
                when '11' disabled = FALSE;
                if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == ...
                when '11' disabled = FALSE;
                if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                SVEAccessTrap(EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);

```

## Library pseudocode for aarch64/functions/sve/CheckSMEAccess

```
// CheckSMEAccess()
// =====
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap,
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap,
```

## Library pseudocode for aarch64/functions/sve/CheckSMEAndZAEEnabled

```
// CheckSMEAndZAEEnabled()
// =====

CheckSMEAndZAEEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEEEnabled

```
// CheckSMEEEnabled()
// =====

CheckSMEEEnabled()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
```

```

        when '11' disabled = FALSE;
if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

// Check SIMD&FP at EL0/EL1
case CPACR_EL1.FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '11' disabled = FALSE;
if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
    if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
        // Check SME at EL2
        case CPTTR_EL2.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == 1;
            when '11' disabled = FALSE;
if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

        // Check SIMD&FP at EL2
        case CPTTR_EL2.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == 1;
            when '11' disabled = FALSE;
if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
else
    if CPTTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
    if CPTTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

// Check if access disabled in CPTTR_EL3
if HaveEL(EL3) then
    if CPTTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
    if CPTTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

```

### Library pseudocode for aarch64/functions/sve/CheckSMEZT0Enabled

```

// CheckSMEZT0Enabled()
// =====
// Checks for ZT0 enabled.

CheckSMEZT0Enabled()
    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEEExceptionType_InactiveZA, PSTATE.EL);

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        if SMCR_EL1.EZT0 == '0' then
            SMEAccessTrap(SMEEExceptionType_InaccessibleZT0, EL1);

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.EZT0 == '0' then
            SMEAccessTrap(SMEEExceptionType_InaccessibleZT0, EL2);

    // Check if all accesses to ZT0 are disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.EZT0 == '0' then
            SMEAccessTrap(SMEEExceptionType_InaccessibleZT0, EL3);

```

### **Library pseudocode for aarch64/functions/sve/CheckSVEEnabled**

```

// CheckSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        CheckSMEEEnabled();
    elseif IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_
        CheckStreamingSVEEnabled();
    else
        CheckOriginalSVEEnabled();

```

### **Library pseudocode for aarch64/functions/sve/ CheckStreamingSVEAndZAEEnabled**

```

// CheckStreamingSVEAndZAEEnabled()
// =====

CheckStreamingSVEAndZAEEnabled()
    CheckStreamingSVEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEEExceptionType_InactiveZA, PSTATE.EL);

```

### **Library pseudocode for aarch64/functions/sve/CheckStreamingSVEEnabled**

```

// CheckStreamingSVEEnabled()
// =====

```

```

CheckStreamingSVEEnabled()
    CheckSMEEnabled\(\);

    if PSTATE.SM == '0' then
        SMEAccessTrap\(SMEExceptionType\_NotStreaming, PSTATE.EL\);

```

### Library pseudocode for aarch64/functions/sve/CurrentNSVL

```

// CurrentNSVL - non-assignment form
// =====
// Non-Streaming VL

integer CurrentNSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost\(\)) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL\(EL3\) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);

    return vl;

```

### Library pseudocode for aarch64/functions/sve/CurrentSVL

```

// CurrentSVL - non-assignment form
// =====
// Streaming SVL

integer CurrentSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)) then
        vl = UInt(SMCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost\(\)) then
        vl = UInt(SMCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) then
        vl = Min(vl, UInt(SMCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elsif HaveEL\(EL3\) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));

    vl = (vl + 1) * 128;

```

```

    vl = ImplementedSMEVectorLength(vl);

    return vl;

```

### Library pseudocode for aarch64/functions/sve/CurrentVL

```

// CurrentVL - non-assignment form
// =====

integer CurrentVL
    return if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then

```

### Library pseudocode for aarch64/functions/sve/DecodePredCount

```

// DecodePredCount()
// =====

integer DecodePredCount(bits(5) bitpattern, integer esize)
    integer elements = CurrentVL DIV esize;
    integer numElem;
    case bitpattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise      numElem = 0;
    endcase
    return numElem;

```

### Library pseudocode for aarch64/functions/sve/ElemFFR

```

// ElemFFR[] - non-assignment form
// =====

bit ElemFFR[integer e, integer esize]
    return PredicateElement(_FFR, e, esize);

// ElemFFR[] - assignment form
// =====

ElemFFR[integer e, integer esize] = bit value
    constant integer psize = esize DIV 8;
    constant integer n = e * psize;

```

```

    assert n >= 0 && (n + psize) <= CurrentVL DIV 8;
    _FFR<(n+psize)-1:n> = ZeroExtend(value, psize);
    return;

```

### Library pseudocode for aarch64/functions/sve/FFR

```

// FFR[] - non-assignment form
// =====

bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =====

FFR[integer width] = bits(width) value
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(value, MAX_PL);
    else
        _FFR<width-1:0> = value;

```

### Library pseudocode for aarch64/functions/sve/FPCCompareNE

```

// FPCCompareNE()
// =====

boolean FPCCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType SNaN, FPType QNaN};
    op2_nan = type2 IN {FPType SNaN, FPType QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPType SNaN || type2 == FPType SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

### Library pseudocode for aarch64/functions/sve/FPCCompareUN

```

// FPCCompareUN()
// =====

boolean FPCCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

```

```

    if type1 == FPType_SNan || type2 == FPType_SNan then
        FPPprocessException(FPEExc_InvalidOp, fpcr);

    result = type1 IN {FPType_SNan, FPType_QNaN} || type2 IN {FPType_SNan, FPType_QNaN}
    if !result then
        FPPprocessDenorms(type1, type2, N, fpcr);

    return result;

```

### Library pseudocode for aarch64/functions/sve/FPConvertSVE

```

// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, FPRounding rounding,
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding, M);

// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, integer M)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);

```

### Library pseudocode for aarch64/functions/sve/FPExpA

```

// FPExpA()
// =====

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx, N];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;

```

### Library pseudocode for aarch64/functions/sve/FPExpCoefficient

```

// FPExpCoefficient()
// =====

bits(N) FPExpCoefficient[integer index, integer N]
    assert N IN {16,32,64};

```

```
integer result;

if N == 16 then
    case index of
        when 0 result = 0x0000;
        when 1 result = 0x0016;
        when 2 result = 0x002d;
        when 3 result = 0x0045;
        when 4 result = 0x005d;
        when 5 result = 0x0075;
        when 6 result = 0x008e;
        when 7 result = 0x00a8;
        when 8 result = 0x00c2;
        when 9 result = 0x00dc;
        when 10 result = 0x00f8;
        when 11 result = 0x0114;
        when 12 result = 0x0130;
        when 13 result = 0x014d;
        when 14 result = 0x016b;
        when 15 result = 0x0189;
        when 16 result = 0x01a8;
        when 17 result = 0x01c8;
        when 18 result = 0x01e8;
        when 19 result = 0x0209;
        when 20 result = 0x022b;
        when 21 result = 0x024e;
        when 22 result = 0x0271;
        when 23 result = 0x0295;
        when 24 result = 0x02ba;
        when 25 result = 0x02e0;
        when 26 result = 0x0306;
        when 27 result = 0x032e;
        when 28 result = 0x0356;
        when 29 result = 0x037f;
        when 30 result = 0x03a9;
        when 31 result = 0x03d4;

elsif N == 32 then
    case index of
        when 0 result = 0x0000000;
        when 1 result = 0x0164d2;
        when 2 result = 0x02cd87;
        when 3 result = 0x043a29;
        when 4 result = 0x05aac3;
        when 5 result = 0x071f62;
        when 6 result = 0x08980f;
        when 7 result = 0x0a14d5;
        when 8 result = 0x0b95c2;
        when 9 result = 0x0d1adf;
        when 10 result = 0x0ea43a;
        when 11 result = 0x1031dc;
        when 12 result = 0x11c3d3;
        when 13 result = 0x135a2b;
        when 14 result = 0x14f4f0;
        when 15 result = 0x16942d;
        when 16 result = 0x1837f0;
        when 17 result = 0x19e046;
        when 18 result = 0x1b8d3a;
        when 19 result = 0x1d3eda;
        when 20 result = 0x1ef532;
```

```
when 21 result = 0x20b051;
when 22 result = 0x227043;
when 23 result = 0x243516;
when 24 result = 0x25fed7;
when 25 result = 0x27cd94;
when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3123f6;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fbaf;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;

else // N == 64
case index of
    when 0 result = 0x0000000000000000;
    when 1 result = 0x02C9A3E778061;
    when 2 result = 0x059B0D3158574;
    when 3 result = 0x0874518759BC8;
    when 4 result = 0x0B5586CF9890F;
    when 5 result = 0x0E3EC32D3D1A2;
    when 6 result = 0x11301D0125B51;
    when 7 result = 0x1429AAEA92DE0;
    when 8 result = 0x172B83C7D517B;
    when 9 result = 0x1A35BEB6FCB75;
    when 10 result = 0x1D4873168B9AA;
    when 11 result = 0x2063B88628CD6;
    when 12 result = 0x2387A6E756238;
    when 13 result = 0x26B4565E27CDD;
```

```

when 14 result = 0x29E9DF51FDEE1;
when 15 result = 0x2D285A6E4030B;
when 16 result = 0x306FE0A31B715;
when 17 result = 0x33C08B26416FF;
when 18 result = 0x371A7373AA9CB;
when 19 result = 0x3A7DB34E59FF7;
when 20 result = 0x3DEA64C123422;
when 21 result = 0x4160A21F72E2A;
when 22 result = 0x44E086061892D;
when 23 result = 0x486A2B5C13CD0;
when 24 result = 0x4BFAD5362A27;
when 25 result = 0x4F9B2769D2CA7;
when 26 result = 0x5342B569D4F82;
when 27 result = 0x56F4736B527DA;
when 28 result = 0x5AB07DD485429;
when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FEB564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422AA0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B0CDC5E5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA5503B23E255D;
when 47 result = 0xA9E6B5579FDBF;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2FB5E47;
when 51 result = 0xBCC1E904BC1D2;
when 52 result = 0xC199BDD85529C;
when 53 result = 0xC67F12E57D14B;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD072D4A07897C;
when 56 result = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A490DA;
when 61 result = 0xEFA1BEE615A27;
when 62 result = 0xF50765B6E4540;
when 63 result = 0xFA7C1819E90D8;

return result<N-1:0>;

```

### Library pseudocode for aarch64/functions/sve/FPLogB

```

// FPLogB()
// ======

```

```

bits(N) FPLogB(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    integer result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN || fptype == FPType_NaN
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = -(2^(N-1)); // MinInt, 100..00
    elsif fptype == FPType_Infinity then
        result = 2^(N-1) - 1; // MaxInt, 011..11
    else
        // FPUnpack has already scaled a subnormal input
        value = Abs(value);
        result = 0;
        while value < 1.0 do
            value = value * 2.0;
            result = result - 1;
        while value >= 2.0 do
            value = value / 2.0;
            result = result + 1;

        FPProcessDenorm(fptype, N, fpcr);

    return result<N-1:0>;

```

### Library pseudocode for aarch64/functions/sve/FPMInNormal

```

// FPMInNormal()
// =====

bits(N) FPMInNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 16);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;

```

### Library pseudocode for aarch64/functions/sve/FPOne

```

// FPOne()
// =====

bits(N) FPOne(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 16);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;

```

### Library pseudocode for aarch64/functions/sve/FPPointFive

```

// FPPointFive()
// =====

```

```

bits(N) FPPointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 1);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;

```

### Library pseudocode for aarch64/functions/sve/FPScale

```

// FPScale()
// =====

bits(N) FPScale(bits(N) op, integer scale, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity then
        result = FPIinfinity(sign, N);
    else
        result = FPRound(value * (2.0^scale), fpcr, N);
        FPPprocessDenorm(fptype, N, fpcr);

    return result;

```

### Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```

// FPTrigMAdd()
// =====

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCRType
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff = FPTrigMAddCoefficient[x, N];
    op2 = FPAbs(op2);
    result = FPMulAdd(coeff, op1, op2, fpcr);
    return result;

```

### Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```

// FPTrigMAddCoefficient()
// =====

bits(N) FPTrigMAddCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when 0 result = 0x3c00;
            when 1 result = 0xb155;
            when 2 result = 0x2030;
            when 3 result = 0x0000;
            when 4 result = 0x0000;
            when 5 result = 0x0000;
            when 6 result = 0x0000;
            when 7 result = 0x0000;
            when 8 result = 0x3c00;
            when 9 result = 0xb800;
            when 10 result = 0x293a;
            when 11 result = 0x0000;
            when 12 result = 0x0000;
            when 13 result = 0x0000;
            when 14 result = 0x0000;
            when 15 result = 0x0000;
    elsif N == 32 then
        case index of
            when 0 result = 0x3f800000;
            when 1 result = 0xbe2aaaab;
            when 2 result = 0x3c088886;
            when 3 result = 0xb95008b9;
            when 4 result = 0x36369d6d;
            when 5 result = 0x00000000;
            when 6 result = 0x00000000;
            when 7 result = 0x00000000;
            when 8 result = 0x3f800000;
            when 9 result = 0xbf000000;
            when 10 result = 0x3d2aaaa6;
            when 11 result = 0xbab60705;
            when 12 result = 0x37cd37cc;
            when 13 result = 0x00000000;
            when 14 result = 0x00000000;
            when 15 result = 0x00000000;
    else // N == 64
        case index of
            when 0 result = 0xff00000000000000;
            when 1 result = 0xbfc555555555543;
            when 2 result = 0x3f8111111110f30c;
            when 3 result = 0xbfa01a019b92fc6;
            when 4 result = 0x3ec71de351f3d22b;
            when 5 result = 0xbe5ae5e2b60f7b91;
            when 6 result = 0x3de5d8408868552f;
            when 7 result = 0x0000000000000000;
            when 8 result = 0x3ff00000000000000;
            when 9 result = 0xbfe00000000000000;
            when 10 result = 0x3fa555555555536;
            when 11 result = 0xbfa56c16c16c13a0b;
            when 12 result = 0x3efa01a019b1e8d8;
            when 13 result = 0xbe927e4f7282f468;
            when 14 result = 0x3e21ee96d2641b13;

```

```

        when 15 result = 0xbda8f76380fbb401;
    return result<N-1:0>;

```

### Library pseudocode for aarch64/functions/sve/FPTrigSMul

```

// FPTrigSMul()
// =====

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (ftype, sign, value) = FPUnpack(result, fpcr, fpexc);

    if !(ftype IN {FPType\_QNaN, FPType\_SNaN}) then
        result<N-1> = op2<0>;

    return result;

```

### Library pseudocode for aarch64/functions/sve/FPTrigSSel

```

// FPTrigSSel()
// =====

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elseif op2<1> == '1' then
        result = FPNeg(op1);
    else
        result = op1;

    return result;

```

### Library pseudocode for aarch64/functions/sve/FirstActive

```

// FirstActive()
// =====

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';

```

### Library pseudocode for aarch64/functions/sve/FloorPow2

```

// FloorPow2()
// =====
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);

```

### **Library pseudocode for aarch64/functions/sve/HaveSMEFullA64**

```

// HaveSMEFullA64()
// =====
// Returns TRUE if the SME FA64 extension is implemented, FALSE otherwise.

boolean HaveSMEFullA64()
    return IsFeatureImplemented(FEAT_SME_FA64);

```

### **Library pseudocode for aarch64/functions/sve/HaveSVE**

```

// HaveSVE()
// =====

boolean HaveSVE()
    return IsFeatureImplemented(FEAT_SVE);

```

### **Library pseudocode for aarch64/functions/sve/HaveSVE2**

```

// HaveSVE2()
// =====
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.

boolean HaveSVE2()
    return IsFeatureImplemented(FEAT_SVE2);

```

### **Library pseudocode for aarch64/functions/sve/HaveSVE2AES**

```

// HaveSVE2AES()
// =====
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.

boolean HaveSVE2AES()
    return IsFeatureImplemented(FEAT_SVE_AES);

```

### **Library pseudocode for aarch64/functions/sve/HaveSVE2BitPerm**

```

// HaveSVE2BitPerm()
// =====

```

```
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise
boolean HaveSVE2BitPerm()
    return IsFeatureImplemented(FEAT_SVE_BitPerm);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2PMULL128

```
// HaveSVE2PMULL128()
// =====
// Returns TRUE if the SVE2 128 bit PMULL extension is implemented, FALSE otherwise
boolean HaveSVE2PMULL128()
    return IsFeatureImplemented(FEAT_SVE_PMULL128);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SHA256

```
// HaveSVE2SHA256()
// =====
// Returns TRUE if the SVE2 SHA256 extension is implemented, FALSE otherwise
boolean HaveSVE2SHA256()
    return (IsFeatureImplemented(FEAT_SVE2) &&
            boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA256 extension");
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SHA3

```
// HaveSVE2SHA3()
// =====
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise
boolean HaveSVE2SHA3()
    return IsFeatureImplemented(FEAT_SVE_SHA3);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SHA512

```
// HaveSVE2SHA512()
// =====
// Returns TRUE if the SVE2 SHA512 extension is implemented, FALSE otherwise
boolean HaveSVE2SHA512()
    return (IsFeatureImplemented(FEAT_SVE2) &&
            boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA512 extension");
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SM3

```
// HaveSVE2SM3()
// =====
// Returns TRUE if the SVE2 SM3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM3()
    return (IsFeatureImplemented(FEAT_SVE2) &&
            boolean IMPLEMENTATION_DEFINED "Have SVE2 SM3 extension");
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2SM4

```
// HaveSVE2SM4()
// =====
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM4()
    return IsFeatureImplemented(FEAT_SVE_SM4);
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2p1

```
// HaveSVE2p1()
// =====
// Returns TRUE if the SVE2.1 extension is implemented, FALSE otherwise.

boolean HaveSVE2p1()
    return IsFeatureImplemented(FEAT_SVE2p1);
```

### Library pseudocode for aarch64/functions/sve/HaveSVEB16B16

```
// HaveSVEB16B16()
// =====
// Returns TRUE if the SVE2.1 non-widening BFLOAT16 instructions are implemented,
// FALSE otherwise.

boolean HaveSVEB16B16()
    return IsFeatureImplemented(FEAT_SVE_B16B16);
```

### Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

```
// HaveSVEFP32MatMulExt()
// =====
// Returns TRUE if single-precision floating-point matrix multiply instructions
// are implemented, FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return IsFeatureImplemented(FEAT_F32MM);
```

### Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

```
// HaveSVEFP64MatMulExt()
// =====
// Returns TRUE if double-precision floating-point matrix multiply instructions
// are implemented, FALSE otherwise.
```

```

boolean HaveSVEFP64MatMulExt()
    return IsFeatureImplemented(FEAT_F64MM);

```

### Library pseudocode for aarch64/functions/sve/ImplementedSMEVectorLength

```

// ImplementedSMEVectorLength()
// =====
// Reduce SVE/SME vector length to a supported value (power of two)

integer ImplementedSMEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedSVL\(\);
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbis
    while nbis > 128 do
        if IsPow2(nbis) && SupportedPowerTwoSVL(nbis) then return nbis;
        nbis = nbis - 128;

    // Return the smallest supported power-of-two VL
    nbis = 128;
    while nbis < maxbits do
        if SupportedPowerTwoSVL(nbis) then return nbis;
        nbis = nbis * 2;

    // The only option is maxbits
    return maxbits;

```

### Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```

// ImplementedSVEVectorLength()
// =====
// Reduce SVE vector length to a supported value (power of two)

integer ImplementedSVEVectorLength(integer nbis_in)
    integer maxbits = MaxImplementedVL\(\);
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbis = Min(nbis_in, maxbits);
    assert 128 <= nbis && nbis <= 2048 && Align(nbis, 128) == nbis;

    while nbis > 128 do
        if IsPow2(nbis) then return nbis;
        nbis = nbis - 128;
    return nbis;

```

### Library pseudocode for aarch64/functions/sve/InStreamingMode

```

// InStreamingMode()
// =====

boolean InStreamingMode()
    return IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1';

```

## Library pseudocode for aarch64/functions/sve/IsEven

```
// IsEven()  
// ======  
  
boolean IsEven(integer val)  
    return val MOD 2 == 0;
```

## Library pseudocode for aarch64/functions/sve/IsFPEEnabled

```
// IsFPEEnabled()  
// =====  
// Returns TRUE if accesses to the Advanced SIMD and floating-point  
// registers are enabled at the target exception level in the current  
// execution state and FALSE otherwise.  
  
boolean IsFPEEnabled(bits(2) el)  
    if ELUsingAArch32(el) then  
        return AArch32.IsFPEEnabled(el);  
    else  
        return AArch64.IsFPEEnabled(el);
```

## Library pseudocode for aarch64/functions/sve/IsFullA64Enabled

```
// IsFullA64Enabled()  
// =====  
// Returns TRUE is full A64 is enabled in Streaming mode and FALSE otherwise.  
  
boolean IsFullA64Enabled()  
    if !HaveSMEFullA64() then return FALSE;  
  
    // Check if full SVE disabled in SMCR_EL1  
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then  
        // Check full SVE at EL0/EL1  
        if SMCR_EL1.FA64 == '0' then return FALSE;  
  
    // Check if full SVE disabled in SMCR_EL2  
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then  
        if SMCR_EL2.FA64 == '0' then return FALSE;  
  
    // Check if full SVE disabled in SMCR_EL3  
    if HaveEL(EL3) then  
        if SMCR_EL3.FA64 == '0' then return FALSE;  
  
    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/IsOdd

```
// IsOdd()  
// =====  
  
boolean IsOdd(integer val)  
    return val MOD 2 == 1;
```

## Library pseudocode for aarch64/functions/sve/IsOriginalSVEEnabled

```
// IsOriginalSVEEnabled()
// =====
// Returns TRUE if access to SVE functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsOriginalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
            if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
                if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/IsPow2

```
// IsPow2()
// =====
// Return TRUE if positive integer X is a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

## Library pseudocode for aarch64/functions/sve/IsSMEEEnabled

```
// IsSMEEEnabled()
// =====
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.
```

```

boolean IsSMEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if IsFeatureImplemented(FEAT_VHE) && HCR_EL2.E2H == '1' then
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;

    return TRUE;

```

### Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```

// IsSVEEnabled()
// =====
// Returns TRUE if access to SVE registers is enabled at the target exception
// level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elseif IsFeatureImplemented(FEAT_SVE) then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;

```

### Library pseudocode for aarch64/functions/sve/LastActive

```

// LastActive()
// =====

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';

```

## Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// =====

integer LastActiveElement(bits(N) mask, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then return e;
    return -1;
```

## Library pseudocode for aarch64/functions/sve/MaxImplementedSVL

```
// MaxImplementedSVL()
// =====

integer MaxImplementedSVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented SVL";
```

## Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// =====

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented VL";
```

## Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// =====

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) t
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEEnabled(EL2);
        else
            lower_enabled = IsFPEEnabled(EL1);
    elseif target_el == EL2 then
        assert EL2Enabled() && !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEEnabled(EL1);
        else
            lower_enabled = IsFPEEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEEnabled(EL0);
```

```

if lower_enabled then
    constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVLI
    constant integer PL = VL DIV 8;
    for n = 0 to 31
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _Z[n] = ZeroExtend(_Z[n]<VL-1:0>, MAX_VL);
    for n = 0 to 15
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
    if IsFeatureImplemented(FEAT_SME) && PSTATE.ZA == '1' then
        constant integer SVL = CurrentSVL;
        constant integer accessiblevecs = SVL DIV 8;
        constant integer allvecs = MaxImplementedSVL() DIV 8;

        for n = 0 to accessiblevecs - 1
            if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER)
                _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
        for n = accessiblevecs to allvecs - 1
            if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER)
                _ZA[n] = Zeros(MAX_VL);

```

## Library pseudocode for aarch64/functions/sve/MemNF

```

// MemNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccessDes
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    boolean bad;

boolean aligned = IsAligned(address, size);

if !aligned && AlignmentEnforced() then
    return (bits(8*size) UNKNOWN, TRUE);

boolean atomic = aligned || size == 1;

if !atomic then
    (value<7:0>, bad) = MemSingleNF[address, 1, accdesc, aligned];

    if bad then
        return (bits(8*size) UNKNOWN, TRUE);

    // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether
    // access will generate an Alignment Fault, as to get this far
    // not, so we must be changing to a new translation page.
    if !aligned then
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

    for i = 1 to size-1
        (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, accdesc];

        if bad then

```

```

        return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, accdesc, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);

```

## Library pseudocode for aarch64/functions/sve/MemSingleNF

```

// MemSingleNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccessDescriptor accdesc_in,
                                     boolean aligned]
    assert accdesc_in.acctype == AccessType\_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.

    bits(8*size) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;
    AccessDescriptor accdesc = accdesc_in;
    FaultRecord fault = NoFault(accdesc);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable\_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    // If the instruction encoding permits tag checking, confer with sys
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, s

    // Non-fault load from Device memory must not be performed external
    if memaddrdesc.memattrs.memtype == MemType\_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        boolean iswrite = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc)
            return (bits(8*size) UNKNOWN, TRUE);
        fault.merrorstate = memstatus.merrorstate;

```

```

        fault.extflag    = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);

    return (value, FALSE);

```

### Library pseudocode for aarch64/functions/sve/NoneActive

```

// NoneActive()
// =====

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) && ActivePredicateElement(x, e, esize)
            return '0';
    return '1';

```

### Library pseudocode for aarch64/functions/sve/P

```

// P[] - non-assignment form
// =====

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
        _P[n] = ZeroExtend(value, MAX\_PL);
    else
        _P[n]<width-1:0> = value;

```

### Library pseudocode for aarch64/functions/sve/PredTest

```

// PredTest()
// =====

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;

```

### Library pseudocode for aarch64/functions/sve/PredicateElement

```

// PredicateElement()
// =====
// Returns the predicate bit

bit PredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

```

### Library pseudocode for aarch64/functions/sve/ReducePredicated

```

// ReducePredicated()
// =====

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask,
    assert(N == M * 8);
    assert IsPow2(N);
    bits(N) operand;
    integer elements = N DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);

```

### Library pseudocode for aarch64/functions/sve/ResetSMEState

```

// ResetSMEState()
// =====

ResetSMEState()
    integer vectors = MAX\_VL DIV 8;
    for n = 0 to vectors - 1
        \_ZA[n] = Zeros(MAX\_VL);
    \_ZT0 = Zeros(ZT0\_LEN);

```

### Library pseudocode for aarch64/functions/sve/ResetSVEState

```

// ResetSVEState()
// =====

ResetSVEState()
    for n = 0 to 31
        \_Z[n] = Zeros(MAX\_VL);
    for n = 0 to 15
        \_P[n] = Zeros(MAX\_PL);
    \_FFR = Zeros(MAX\_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);

```

## Library pseudocode for aarch64/functions/sve/SMEAccessTrap

```
// SMEAccessTrap()
// =====
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_
SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2;
    route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled();
    except = ExceptionSyndrome(Exception_SMEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap
            except.syndrome<2:0> = '000';
        when SMEExceptionType_Streaming
            except.syndrome<2:0> = '001';
        when SMEExceptionType_NotStreaming
            except.syndrome<2:0> = '010';
        when SMEExceptionType_InactiveZA
            except.syndrome<2:0> = '011';
        when SMEExceptionType_InaccessibleZT0
            except.syndrome<2:0> = '100';

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return,
    else
        AArch64.TakeException(target_el, except, preferred_exception_re
```

## Library pseudocode for aarch64/functions/sve/SMEExceptionType

```
// SMEExceptionType
// =====
enumeration SMEExceptionType {
    SMEExceptionType_AccessTrap,           // SME functionality trapped or
    SMEExceptionType_Streaming,            // Illegal instruction in Stream
    SMEExceptionType_NotStreaming,         // Illegal instruction not in S
    SMEExceptionType_InactiveZA,          // Illegal instruction when ZA
    SMEExceptionType_InaccessibleZT0,     // Access to ZT0 is disabled
};
```

## Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// =====
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_
SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && Has
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '
```

```

except = ExceptionSyndrome(Exception_SVEAccessTrap);
bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

if route_to_el2 then
    AArch64.TakeException(EL2, except, preferred_exception_return,
else
    AArch64.TakeException(target_el, except, preferred_exception_re

```

## Library pseudocode for aarch64/functions/sve/SVECmp

```

// SVECmp
// =====

enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cm

```

## Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```

// SVEMoveMaskPreferred()
// =====
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffxy' or '0000000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if (imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> &&
            (IsZero(imm<15:7>) || IsOnes(imm<15:7>))) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<
            return FALSE;

    // Check for 16 bit immediates
else
    // Check for 'ffffffffffffxy00' or '000000000000xy00'
    if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
        return FALSE;

    // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
    if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<
        return FALSE;

```

```

    // Check for 'xy00xy00xy00xy00'
    if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
        return FALSE;

    return TRUE;

```

### Library pseudocode for aarch64/functions/sve/SetPSTATE\_SM

```

// SetPSTATE_SM()
// =====

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVESTate\(\);
        PSTATE.SM = value;

```

### Library pseudocode for aarch64/functions/sve/SetPSTATE\_SVCR

```

// SetPSTATE_SVCR
// =====

SetPSTATE_SVCR(bits(32) svcr)
    SetPSTATE\_SM(svcr<0>);
    SetPSTATE\_ZA(svcr<1>);

```

### Library pseudocode for aarch64/functions/sve/SetPSTATE\_ZA

```

// SetPSTATE_ZA()
// =====

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMESTate\(\);
        PSTATE.ZA = value;

```

### Library pseudocode for aarch64/functions/sve/ShiftSat

```

// ShiftSat()
// =====

integer ShiftSat(integer shift, integer esize)
    if shift > esize+1 then return esize+1;
    elseif shift < -(esize+1) then return -(esize+1);
    return shift;

```

### Library pseudocode for aarch64/functions/sve/SupportedPowerTwoSVL

```

// SupportedPowerTwoSVL()
// =====
// Return an IMPLEMENTATION DEFINED specific value

```

```
// returns TRUE if SVL is supported and is a power of two, FALSE otherwise
boolean SupportedPowerTwoSVL(integer nbits);
```

### Library pseudocode for aarch64/functions/sve/System

```
constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
constant integer ZTO_LEN = 512;
bits(MAX_PL) _FFR;

array bits(MAX_VL) _Z[0..31];

array bits(MAX_PL) _P[0..15];
```

### Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =====

bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====

Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<width-1:0> = value;
```

### Library pseudocode for aarch64/functions/syshintop/SystemHintOp

```
// SystemHintOp
// =====
// System Hint instruction types.

enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
```

```

        SystemHintOp_WFIT,
        SystemHintOp_CLRBHB,
        SystemHintOp_GCSB,
        SystemHintOp_CHKFEAT,
        SystemHintOp_CSDB
    };
}

```

## Library pseudocode for aarch64/functions/sysop/SysOp

```

// SysOp()
// =====

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;           // S1E1R
        when '000 0111 1000 001' return Sys_AT;           // S1E1W
        when '000 0111 1000 010' return Sys_AT;           // S1E0R
        when '000 0111 1000 011' return Sys_AT;           // S1E0W
        when '000 0111 1001 000' return Sys_AT;           // S1E1RP
        when '000 0111 1001 001' return Sys_AT;           // S1E1WP
        when '100 0111 1000 000' return Sys_AT;           // S1E2R
        when '100 0111 1000 001' return Sys_AT;           // S1E2W
        when '100 0111 1000 100' return Sys_AT;           // S12E1R
        when '100 0111 1000 101' return Sys_AT;           // S12E1W
        when '100 0111 1000 110' return Sys_AT;           // S12E0R
        when '100 0111 1000 111' return Sys_AT;           // S12E0W
        when '110 0111 1000 000' return Sys_AT;           // S1E3R
        when '110 0111 1000 001' return Sys_AT;           // S1E3W
        when '001 0111 0010 100' return Sys_BRB;          // IALL
        when '001 0111 0010 101' return Sys_BRB;          // INJ
        when '000 0111 0110 001' return Sys_DC;           // IVAC
        when '000 0111 0110 010' return Sys_DC;           // ISW
        when '000 0111 0110 011' return Sys_DC;           // IGVAC
        when '000 0111 0110 100' return Sys_DC;           // IGSW
        when '000 0111 0110 101' return Sys_DC;           // IGDVAC
        when '000 0111 0110 110' return Sys_DC;           // IGDSW
        when '000 0111 1010 010' return Sys_DC;           // CSW
        when '000 0111 1010 100' return Sys_DC;           // CGSW
        when '000 0111 1010 110' return Sys_DC;           // CGDSW
        when '000 0111 1110 010' return Sys_DC;           // CISW
        when '000 0111 1110 100' return Sys_DC;           // CIGSW
        when '000 0111 1110 110' return Sys_DC;           // CIGDSW
        when '011 0111 0100 001' return Sys_DC;           // ZVA
        when '011 0111 0100 011' return Sys_DC;           // GVA
        when '011 0111 0100 100' return Sys_DC;           // GZVA
        when '011 0111 1010 001' return Sys_DC;           // CVAC
        when '011 0111 1010 011' return Sys_DC;           // CGVAC
        when '011 0111 1010 101' return Sys_DC;           // CGDVAC
        when '011 0111 1011 001' return Sys_DC;           // CVAU
        when '011 0111 1100 001' return Sys_DC;           // CVAP
        when '011 0111 1100 011' return Sys_DC;           // CGVAP
        when '011 0111 1100 101' return Sys_DC;           // CGDVAP
        when '011 0111 1101 001' return Sys_DC;           // CVADP
        when '011 0111 1101 011' return Sys_DC;           // CGVADP
        when '011 0111 1101 101' return Sys_DC;           // CGDVADP
        when '011 0111 1110 001' return Sys_DC;           // CIVAC
        when '011 0111 1110 011' return Sys_DC;           // CIGVAC
        when '011 0111 1110 101' return Sys_DC;           // CIGDVAC

```

```

when '100 0111 1110 000' return Sys_DC; // CIPAE
when '100 0111 1110 111' return Sys_DC; // CIGDPAE
when '110 0111 1110 001' return Sys_DC; // CIPAPA
when '110 0111 1110 101' return Sys_DC; // CIGDPAPA
when '000 0111 0001 000' return Sys_IC; // IALLUIS
when '000 0111 0101 000' return Sys_IC; // IALLU
when '011 0111 0101 001' return Sys_IC; // IVAU
when '000 1000 0001 000' return Sys_TLBI; // VMALLE1OS
when '000 1000 0001 001' return Sys_TLBI; // VAE1OS
when '000 1000 0001 010' return Sys_TLBI; // ASIDE1OS
when '000 1000 0001 011' return Sys_TLBI; // VAAE1OS
when '000 1000 0001 101' return Sys_TLBI; // VALE1OS
when '000 1000 0001 111' return Sys_TLBI; // VAALE1OS
when '000 1000 0010 001' return Sys_TLBI; // RVAE1IS
when '000 1000 0010 011' return Sys_TLBI; // RVAE1IS
when '000 1000 0010 101' return Sys_TLBI; // RVALE1IS
when '000 1000 0010 111' return Sys_TLBI; // RVAALE1IS
when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
when '000 1000 0101 001' return Sys_TLBI; // RVAE1OS
when '000 1000 0101 011' return Sys_TLBI; // RVAE1OS
when '000 1000 0101 101' return Sys_TLBI; // RVALE1OS
when '000 1000 0101 111' return Sys_TLBI; // RVAALE1OS
when '000 1000 0110 001' return Sys_TLBI; // RVAE1
when '000 1000 0110 011' return Sys_TLBI; // RVAE1
when '000 1000 0110 101' return Sys_TLBI; // RVALE1
when '000 1000 0110 111' return Sys_TLBI; // RVAALE1
when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
when '000 1000 0111 001' return Sys_TLBI; // VAE1
when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
when '000 1000 0111 011' return Sys_TLBI; // VAAE1
when '000 1000 0111 101' return Sys_TLBI; // VALE1
when '000 1000 0111 111' return Sys_TLBI; // VAALE1
when '000 1001 0001 000' return Sys_TLBI; // VMALLE1OSNXS
when '000 1001 0001 001' return Sys_TLBI; // VAE1OSNXS
when '000 1001 0001 010' return Sys_TLBI; // ASIDE1OSNXS
when '000 1001 0001 011' return Sys_TLBI; // VAAE1OSNXS
when '000 1001 0001 101' return Sys_TLBI; // VALE1OSNXS
when '000 1001 0001 111' return Sys_TLBI; // VAALE1OSNXS
when '000 1001 0010 001' return Sys_TLBI; // RVAE1ISNXS
when '000 1001 0010 011' return Sys_TLBI; // RVAE1ISNXS
when '000 1001 0010 101' return Sys_TLBI; // RVALE1ISNXS
when '000 1001 0010 111' return Sys_TLBI; // RVAALE1ISNXS
when '000 1001 0011 000' return Sys_TLBI; // VMALLE1ISNXS
when '000 1001 0011 001' return Sys_TLBI; // VAE1ISNXS
when '000 1001 0011 010' return Sys_TLBI; // ASIDE1ISNXS
when '000 1001 0011 011' return Sys_TLBI; // VAAE1ISNXS
when '000 1001 0011 101' return Sys_TLBI; // VALE1ISNXS
when '000 1001 0011 111' return Sys_TLBI; // VAALE1ISNXS
when '000 1001 0101 001' return Sys_TLBI; // RVAE1OSNXS
when '000 1001 0101 011' return Sys_TLBI; // RVAE1OSNXS
when '000 1001 0101 101' return Sys_TLBI; // RVAE1OSNXS
when '000 1001 0101 111' return Sys_TLBI; // RVAE1OSNXS
when '000 1001 0110 001' return Sys_TLBI; // RVAE1NXS
when '000 1001 0110 011' return Sys_TLBI; // RVAE1NXS
when '000 1001 0110 101' return Sys_TLBI; // RVALE1NXS

```

```
when '000 1001 0110 111' return Sys_TLBi; // RVAALE1NXS
when '000 1001 0111 000' return Sys_TLBi; // VMALLE1NXS
when '000 1001 0111 001' return Sys_TLBi; // VAE1NXS
when '000 1001 0111 010' return Sys_TLBi; // ASIDE1NXS
when '000 1001 0111 011' return Sys_TLBi; // VAAE1NXS
when '000 1001 0111 101' return Sys_TLBi; // VALE1NXS
when '000 1001 0111 111' return Sys_TLBi; // VAALE1NXS
when '100 1000 0000 001' return Sys_TLBi; // IPAS2E1IS
when '100 1000 0000 010' return Sys_TLBi; // RIPAS2E1IS
when '100 1000 0000 101' return Sys_TLBi; // IPAS2LE1IS
when '100 1000 0000 110' return Sys_TLBi; // RIPAS2LE1IS
when '100 1000 0001 000' return Sys_TLBi; // ALLE2OS
when '100 1000 0001 001' return Sys_TLBi; // VAE2OS
when '100 1000 0001 100' return Sys_TLBi; // ALLE1OS
when '100 1000 0001 101' return Sys_TLBi; // VALE2OS
when '100 1000 0001 110' return Sys_TLBi; // VMALLS12E1OS
when '100 1000 0010 001' return Sys_TLBi; // RVAE2IS
when '100 1000 0010 101' return Sys_TLBi; // RVALE2IS
when '100 1000 0011 000' return Sys_TLBi; // ALLE2IS
when '100 1000 0011 001' return Sys_TLBi; // VAE2IS
when '100 1000 0011 100' return Sys_TLBi; // ALLE1IS
when '100 1000 0011 101' return Sys_TLBi; // VALE2IS
when '100 1000 0011 110' return Sys_TLBi; // VMALLS12E1IS
when '100 1000 0100 000' return Sys_TLBi; // IPAS2E1OS
when '100 1000 0100 001' return Sys_TLBi; // IPAS2E1
when '100 1000 0100 010' return Sys_TLBi; // RIPAS2E1
when '100 1000 0100 011' return Sys_TLBi; // RIPAS2E1OS
when '100 1000 0100 100' return Sys_TLBi; // IPAS2LE1OS
when '100 1000 0100 101' return Sys_TLBi; // IPAS2LE1
when '100 1000 0100 110' return Sys_TLBi; // RIPAS2LE1
when '100 1000 0100 111' return Sys_TLBi; // RIPAS2LE1OS
when '100 1000 0101 001' return Sys_TLBi; // RVAE2OS
when '100 1000 0101 101' return Sys_TLBi; // RVALE2OS
when '100 1000 0110 001' return Sys_TLBi; // RVAE2
when '100 1000 0110 101' return Sys_TLBi; // RVALE2
when '100 1000 0111 000' return Sys_TLBi; // ALLE2
when '100 1000 0111 001' return Sys_TLBi; // VAE2
when '100 1000 0111 100' return Sys_TLBi; // ALLE1
when '100 1000 0111 101' return Sys_TLBi; // VALE2
when '100 1000 0111 110' return Sys_TLBi; // VMALLS12E1
when '100 1001 0000 001' return Sys_TLBi; // IPAS2E1ISNXS
when '100 1001 0000 010' return Sys_TLBi; // RIPAS2E1ISNXS
when '100 1001 0000 101' return Sys_TLBi; // IPAS2LE1ISNXS
when '100 1001 0000 110' return Sys_TLBi; // RIPAS2LE1ISNXS
when '100 1001 0001 000' return Sys_TLBi; // ALLE2OSNXS
when '100 1001 0001 001' return Sys_TLBi; // VAE2OSNXS
when '100 1001 0001 100' return Sys_TLBi; // ALLE1OSNXS
when '100 1001 0001 101' return Sys_TLBi; // VALE2OSNXS
when '100 1001 0001 110' return Sys_TLBi; // VMALLS12E1OSNXS
when '100 1001 0010 001' return Sys_TLBi; // RVAE2ISNXS
when '100 1001 0010 101' return Sys_TLBi; // RVALE2ISNXS
when '100 1001 0011 000' return Sys_TLBi; // ALLE2ISNXS
when '100 1001 0011 001' return Sys_TLBi; // VAE2ISNXS
when '100 1001 0011 100' return Sys_TLBi; // ALLE1ISNXS
when '100 1001 0011 101' return Sys_TLBi; // VALE2ISNXS
when '100 1001 0011 110' return Sys_TLBi; // VMALLS12E1ISNXS
when '100 1001 0100 000' return Sys_TLBi; // IPAS2E1OSNXS
when '100 1001 0100 001' return Sys_TLBi; // IPAS2E1NXS
when '100 1001 0100 010' return Sys_TLBi; // RIPAS2E1NXS
when '100 1001 0100 011' return Sys_TLBi; // RIPAS2E1OSNXS
```

```

when '100 1001 0100 100' return Sys_TLBI; // IPAS2LE1OSNXS
when '100 1001 0100 101' return Sys_TLBI; // IPAS2LE1NXS
when '100 1001 0100 110' return Sys_TLBI; // RIPAS2LE1NXS
when '100 1001 0100 111' return Sys_TLBI; // RIPAS2LE1OSNXS
when '100 1001 0101 001' return Sys_TLBI; // RVAE2OSNXS
when '100 1001 0101 101' return Sys_TLBI; // RVALE2OSNXS
when '100 1001 0110 001' return Sys_TLBI; // RVAE2NXS
when '100 1001 0110 101' return Sys_TLBI; // RVALE2NXS
when '100 1001 0111 000' return Sys_TLBI; // ALLE2NXS
when '100 1001 0111 001' return Sys_TLBI; // VAE2NXS
when '100 1001 0111 100' return Sys_TLBI; // ALLE1NXS
when '100 1001 0111 101' return Sys_TLBI; // VALE2NXS
when '100 1001 0111 110' return Sys_TLBI; // VMALLS12E1NXS
when '110 1000 0001 000' return Sys_TLBI; // ALLE3OS
when '110 1000 0001 001' return Sys_TLBI; // VAE3OS
when '110 1000 0001 100' return Sys_TLBI; // PAALLOS
when '110 1000 0001 101' return Sys_TLBI; // VALE3OS
when '110 1000 0010 001' return Sys_TLBI; // RVAE3IS
when '110 1000 0010 101' return Sys_TLBI; // RVALE3IS
when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
when '110 1000 0100 011' return Sys_TLBI; // RPAOS
when '110 1000 0100 111' return Sys_TLBI; // RPALOS
when '110 1000 0101 001' return Sys_TLBI; // RVAE3OS
when '110 1000 0101 101' return Sys_TLBI; // RVALE3OS
when '110 1000 0110 001' return Sys_TLBI; // RVAE3
when '110 1000 0110 101' return Sys_TLBI; // RVALE3
when '110 1000 0111 000' return Sys_TLBI; // ALLE3
when '110 1000 0111 001' return Sys_TLBI; // VAE3
when '110 1000 0111 100' return Sys_TLBI; // PAALL
when '110 1000 0111 101' return Sys_TLBI; // VALE3
when '110 1001 0001 000' return Sys_TLBI; // ALLE3OSNXS
when '110 1001 0001 001' return Sys_TLBI; // VAE3OSNXS
when '110 1001 0001 101' return Sys_TLBI; // VALE3OSNXS
when '110 1001 0010 001' return Sys_TLBI; // RVAE3ISNXS
when '110 1001 0010 101' return Sys_TLBI; // RVALE3ISNXS
when '110 1001 0011 000' return Sys_TLBI; // ALLE3ISNXS
when '110 1001 0011 001' return Sys_TLBI; // VAE3ISNXS
when '110 1001 0011 101' return Sys_TLBI; // VALE3ISNXS
when '110 1001 0101 001' return Sys_TLBI; // RVAE3OSNXS
when '110 1001 0101 101' return Sys_TLBI; // RVALE3OSNXS
when '110 1001 0110 001' return Sys_TLBI; // RVAE3NXS
when '110 1001 0110 101' return Sys_TLBI; // RVALE3NXS
when '110 1001 0111 000' return Sys_TLBI; // ALLE3NXS
when '110 1001 0111 001' return Sys_TLBI; // VAE3NXS
when '110 1001 0111 101' return Sys_TLBI; // VALE3NXS
otherwise return Sys_SYS;

```

## Library pseudocode for aarch64/functions/sysop/SystemOp

```

// SystemOp
// ======
// System instruction types.

enumeration SystemOp {Sys_AT, Sys_BRB, Sys_DC, Sys_IC, Sys_TLBI, Sys_SY}

```

## Library pseudocode for aarch64/functions/sysop\_128/SysOp128

```
// SysOp128()
// =====

SystemOp128 SysOp128(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 1000 0001 001' return Sys TLBIP; // VAE1OS
        when '000 1000 0001 011' return Sys TLBIP; // VAAE1OS
        when '000 1000 0001 101' return Sys TLBIP; // VALE1OS
        when '000 1000 0001 111' return Sys TLBIP; // VAALE1OS
        when '000 1000 0011 001' return Sys TLBIP; // VAE1IS
        when '000 1000 0011 011' return Sys TLBIP; // VAAE1IS
        when '000 1000 0011 101' return Sys TLBIP; // VALE1IS
        when '000 1000 0011 111' return Sys TLBIP; // VAALE1IS
        when '000 1000 0111 001' return Sys TLBIP; // VAE1
        when '000 1000 0111 011' return Sys TLBIP; // VAAE1
        when '000 1000 0111 101' return Sys TLBIP; // VALE1
        when '000 1000 0111 111' return Sys TLBIP; // VAALE1
        when '000 1001 0001 001' return Sys TLBIP; // VAE1OSNXS
        when '000 1001 0001 011' return Sys TLBIP; // VAAE1OSNXS
        when '000 1001 0001 101' return Sys TLBIP; // VALE1OSNXS
        when '000 1001 0001 111' return Sys TLBIP; // VAALE1OSNXS
        when '000 1001 0011 001' return Sys TLBIP; // VAE1ISNXS
        when '000 1001 0011 011' return Sys TLBIP; // VAAE1ISNXS
        when '000 1001 0011 101' return Sys TLBIP; // VALE1ISNXS
        when '000 1001 0011 111' return Sys TLBIP; // VAALE1ISNXS
        when '000 1001 0111 001' return Sys TLBIP; // VAE1NXS
        when '000 1001 0111 011' return Sys TLBIP; // VAAE1NXS
        when '000 1001 0111 101' return Sys TLBIP; // VALE1NXS
        when '000 1001 0111 111' return Sys TLBIP; // VAALE1NXS
        when '100 1000 0001 001' return Sys TLBIP; // VAE2OS
        when '100 1000 0001 101' return Sys TLBIP; // VALE2OS
        when '100 1000 0011 001' return Sys TLBIP; // VAE2IS
        when '100 1000 0011 101' return Sys TLBIP; // VALE2IS
        when '100 1000 0111 001' return Sys TLBIP; // VAE2
        when '100 1000 0111 101' return Sys TLBIP; // VALE2
        when '100 1001 0001 001' return Sys TLBIP; // VAE2OSNXS
        when '100 1001 0001 101' return Sys TLBIP; // VALE2OSNXS
        when '100 1001 0011 001' return Sys TLBIP; // VAE2ISNXS
        when '100 1001 0011 101' return Sys TLBIP; // VALE2ISNXS
        when '100 1001 0111 001' return Sys TLBIP; // VAE2NXS
        when '100 1001 0111 101' return Sys TLBIP; // VALE2NXS
        when '110 1000 0001 001' return Sys TLBIP; // VAE3OS
        when '110 1000 0001 101' return Sys TLBIP; // VALE3OS
        when '110 1000 0011 001' return Sys TLBIP; // VAE3IS
        when '110 1000 0011 101' return Sys TLBIP; // VALE3IS
        when '110 1000 0111 001' return Sys TLBIP; // VAE3
        when '110 1000 0111 101' return Sys TLBIP; // VALE3
        when '110 1001 0001 001' return Sys TLBIP; // VAE3OSNXS
        when '110 1001 0001 101' return Sys TLBIP; // VALE3OSNXS
        when '110 1001 0011 001' return Sys TLBIP; // VAE3ISNXS
        when '110 1001 0011 101' return Sys TLBIP; // VALE3ISNXS
        when '110 1001 0111 001' return Sys TLBIP; // VAE3NXS
        when '110 1001 0111 101' return Sys TLBIP; // VALE3NXS
        when '100 1000 0000 001' return Sys TLBIP; // IPAS2E1IS
        when '100 1000 0000 101' return Sys TLBIP; // IPAS2LE1IS
        when '100 1000 0100 000' return Sys TLBIP; // IPAS2E1OS
        when '100 1000 0100 001' return Sys TLBIP; // IPAS2E1
```

```

when '100 1000 0100 100' return Sys_TLBIP; // IPAS2LE1OS
when '100 1000 0100 101' return Sys_TLBIP; // IPAS2LE1
when '100 1001 0000 001' return Sys_TLBIP; // IPAS2E1ISNXS
when '100 1001 0000 101' return Sys_TLBIP; // IPAS2LE1ISNXS
when '100 1001 0100 000' return Sys_TLBIP; // IPAS2E1OSNXS
when '100 1001 0100 001' return Sys_TLBIP; // IPAS2E1NXS
when '100 1001 0100 100' return Sys_TLBIP; // IPAS2LE1OSNXS
when '100 1001 0100 101' return Sys_TLBIP; // IPAS2LE1NXS
when '000 1000 0010 001' return Sys_TLBIP; // RVAE1IS
when '000 1000 0010 011' return Sys_TLBIP; // RVAE1IS
when '000 1000 0010 101' return Sys_TLBIP; // RVALE1IS
when '000 1000 0010 111' return Sys_TLBIP; // RVALE1IS
when '000 1000 0101 001' return Sys_TLBIP; // RVAE1OS
when '000 1000 0101 011' return Sys_TLBIP; // RVAE1OS
when '000 1000 0101 101' return Sys_TLBIP; // RVALE1OS
when '000 1000 0101 111' return Sys_TLBIP; // RVALE1OS
when '000 1000 0110 001' return Sys_TLBIP; // RVAE1
when '000 1000 0110 011' return Sys_TLBIP; // RVAE1
when '000 1000 0110 101' return Sys_TLBIP; // RVALE1
when '000 1000 0110 111' return Sys_TLBIP; // RVALE1
when '000 1001 0010 001' return Sys_TLBIP; // RVAE1ISNXS
when '000 1001 0010 011' return Sys_TLBIP; // RVAE1ISNXS
when '000 1001 0010 101' return Sys_TLBIP; // RVALE1ISNXS
when '000 1001 0010 111' return Sys_TLBIP; // RVALE1ISNXS
when '000 1001 0101 001' return Sys_TLBIP; // RVAE1OSNXS
when '000 1001 0101 011' return Sys_TLBIP; // RVAE1OSNXS
when '000 1001 0101 101' return Sys_TLBIP; // RVALE1OSNXS
when '000 1001 0101 111' return Sys_TLBIP; // RVALE1OSNXS
when '000 1001 0101 001' return Sys_TLBIP; // RVAE1OS
when '000 1001 0101 101' return Sys_TLBIP; // RVALE1OS
when '000 1001 0101 111' return Sys_TLBIP; // RVALE1OS
when '000 1001 0101 011' return Sys_TLBIP; // RVAE1
when '000 1001 0101 101' return Sys_TLBIP; // RVAE1
when '000 1001 0101 111' return Sys_TLBIP; // RVALE1
when '000 1001 0101 001' return Sys_TLBIP; // RVAE2IS
when '000 1000 0010 101' return Sys_TLBIP; // RVAE2IS
when '100 1000 0010 001' return Sys_TLBIP; // RVAE2IS
when '100 1000 0010 101' return Sys_TLBIP; // RVALE2IS
when '100 1000 0101 001' return Sys_TLBIP; // RVAE2OS
when '100 1000 0101 101' return Sys_TLBIP; // RVALE2OS
when '100 1000 0110 001' return Sys_TLBIP; // RVAE2
when '100 1000 0110 101' return Sys_TLBIP; // RVALE2
when '100 1001 0010 001' return Sys_TLBIP; // RVAE2ISNXS
when '100 1001 0010 101' return Sys_TLBIP; // RVALE2ISNXS
when '100 1001 0101 001' return Sys_TLBIP; // RVAE2OSNXS
when '100 1001 0101 101' return Sys_TLBIP; // RVALE2OSNXS
when '100 1001 0101 001' return Sys_TLBIP; // RVAE2NXS
when '100 1001 0110 001' return Sys_TLBIP; // RVALE2NXS
when '100 1001 0110 101' return Sys_TLBIP; // RVALE2NXS
when '110 1000 0010 001' return Sys_TLBIP; // RVAE3IS
when '110 1000 0010 101' return Sys_TLBIP; // RVALE3IS
when '110 1000 0101 001' return Sys_TLBIP; // RVAE3OS
when '110 1000 0101 101' return Sys_TLBIP; // RVALE3OS
when '110 1000 0110 001' return Sys_TLBIP; // RVAE3
when '110 1000 0110 101' return Sys_TLBIP; // RVALE3
when '110 1001 0010 001' return Sys_TLBIP; // RVAE3ISNXS
when '110 1001 0010 101' return Sys_TLBIP; // RVALE3ISNXS
when '110 1001 0101 001' return Sys_TLBIP; // RVAE3OSNXS
when '110 1001 0101 101' return Sys_TLBIP; // RVALE3OSNXS
when '110 1001 0110 001' return Sys_TLBIP; // RVAE3NXS
when '110 1001 0110 101' return Sys_TLBIP; // RVALE3NXS
when '100 1000 0000 010' return Sys_TLBIP; // RIPAS2E1IS
when '100 1000 0000 110' return Sys_TLBIP; // RIPAS2LE1IS
when '100 1000 0100 010' return Sys_TLBIP; // RIPAS2E1
when '100 1000 0100 011' return Sys_TLBIP; // RIPAS2E1OS

```

```

when '100 1000 0100 110' return Sys_TLBIP; // RIPAS2LE1
when '100 1000 0100 111' return Sys_TLBIP; // RIPAS2LE1OS
when '100 1001 0000 010' return Sys_TLBIP; // RIPAS2E1ISNXS
when '100 1001 0000 110' return Sys_TLBIP; // RIPAS2LE1ISNXS
when '100 1001 0100 010' return Sys_TLBIP; // RIPAS2E1NXS
when '100 1001 0100 011' return Sys_TLBIP; // RIPAS2E1OSNXS
when '100 1001 0100 110' return Sys_TLBIP; // RIPAS2LE1NXS
when '100 1001 0100 111' return Sys_TLBIP; // RIPAS2LE1OSNXS
otherwise return Sys_SYSP;

```

### Library pseudocode for aarch64/functions/sysop\_128/SystemOp128

```

// SystemOp128()
// =====
// System instruction types.

enumeration SystemOp128 {Sys_TLBIP, Sys_SYSP};

```

### Library pseudocode for aarch64/functions/sysregisters/ELR\_EL

```

// ELR_EL[] - non-assignment form
// =====

bits(64) ELR_EL[bits(2) el]
    bits(64) r;
    case el of
        when EL1 r = ELR_EL1;
        when EL2 r = ELR_EL2;
        when EL3 r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR_EL[] - assignment form
// =====

ELR_EL[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1 ELR_EL1 = r;
        when EL2 ELR_EL2 = r;
        when EL3 ELR_EL3 = r;
        otherwise Unreachable();
    return;

```

### Library pseudocode for aarch64/functions/sysregisters/ELR\_ELx

```

// ELR_ELx[] - non-assignment form
// =====

bits(64) ELR_ELx[]
    assert PSTATE.EL != EL0;
    return ELR_EL[PSTATE.EL];

// ELR_ELx[] - assignment form
// =====

```

```

ELR_ELx[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR_EL[PSTATE.EL] = value;
    return;

```

### Library pseudocode for aarch64/functions/sysregisters/ESRTYPE

```
type ESRTYPE;
```

### Library pseudocode for aarch64/functions/sysregisters/ESR\_EL

```

// ESR_EL[] - non-assignment form
// =====

ESRTYPE ESR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = ESR_EL1;
        when EL2    r = ESR_EL2;
        when EL3    r = ESR_EL3;
        otherwise   Unreachable();
    return r;

```

### Library pseudocode for aarch64/functions/sysregisters/ESR\_ELx

```

// ESR_ELx[] - non-assignment form
// =====

ESRTYPE ESR_ELx[]
    return ESR_EL[S1TranslationRegime()];

// ESR_ELx[] - assignment form
// =====

ESR_ELx[] = ESRTYPE value
    ESR_EL[S1TranslationRegime()] = value;

```

### Library pseudocode for aarch64/functions/sysregisters/ES\_EL

```

// ES_EL[] - assignment form
// =====

ESR_EL[bits(2) regime] = ESRTYPE value
    bits(64) r = value;
    case regime of
        when EL1    ESR_EL1 = r;
        when EL2    ESR_EL2 = r;
        when EL3    ESR_EL3 = r;
        otherwise   Unreachable();
    return;

```

## Library pseudocode for aarch64/functions/sysregisters/FAR\_EL

```
// FAR_EL[] - non-assignment form
// =====

bits(64) FAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = FAR_EL1;
        when EL2    r = FAR_EL2;
        when EL3    r = FAR_EL3;
        otherwise  Unreachable();
    return r;

// FAR_EL[] - assignment form
// =====

FAR_EL[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1    FAR_EL1 = r;
        when EL2    FAR_EL2 = r;
        when EL3    FAR_EL3 = r;
        otherwise  Unreachable();
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/FAR\_ELx

```
// FAR_ELx[] - non-assignment form
// =====

bits(64) FAR_ELx[]
    return FAR_EL[S1TranslationRegime()];

// FAR_ELx[] - assignment form
// =====

FAR_ELx[] = bits(64) value
    FAR_EL[S1TranslationRegime()] = value;
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/PFAR\_EL

```
// PFAR_EL[] - non-assignment form
// =====

bits(64) PFAR_EL[bits(2) regime]
    assert (IsFeatureImplemented(FEAT_PFAR) || (regime == EL3 && IsFeat
    bits(64) r;
    case regime of
        when EL1    r = PFAR_EL1;
        when EL2    r = PFAR_EL2;
        when EL3    r = MFAR_EL3;
        otherwise  Unreachable();
    return r;
```

```

// PFAR_EL[] - assignment form
// =====

PFAR_EL[bits(2) regime] = bits(64) value
    bits(64) r = value;
    assert (IsFeatureImplemented(FEAT_PFAR) || (IsFeatureImplemented(FEAT_MFAR) & IsFeatureImplemented(FEAT_SFAR)));
    case regime of
        when EL1 PFAR_EL1 = r;
        when EL2 PFAR_EL2 = r;
        when EL3 MFAR_EL3 = r;
        otherwise Unreachable();
    return;

```

### **Library pseudocode for aarch64/functions/sysregisters/PFAR\_ELx**

```

// PFAR_ELx[] - non-assignment form
// =====

bits(64) PFAR_ELx[]
    return PFAR_EL[S1TranslationRegime()];

// PFAR_ELx[] - assignment form
// =====

PFAR_ELx[] = bits(64) value
    PFAR_EL[S1TranslationRegime()] = value;
    return;

```

### **Library pseudocode for aarch64/functions/sysregisters/S1PIRTType**

```
type S1PIRTType;
```

### **Library pseudocode for aarch64/functions/sysregisters/S1PORTType**

```
type S1PORTType;
```

### **Library pseudocode for aarch64/functions/sysregisters/S2PIRTType**

```
type S2PIRTType;
```

### **Library pseudocode for aarch64/functions/sysregisters/S2PORTType**

```
type S2PORTType;
```

### **Library pseudocode for aarch64/functions/sysregisters/SCTLRTType**

```
type SCTLRTType;
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLR\_EL

```
// SCTLR_EL[] - non-assignment form
// =====

SCTLRTypE SCTLR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = SCTLR_EL1;
        when EL2 r = SCTLR_EL2;
        when EL3 r = SCTLR_EL3;
        otherwise Unreachable();
    return r;
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLR\_ELx

```
// SCTLR_ELx[] - non-assignment form
// =====

SCTLRTypE SCTLR_ELx[]
    return SCTLR_EL[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/sysregisters/VBAR\_EL

```
// VBAR_EL[] - non-assignment form
// =====

bits(64) VBAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = VBAR_EL1;
        when EL2 r = VBAR_EL2;
        when EL3 r = VBAR_EL3;
        otherwise Unreachable();
    return r;
```

## Library pseudocode for aarch64/functions/sysregisters/VBAR\_ELx

```
// VBAR_ELx[] - non-assignment form
// =====

bits(64) VBAR_ELx[]
    return VBAR_EL[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/system/ AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(bits(2) el)
```

```
if SCR_EL3.ATA == '0' && el IN {ELO, EL1, EL2} then
    return FALSE;
if HCR_EL2.ATA == '0' && el IN {ELO, EL1} && EL2Enabled() && HCR_EI
    return FALSE;

Regime regime = TranslationRegime(el);
case regime of
    when Regime_EL3 return SCTLR_EL3.ATA == '1';
    when Regime_EL2 return SCTLR_EL2.ATA == '1';
    when Regime_EL20 return if el == ELO then SCTLR_EL2.ATA0 == '1'
    when Regime_EL10 return if el == ELO then SCTLR_EL1.ATA0 == '1'
    otherwise Unreachable();
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckDAIFAccess

```
// AArch64.CheckDAIFAccess()
// =====
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

AArch64.CheckDAIFAccess(PSTATEField field)
    if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFGet}
        if IsInHost() || SCLTR_EL1.UMA == '0' then
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL1, 0x18);
```

## Library pseudocode for aarch64/functions/system/ AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// =====

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                           bits(4) crm, bits(3) op2, bits(5) rt, bit ready)
    if IsFeatureImplemented(FEAT_BTI) then
        BranchTargetCheck();
    if (IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 &&
        !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, ready))
        FailTransaction(TMFailure_ERR, FALSE);
    return;
```

## Library pseudocode for aarch64/functions/system/ AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// =====
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in,
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;
```

```

if IsOnes(exclude) then
    return '0000';

if offset == '0000' then
    while exclude<UInt(tag)> == '1' do
        tag = tag + '0001';

while offset != '0000' do
    offset = offset - '0001';
    tag = tag + '0001';
    while exclude<UInt(tag)> == '1' do
        tag = tag + '0001';

return tag;

```

### **Library pseudocode for aarch64/functions/system/ AArch64.ExecutingBROrBLROrRetInstr**

```

// AArch64.ExecutingBROrBLROrRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B] [Z], ...

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !IsFeatureImplemented(FEAT_BTI) then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;

```

### **Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr**

```

// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !IsFeatureImplemented(FEAT_BTI) then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' &&
        CRm  = instr<11:8>;
        op2  = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;

```

### **Library pseudocode for aarch64/functions/system/ AArch64.ExecutingERETInstr**

```

// AArch64.ExecutingERETInstr()
// =====

```

```
// Returns TRUE if current instruction is ERET.  
  
boolean AArch64.ExecutingERETInstr()  
    instr = ThisInstr\(\);  
    return instr<31:12> == '11010110100111110000';
```

### Library pseudocode for aarch64/functions/system/**AArch64.ImpDefSysInstr**

```
// AArch64.ImpDefSysInstr()  
// =====  
// Execute an implementation-defined system instruction with write (source)  
  
AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm,  
                        integer t);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.ImpDefSysInstr128**

```
// AArch64.ImpDefSysInstr128()  
// =====  
// Execute an implementation-defined system instruction with write (128-bit source)  
  
AArch64.ImpDefSysInstr128(integer el, bits(3) op1, bits(4) CRn,  
                           bits(4) CRm, bits(3) op2,  
                           integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.ImpDefSysInstrWithResult**

```
// AArch64.ImpDefSysInstrWithResult()  
// =====  
// Execute an implementation-defined system instruction with read (result)  
  
AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn,  
                                   bits(4) CRm, integer t);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.ImpDefSysRegRead**

```
// AArch64.ImpDefSysRegRead()  
// =====  
// Read from an implementation-defined System register and write the content  
// to X[t].  
  
AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm,  
                        integer t);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.ImpDefSysRegRead128**

```
// AArch64.ImpDefSysRegRead128()  
// =====  
// Read from an 128-bit implementation-defined System register
```

```
// and write the contents of the register to X[t], X[t+1].  
  
AArch64.ImpDefSysRegRead128(bits(2) op0, bits(3) op1, bits(4) CRn,  
                           bits(4) CRm, bits(3) op2,  
                           integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.ImpDefSysRegWrite**

```
// AArch64.ImpDefSysRegWrite()  
// =====  
// Write to an implementation-defined System register.  
  
AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4)  
                           integer t);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.ImpDefSysRegWrite128**

```
// AArch64.ImpDefSysRegWrite128()  
// =====  
// Write the contents of X[t], X[t+1] to an 128-bit implementation-defined  
// System register.  
  
AArch64.ImpDefSysRegWrite128(bits(2) op0, bits(3) op1, bits(4) CRn,  
                           bits(4) CRm, bits(3) op2,  
                           integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/ **AArch64.NextRandomTagBit**

```
// AArch64.NextRandomTagBit()  
// =====  
// Generate a random bit suitable for generating a random Allocation Tag.  
  
bit AArch64.NextRandomTagBit()  
    assert GCR_E11.RRND == '0';  
    bits(16) lfsr = RGSR_E11.SEED<15:0>;  
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;  
    RGSR_E11.SEED<15:0> = top:lfsr<15:1>;  
    return top;
```

### Library pseudocode for aarch64/functions/system/**AArch64.RandomTag**

```
// AArch64.RandomTag()  
// =====  
// Generate a random Allocation Tag.  
  
bits(4) AArch64.RandomTag()  
    bits(4) tag;  
    for i = 0 to 3  
        tag<i> = AArch64.NextRandomTagBit\(\);  
    return tag;
```

## **Library pseudocode for aarch64/functions/system/AArch64.SysInstr**

```
// AArch64.SysInstr()  
// =====  
// Execute a system instruction with write (source operand).  
  
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, in
```

## **Library pseudocode for aarch64/functions/system/ AArch64.SysInstrWithResult**

```
// AArch64.SysInstrWithResult()  
// =====  
// Execute a system instruction with read (result operand).  
// Writes the result of the instruction to X[t].  
  
AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer t);
```

## **Library pseudocode for aarch64/functions/system/AArch64.SysRegRead**

```
// AArch64.SysRegRead()  
// =====  
// Read from a System register and write the contents of the register to  
// memory.  
  
AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm,
```

## **Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite**

```
// AArch64.SysRegWrite()  
// =====  
// Write to a System register.  
  
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm,
```

## **Library pseudocode for aarch64/functions/system/BTypeCompatible**

```
boolean BTypeCompatible;
```

## **Library pseudocode for aarch64/functions/system/BTypeCompatible\_BTI**

```
// BTypeCompatible_BTI  
// =====  
// This function determines whether a given hint encoding is compatible  
// with PSTATE.BTTYPE. A value of TRUE here indicates a valid Branch Target  
// Instruction.  
  
boolean BTypeCompatible_BTI(bits(2) hintcode)  
    case hintcode of  
        when '00'  
            return FALSE;
```

```

        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;

```

### **Library pseudocode for aarch64/functions/system/BTypeCompatible\_PACIXSP**

```

// BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == EL0 then 35 else 36;
        return SCLTR_ELx[]<index> == '0';
    else
        return FALSE;

```

### **Library pseudocode for aarch64/functions/system/BTypeNext**

```
bits(2) BTypeNext;
```

### **Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag**

```

// ChooseRandomNonExcludedTag()
// =====
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND ==
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], exclude
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Alloc
//
// This function is permitted to generate a non-deterministic selection
// Allocation Tags. A reasonable implementation is described by the Pse
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of Ne
// Implementations may choose to behave the same as GCR_EL1.RRND=0.
//
// This function can read RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENT
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a p
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the random
// impacted if RGSR_EL1.SEED[15:0] is set to zero.

bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);

```

### **Library pseudocode for aarch64/functions/system/InGuardedPage**

```
boolean InGuardedPage;
```

## Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// =====
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    if !IsFeatureImplemented(FEAT_HCX) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled\(\);
```

## Library pseudocode for aarch64/functions/system/IsSCTRLR2EL1Enabled

```
// IsSCTRLR2EL1Enabled()
// =====
// Returns TRUE if access to SCTRLR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of SCTRLR2_EL1 returns 0 when access is not enabled.

boolean IsSCTRLR2EL1Enabled()
    if !IsFeatureImplemented(FEAT_SCTRLR2) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.SCTRLR2En == '0' then
        return FALSE;
    elseif (EL2Enabled\(\) && (!IsHCRXEL2Enabled\(\) || HCRX_EL2.SCTRLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

## Library pseudocode for aarch64/functions/system/IsSCTRLR2EL2Enabled

```
// IsSCTRLR2EL2Enabled()
// =====
// Returns TRUE if access to SCTRLR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of SCTRLR2_EL2 returns 0 when access is not enabled.

boolean IsSCTRLR2EL2Enabled()
    if !IsFeatureImplemented(FEAT_SCTRLR2) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.SCTRLR2En == '0' then
        return FALSE;

    return EL2Enabled\(\);
```

## Library pseudocode for aarch64/functions/system/IsTCR2EL1Enabled

```
// IsTCR2EL1Enabled()
// =====
// Returns TRUE if access to TCR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL1 returns 0 when access is not enabled.

boolean IsTCR2EL1Enabled()
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE;
```

```

    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;
    elseif (EL2Enabled()) && (!IsHCRXEL2Enabled()) || HCRX_EL2.TCR2En == '0'
        return FALSE;
    else
        return TRUE;

```

### **Library pseudocode for aarch64/functions/system/IsTCR2EL2Enabled**

```

// IsTCR2EL2Enabled()
// =====
// Returns TRUE if access to TCR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL2 returns 0 when access is not enabled.

boolean IsTCR2EL2Enabled()
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;

    return EL2Enabled();

```

### **Library pseudocode for aarch64/functions/system/SetBTypeCompatible**

```

// SetBTypeCompatible()
// =====
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;

```

### **Library pseudocode for aarch64/functions/system/SetBTypeNext**

```

// SetBTypeNext()
// =====
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;

```

### **Library pseudocode for aarch64/functions/system/SetInGuardedPage**

```

// SetInGuardedPage()
// =====
// Global state updated to denote if memory access is from a guarded page

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;

```

### **Library pseudocode for aarch64/functions/system128/AArch64.SysInstr128**

```

// AArch64.SysInstr128()
// =====
// Execute a system instruction with write (2 64-bit source operands).

AArch64.SysInstr128(integer op0, integer op1, integer crn, integer crm,
                     integer op2, integer t, integer t2);

```

**Library pseudocode for aarch64/functions/system128/  
AArch64.SysRegRead128**

```

// AArch64.SysRegRead128()
// =====
// Read from a 128-bit System register and write the contents of the re

AArch64.SysRegRead128(integer op0, integer op1, integer crn, integer cr
                      integer op2, integer t, integer t2);

```

**Library pseudocode for aarch64/functions/system128/  
AArch64.SysRegWrite128**

```

// AArch64.SysRegWrite128()
// =====
// Read the contents of X[t] and X[t2] and write the contents to a 128-bit

AArch64.SysRegWrite128(integer op0, integer op1, integer crn, integer cr
                      integer op2, integer t, integer t2);

```

**Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_IPAS2**

```

// AArch64.TLBIP_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated share
// domain matching the indicated VMID in the indicated regime with the
// Note: stage 1 and stage 2 combined entries are not in the scope of t
// IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_IPAS2(SecurityState security, Regime regime, bits(16) vmi
                     Shareability shareability, TLBILevel level, TLBIMem
                     assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
r.op          = TLBIOp\_IPAS2;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = level;
r.attr         = attr;
r.ttl         = Xt<47:44>;
r.address     = ZeroExtend(Xt<107:64> : Zeros(12), 64);
r.d64          = r.ttl IN {'00xx'};
r.d128         = TRUE;

case security of
  when SS\_NonSecure

```

```

        r.ipaspace = PAS_NonSecure;
when SS_Secure
        r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
when SS_Realm
        r.ipaspace = PAS_Realm;
otherwise
        // Root security state does not have stage 2 translation
        Unreachable();
TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_RIPAS2

```

// AArch64.TLBIP_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated region
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, TLBI_Level level, TLBI_MemoryRegion region,
                     assert PSTATE.EL IN {EL3, EL2, EL1} ;

TLBIRecord r;
r.op          = TLBIOp_RIPAS2;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = level;
r.attr         = attr;
r.ttl<1:0>    = Xt<38:37>;
r.d64          = r.ttl<1:0> == '00';
r.d128         = TRUE;

bits(2) tg      = Xt<47:46>;
integer scale   = UInt(Xt<45:44>);
integer num     = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

boolean valid;
(valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

if !valid then return;

case security of
    when SS_NonSecure
        r.ipaspace = PAS_NonSecure;
    when SS_Secure
        r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
    when SS_Realm
        r.ipaspace = PAS_Realm;
otherwise

```

```

    // Root security state does not have stage 2 translation
    Unreachable();

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r)
    return;

```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_RVA

```

// AArch64.TLBIP_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regi
// supports VMID, ASID) in the indicated regime with the indicated secu
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBIP_RVA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp\_RVA;
r.from_aarch64 = TRUE;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = level;
r.attr         = attr;
r.asid         = Xt<63:48>;
r.ttl<1:0>      = Xt<38:37>;
r.d64          = r.ttl<1:0> == '00';
r.d128         = TRUE;

boolean valid;
(valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r)
return;

```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_RVAA

```

// AArch64.TLBIP_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupport
// and all ASID in the indicated regime with the indicated security sta
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBIP_RVAA(SecurityState security, Regime regime, bits(16) vmi
                  Shareability shareability, TLBILevel level, TLBIMemA
assert PSTATE.EL IN {EL3, EL2, EL1};

```

```

TLBIRRecord r;
r.op          = TLBIOp_RVAA;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level        = level;
r.attr         = attr;
r.ttl<1:0>     = Xt<38:37>;
r.d64          = r.ttl<1:0> == '00';
r.d128         = TRUE;

bits(2) tg      = Xt<47:46>;
integer scale   = UInt(Xt<45:44>);
integer num     = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

boolean valid;

(valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

### **Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_VA**

```

// AArch64.TLBIP_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability
// matching the indicated VMID and ASID (where regime supports VMID, ASID)
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBIP_VA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIlevel level, TLBIMemAt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_VA;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = level;
r.attr         = attr;
r.asid         = Xt<63:48>;
r.ttl          = Xt<47:44>;
r.address     = ZeroExtend(Xt<107:64> : Zeros(12), 64);
r.d64          = r.ttl IN {'00xx'};
r.d128         = TRUE;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP\_VAA

```

// AArch64.TLBIP_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability
// matching the indicated VMID (where regime supports VMID) and all ASIDs
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBIP_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIlevel level, TLBIMemAt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_VAA;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = level;
r.attr         = attr;
r.ttl          = Xt<47:44>;
r.address     = ZeroExtend(Xt<107:64> : Zeros(12), 64);
r.d64          = r.ttl IN {'00xx'};
r.d128         = TRUE;

TLBI(r);

```

```
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;
```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_ALL

```
// AArch64.TLBI_ALL()
// =====
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// Invalidations applies to all applicable stage 1 and stage 2 entries.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security    = security;
    r.regime      = regime;
    r.level       = TLBILevel_Any;
    r.attr        = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;
```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_ASID

```
// AArch64.TLBI_ASID()
// =====
// Invalidate all stage 1 entries matching the indicated VMID (where relevant)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBIMemAttr attr, bits(64) asid)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp_ASID;
    r.from_aarch64 = TRUE;
    r.security    = security;
    r.regime      = regime;
    r.vmid        = vmid;
    r.level       = TLBILevel_Any;
    r.attr        = attr;
    r.asid        = Xt<63:48>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;
```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_IPAS2

```

// AArch64.TLB1_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability domain matching the indicated VMID in the indicated regime with the specified security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLB1_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBILevel level, TLBIMemAddress address,
                    assert PSTATE.EL IN {EL3, EL2};

        TLBIRecord r;
        r.op          = TLBIOp_IPAS2;
        r.from_aarch64 = TRUE;
        r.security     = security;
        r.regime       = regime;
        r.vmid         = vmid;
        r.level         = level;
        r.attr          = attr;
        r.ttl           = Xt<47:44>;
        r.address       = ZeroExtend(Xt<39:0> : Zeros(12), 64);
        r.d64           = TRUE;
        r.d128          = r.ttl IN {'00xx'};

        case security of
            when SS_NonSecure
                r.ipaspace = PAS_NonSecure;
            when SS_Secure
                r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
            when SS_Realm
                r.ipaspace = PAS_Realm;
            otherwise
                // Root security state does not have stage 2 translation
                Unreachable();
        end;

        TLBI(r);
        if shareability != Shareability_NSH then Broadcast(shareability, r);
        return;
    
```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLB1\_PAALL

```

// AArch64.TLB1_PAALL()
// =====
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated shareability domain.
// The invalidation applies to all TLB entries containing GPT information.

AArch64.TLB1_PAALL(Shareability shareability)
                    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

        TLBIRecord r;

        // r.security and r.regime do not apply for TLBI by PA operations
        r.op      = TLBIOp_PAALL;
        r.level   = TLBILevel_Any;
        r.attr    = TLBI_AllAttr;
    
```

```

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RIPAS2

```

// AArch64.TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated reg
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of t
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmi
                      Shareability shareability, TLBILevel level, TLBIMem
                      assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security    = security;
    r.regime      = regime;
    r.vmid        = vmid;
    r.level       = level;
    r.attr         = attr;
    r.ttl<1:0>    = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    bits(2) tg      = Xt<47:46>;
    integer scale   = UInt(Xt<45:44>);
    integer num     = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_S
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RPA

```
// AArch64.TLBI_RPA()
// =====
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareability domain.
// The invalidation applies to TLB entries containing GPT information
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries containing GPT info
//                     from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT info
//                      from the last level of the GPT walk

AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Shareability shareability
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

TLBIRecord r;
integer range_bits;
integer p;

// r.security and r.regime do not apply for TLBI by PA operations
r.op    = TLBIOp\_RPA;
r.level = level;
r.attr  = TLBI\_AllAttr;

// SIZE field
case Xt<47:44> of
    when '0000' range_bits = 12; // 4KB
    when '0001' range_bits = 14; // 16KB
    when '0010' range_bits = 16; // 64KB
    when '0011' range_bits = 21; // 2MB
    when '0100' range_bits = 25; // 32MB
    when '0101' range_bits = 29; // 512MB
    when '0110' range_bits = 30; // 1GB
    when '0111' range_bits = 34; // 16GB
    when '1000' range_bits = 36; // 64GB
    when '1001' range_bits = 39; // 512GB
    otherwise range_bits = 0; // Reserved encoding

// If SIZE selects a range smaller than PGS, then PGS is used instead
case DecodePGS(GPCCR_EL3.PGS) of
    when PGS\_4KB p = 12;
    when PGS\_16KB p = 14;
    when PGS\_64KB p = 16;

if range_bits < p then
    range_bits = p;

bits(52) BaseADDR = Zeros(52);
case GPCCR_EL3.PGS of
    when '00' BaseADDR<51:12> = Xt<39:0>; // 4KB
    when '10' BaseADDR<51:14> = Xt<39:2>; // 16KB
    when '01' BaseADDR<51:16> = Xt<39:4>; // 64KB

// The calculation here automatically aligns BaseADDR to the size of
// the region specified in SIZE. However, the architecture does not
// require this alignment and if BaseADDR is not aligned to the required
```

```

// specified by SIZE then no entries are required to be invalidated.
bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_bits),
bits(52) end_addr   = start_addr + ZeroExtend(Ones(range_bits), 52)

// PASpace is not considered in TLBI by PA operations
r.address      = ZeroExtend(start_addr, 64);
r.end_address  = ZeroExtend(end_addr, 64);

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)

```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RVA

```

// AArch64.TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regi
// supports VMID, ASID) in the indicated regime with the indicated secu
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAtt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_RVA;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime      = regime;
r.vmid        = vmid;
r.level       = level;
r.attr         = attr;
r.asid         = Xt<63:48>;
r.ttl<1:0>     = Xt<38:37>;
r.d64          = TRUE;
r.d128         = r.ttl<1:0> == '00';

boolean valid;
(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_RVAA

```

// AArch64.TLBI_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupport
// and all ASID in the indicated regime with the indicated security sta
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

```

```

AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBILevel level, TLBIMemAt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_RVAA;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime     = regime;
r.vmid       = vmid;
r.level      = level;
r.attr        = attr;
r.ttl<1:0>   = Xt<38:37>;
r.d64         = TRUE;
r.d128        = r.ttl<1:0> == '00';

bits(2) tg      = Xt<47:46>;
integer scale   = UInt(Xt<45:44>);
integer num     = UInt(Xt<43:39>);
integer baseaddr = SInt(Xt<36:0>);

boolean valid;

(valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

if !valid then return;

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VA

```

// AArch64.TLBI_VA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability
// matching the indicated VMID and ASID (where regime supports VMID, ASID)
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAt
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_VA;
r.from_aarch64 = TRUE;
r.security    = security;
r.regime     = regime;
r.vmid       = vmid;
r.level      = level;
r.attr        = attr;
r.asid        = Xt<63:48>;
r.ttl         = Xt<47:44>;
r.address     = ZeroExtend(Xt<43:0> : Zeros(12), 64);
r.d64         = TRUE;

```

```

    r.d128          = r.ttl IN {'00xx'};

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VAA

```

// AArch64.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability
// matching the indicated VMID (where regime supports VMID) and all ASI
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_VAA;
r.from_aarch64 = TRUE;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level         = level;
r.attr          = attr;
r.ttl          = Xt<47:44>;
r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);
r.d64          = TRUE;
r.d128         = r.ttl IN {'00xx'};

TLBI(r);
if shareability != Shareability_NSH then Broadcast(shareability, r)
return;

```

### Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VMALL

```

// AArch64.TLBI_VMALL()
// =====
// Invalidate all stage 1 entries for the indicated translation regime
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this
// Note: stage 2 only entries are not in the scope of this operation.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, TLBIMemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op          = TLBIOp_VMALL;
r.from_aarch64 = TRUE;
r.security     = security;
r.regime       = regime;
r.level         = TLBILevel_Any;

```

```

    r.vmid          = vmid;
    r.attr          = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

## Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI\_VMALLS12

```

// AArch64.TLBI_VMALLS12()
// =====
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op          = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security    = security;
    r.regime      = regime;
    r.level       = TLBILevel_Any;
    r.vmid        = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r)
    return;

```

## Library pseudocode for aarch64/functions/tlbi/ASID\_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

## Library pseudocode for aarch64/functions/tlbi/Broadcast

```

// Broadcast()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;

```

## Library pseudocode for aarch64/functions/tlbi/DecodeTLBITG

```

// DecodeTLBITG()
// =====
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)

```

```

        case tg of
            when '01'    return TGx_4KB;
            when '10'    return TGx_16KB;
            when '11'    return TGx_64KB;

```

### Library pseudocode for aarch64/functions/tlbi/GPTTLBIMatch

```

// GPTTLBIMatch()
// =====
// Determine whether the GPT TLB entry lies within the scope of invalidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry gpt_entry)
    assert tlbi.op IN {TLBIOp_RPA, TLBIOp_PAALL};

    boolean match;
    bits(64) entry_size_mask      = ZeroExtend(Ones(gpt_entry.size), 64);
    bits(64) entry_end_address   = ZeroExtend(gpt_entry.pa<55:0> OR entry_start_address);
    bits(64) entry_start_address = ZeroExtend(gpt_entry.pa<55:0> AND NOT entry_end_address);

    case tlbi.op of
        when TLBIOp_RPA
            match = (UInt(tlbi.address<55:0>) <= UInt(entry_end_address) && UInt(tlbi.end_address<55:0>) > UInt(entry_start_address)) || (tlbi.level == TLBIlevel_Any) || gpt_entry.level == TLBIlevel_Any;
        when TLBIOp_PAALL
            match = TRUE;

    return match;

```

### Library pseudocode for aarch64/functions/tlbi/HasLargeAddress

```

// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise

boolean HasLargeAddress(Regime regime)
    if !IsFeatureImplemented(FEAT_LPA2) then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL20
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();

```

### Library pseudocode for aarch64/functions/tlbi/ResTLBIRTTL

```

// ResTLBIRTTL()
// =====
// Determine whether the TTL field in TLBI instructions that do apply

```

```

// to a range of addresses contains a reserved value

boolean ResTLBIRTL(bits(2) tg, bits(2) ttl)
    case ttl of
        when '00' return TRUE;
        when '01' return DecodeTLBITG(tg) == TGx_16KB && !IsFeatureImplemented(FEAT_LPA2);
        otherwise return FALSE;

```

### Library pseudocode for aarch64/functions/tlbi/ResTLBITTL

```

// ResTLBITTL()
// =====
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

boolean ResTLBITTL(bits(4) ttl)
    case ttl of
        when '00xx' return TRUE;
        when '0100' return !IsFeatureImplemented(FEAT_LPA2);
        when '1000' return TRUE;
        when '1001' return !IsFeatureImplemented(FEAT_LPA2);
        when '1100' return TRUE;
        otherwise return FALSE;

```

### Library pseudocode for aarch64/functions/tlbi/TGBits

```

// TGBits()
// =====
// Return the number of bits required for a Tag Granule.

integer TGBits(bits(2) tg)
    case tg of
        when '01' return 12; // 4KB
        when '10' return 14; // 16KB
        when '11' return 16; // 64KB
        otherwise
            Unreachable();

```

### Library pseudocode for aarch64/functions/tlbi/TLBI

```

// TLBI()
// =====
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;

```

### Library pseudocode for aarch64/functions/tlbi/TLBILevel

```

// TLBILevel
// =====
enumeration TLBILevel {

```

```

    TLBILevel_Any,           // this applies to TLB entries at all levels
    TLBILevel_Last          // this applies to TLB entries at last level
} ;

```

## Library pseudocode for aarch64/functions/tlbi/TLBIMatch

```

// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord tlb_entry)
    boolean match;
    bits(64) entry_block_mask      = ZeroExtend(Ones(tlb_entry.blocksize));
    bits(64) entry_end_address    = tlb_entry.context.ia OR entry_block;
    bits(64) entry_start_address = tlb_entry.context.ia AND NOT entry_b;
    case tlbi.op of
        when TLBIOp\_DALL, TLBIOp\_IALL
            match = (tlbi.security == tlb_entry.context.ss &&
                      tlbi.regime == tlb_entry.context.regime);
        when TLBIOp\_DASID, TLBIOp\_IASID
            match = (tlb_entry.context.includes_s1 &&
                      tlbi.security == tlb_entry.context.ss &&
                      tlbi.regime == tlb_entry.context.regime &&
                      (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
                      (UseASID(tlb_entry.context) && tlb_entry.context.r
                          tlbi.asid == tlb_entry.context.asid));
        when TLBIOp\_DVA, TLBIOp\_IVA
            boolean regime_match;
            boolean context_match;
            boolean address_match;
            boolean level_match;
            regime_match = (tlb_entry.context.includes_s1 &&
                            tlbi.security == tlb_entry.context.ss &&
                            tlbi.regime == tlb_entry.context.regime);
            context_match = ((!UseVMID(tlb_entry.context) || tlbi.vmid
                            (!UseASID(tlb_entry.context) || tlbi.asid
                                tlb_entry.context.nG == '0'));
            constant integer addr_lsb = tlb_entry.blocksize;
            address_match = tlbi.address<55:addr_lsb> == tlb_entry.conte
            level_match = (tlbi.level == TLBILevel\_Any || !tlb_entry.wa
            match = regime_match && context_match && address_match &&
        when TLBIOp\_ALL
            relax_regime = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime\_EL20, Regime\_EL2} &&
                            tlb_entry.context.regime IN {Regime\_EL20, Regime\_EL2});
            match = (tlbi.security == tlb_entry.context.ss &&
                      (tlbi.regime == tlb_entry.context.regime || relax_
        when TLBIOp\_ASID
            match = (tlb_entry.context.includes_s1 &&
                      tlbi.security == tlb_entry.context.ss &&
                      tlbi.regime == tlb_entry.context.regime &&
                      (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
                      (UseASID(tlb_entry.context) && tlb_entry.context.r
                          tlbi.asid == tlb_entry.context.asid));
        when TLBIOp\_IPAS2, TLBIOp\_IPAS2
            constant integer addr_lsb = tlb_entry.blocksize;
            match = (!tlb_entry.context.includes_s1 && tlb_entry.conte
                      tlbi.security == tlb_entry.context.ss &&

```

```

        tlbi.regime == tlb_entry.context.regime &&
        (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
        tlbi.ipaspace == tlb_entry.context.ipaspace &&
        tlbi.address<55:addr_lsb> == tlb_entry.context.ia<
        (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
            DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.conte
            UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level
        ) &&
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
         (tlbi.level == TLBILevel_Any || !tlb_entry.walksta
when TLBIOp_VAA, TLBIPOp_VAA
    constant integer addr_lsb = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
              tlbi.security == tlb_entry.context.ss &&
              tlbi.regime == tlb_entry.context.regime &&
              (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
              tlbi.address<55:addr_lsb> == tlb_entry.context.ia<
              (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                  DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.conte
                  UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level
              ) &&
              ((tlbi.d128 && tlb_entry.context.isd128) ||
               (tlbi.d64 && !tlb_entry.context.isd128) ||
               (tlbi.d64 && tlbi.d128)) &&
               (tlbi.level == TLBILevel_Any || !tlb_entry.walksta
when TLBIOp_VA, TLBIPOp_VA
    constant integer addr_lsb = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
              tlbi.security == tlb_entry.context.ss &&
              tlbi.regime == tlb_entry.context.regime &&
              (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
              (!UseASID(tlb_entry.context) || tlbi.asid == tlb_e
                  tlb_entry.context.nG == '0') &&
                  tlbi.address<55:addr_lsb> == tlb_entry.context.ia<
                  (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                      DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.conte
                      UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level
                  ) &&
                  ((tlbi.d128 && tlb_entry.context.isd128) ||
                   (tlbi.d64 && !tlb_entry.context.isd128) ||
                   (tlbi.d64 && tlbi.d128)) &&
                   (tlbi.level == TLBILevel_Any || !tlb_entry.walksta
when TLBIOp_VMAIL
    match = (tlb_entry.context.includes_s1 &&
              tlbi.security == tlb_entry.context.ss &&
              tlbi.regime == tlb_entry.context.regime &&
              (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
when TLBIOp_VMAILS12
    match = (tlbi.security == tlb_entry.context.ss &&
              tlbi.regime == tlb_entry.context.regime &&
              (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
when TLBIOp_RIPAS2, TLBIPOp_RIPAS2
    match = (!tlb_entry.context.includes_s1 && tlb_entry.context.
              tlbi.security == tlb_entry.context.ss &&
              tlbi.regime == tlb_entry.context.regime &&
              (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_e
              tlbi.ipaspace == tlb_entry.context.ipaspace &&
              (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_e

```

```

        (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.t
          UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level
        ((tlbi.d128 && tlb_entry.context.isd128) ||
         (tlbi.d64 && !tlb_entry.context.isd128) ||
         (tlbi.d64 && tlbi.d128)) &&
        UInt(tlbi.address<55:0>) <= UInt(entry_end_address)
        UInt(tlbi.end_address<55:0>) > UInt(entry_start_address)
when TLBIOp_RVAA, TLBIPOp_RVAA
  match = (tlb_entry.context.includes_s1 &&
            tlbi.security == tlb_entry.context.ss &&
            tlbi.regime == tlb_entry.context.regime &&
            (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_entry.vmid) &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.tg) &&
            (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.t
              UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level
            ((tlbi.d128 && tlb_entry.context.isd128) ||
             (tlbi.d64 && !tlb_entry.context.isd128) ||
             (tlbi.d64 && tlbi.d128)) &&
             UInt(tlbi.address<55:0>) <= UInt(entry_end_address)
             UInt(tlbi.end_address<55:0>) > UInt(entry_start_address)
when TLBIOp_RVA, TLBIPOp_RVA
  match = (tlb_entry.context.includes_s1 &&
            tlbi.security == tlb_entry.context.ss &&
            tlbi.regime == tlb_entry.context.regime &&
            (!UseVMID(tlb_entry.context) || tlbi.vmid == tlb_entry.vmid) &&
            (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.asid) &&
            (tlb_entry.context.nG == '0') &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.tg) &&
            (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.t
              UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level
            ((tlbi.d128 && tlb_entry.context.isd128) ||
             (tlbi.d64 && !tlb_entry.context.isd128) ||
             (tlbi.d64 && tlbi.d128)) &&
             UInt(tlbi.address<55:0>) <= UInt(entry_end_address)
             UInt(tlbi.end_address<55:0>) > UInt(entry_start_address)
when TLBIOp_RPA
  entry_end_address<55:0> = (tlb_entry.walkstate.baseaddress +
                                entry_block_mask<55:0>);
  entry_start_address<55:0> = (tlb_entry.walkstate.baseaddress +
                                NOT entry_block_mask<55:0>);
  match = (tlb_entry.context.includes_gpt &&
            UInt(tlbi.address<55:0>) <= UInt(entry_end_address)
            UInt(tlbi.end_address<55:0>) > UInt(entry_start_address)
when TLBIOp_PAALL
  match = tlb_entry.context.includes_gpt;

if tlbi.attr == TLBI_ExcludeXS && tlb_entry.context.xs == '1' then
  match = FALSE;

return match;

```

## Library pseudocode for aarch64/functions/tlbi/TLBIMemAttr

```

// TLBIMemAttr
// =====
// Defines the attributes of the memory operations that must be completed
// in order to deem the TLBI operation as completed.

```

```

enumeration TLBIMemAttr {
    TLBI_AllAttr,           // All TLB entries within the scope of the instruction
    TLBI_ExcludeXS         // Only TLB entries with XS=0 within the scope
};

```

### Library pseudocode for aarch64/functions/tlbi/TLBIOp

```

// TLBIOp
// =====

enumeration TLBIOp {
    TLBIOp_DALL,           // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,           // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
};

```

### Library pseudocode for aarch64/functions/tlbi/TLBIPRange

```

// TLBIPRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIPRange(Regime regime, bits(1)
    boolean valid          = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    bits(2) tg            = Xt<47:46>;
    integer scale          = UInt(Xt<45:44>);
    integer num             = UInt(Xt<43:39>);

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB

```

```

        tg_bits = 12;
        start_address<55:12> = Xt<107:64>;
        start_address<63:56> = Replicate(Xt<107>, 8);
when '10' // 16KB
        tg_bits = 14;
        start_address<55:14> = Xt<107:66>;
        start_address<63:56> = Replicate(Xt<107>, 8);
when '11' // 64KB
        tg_bits = 16;
        start_address<55:16> = Xt<107:68>;
        start_address<63:56> = Replicate(Xt<107>, 8);
otherwise
    Unreachable();

integer range = (num+1) << (5*scale + 1 + tg_bits);
end_address = start_address + range<63:0>

if end_address<55> != start_address<55> then
    // overflow, saturate it
    end_address = Replicate(start_address<55>, 64-55) : Ones(55);

return (valid, tg, start_address, end_address);

```

## Library pseudocode for aarch64/functions/tlbi/TLBIRange

```

// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64)
    boolean valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address = Zeros(64);

    bits(2) tg      = Xt<47:46>;
    integer scale  = UInt(Xt<45:44>);
    integer num    = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<48:12> = Xt<36:0>;
                start_address<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<50:14> = Xt<36:0>;

```

```

        start_address<63:51> = Replicate(Xt<36>, 13);
when '11' // 64KB
    tg_bits = 16;
    start_address<52:16> = Xt<36:0>;
    start_address<63:53> = Replicate(Xt<36>, 11);
otherwise
    Unreachable();

integer range = (num+1) << (5*scale + 1 + tg_bits);
end_address = start_address + range<63:0>;

if end_address<52> != start_address<52> then
    // overflow, saturate it
    end_address = Replicate(start_address<52>, 64-52) : Ones(52);

return (valid, tg, start_address, end_address);

```

## Library pseudocode for aarch64/functions/tlbi/TLBIRecord

```

// TLBIRecord
// =====
// Details related to a TLBI operation.

type TLBIRecord is (
    TLBIOp          op,
    boolean            from_aarch64, // originated as an AArch64 operation
    SecurityState   security,
    Regime          regime,
    bits(16)           vmid,
    bits(16)           asid,
    TLBILevel       level,
    TLBIMemAttr    attr,
    PASpace         ipaspace,      // For operations that take IPA as input
    bits(64)           address,       // input address, for range operations
    bits(64)           end_address,  // for range operations, end address
    boolean             d64,          // For operations that evict VMSAv8-64
    boolean             d128,         // For operations that evict VMSAv9-128
    bits(4)            ttl,          // translation table walk level holding
                                    // for the address being invalidated
                                    // For Non-Range Invalidations:
                                    // When the ttl is
                                    //   '00xx' : this applies to all
                                    // Otherwise : TLBIP instructions
                                    //               entries only
                                    //               TLBI instructions
                                    //               entries only
                                    // For Range Invalidations:
                                    // When the ttl is
                                    //   '00' : this applies to all
                                    // Otherwise : TLBIP instructions
                                    //               entries only
                                    //               TLBI instructions
                                    //               entries only
    bits(2)            tg            // for range operations, translation
)

```

## Library pseudocode for aarch64/functions/tlbi/VMID

```

// VMID []
// =====
// Effective VMID.

bits(16) VMID []
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if IsFeatureImplemented(FEAT_VMID16) && VTCR_EL2.VS == '1'
                return VTTBR_EL2.VMID;
            else
                return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            return ZeroExtend(VTTBR.VMID, 16);
    elseif HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        return Zeros(16);
    else
        return VMID_NONE;

```

### Library pseudocode for aarch64/functions/tlbi/VMID\_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```

### Library pseudocode for aarch64/functions/tme/ CheckTransactionalSystemAccess

```

// CheckTransactionalSystemAccess()
// =====
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(bits(2) op0, bits(3) op1, bits(4)
                                       bits(3) op2, bit read)
    case read:op0:op1:crn:crm:op2 of
        when '0 00 011 0100 xxxx 11x' return TRUE;          // MSR (imm): D
        when '0 01 011 0111 0100 001' return TRUE;          // DC ZVA
        when '0 11 011 0100 0010 00x' return TRUE;          // MSR: NZCV, D
        when '0 11 011 0100 0100 00x' return TRUE;          // MSR: FPCR, F
        when '0 11 000 0100 0110 000' return TRUE;          // MSR: ICC_PMR
        when '0 11 011 1001 1100 100' return TRUE;          // MRS: PMSWINC
        when '1 11 011 0010 0101 001'                         // MRS: GCSPR_E
            return PSTATE.EL == EL0;
        // MRS: GCSPR_EL1 at EL1 OR at EL2 when E2H is '1'
        when '1 11 000 0010 0101 001'
            return PSTATE.EL == EL1 || (PSTATE.EL == EL2 && HCR_EL2.E2H
        when '1 11 100 0010 0101 001'                         // MRS: GCSPR_E
            return PSTATE.EL == EL2 && HCR_EL2.E2H == '0';
        when '1 11 110 0010 0101 001'                         // MRS: GCSPR_E
            return PSTATE.EL == EL3;
        when '0 01 011 0111 0111 000' return TRUE;          // GCSPUSHM
        when '1 01 011 0111 0111 001' return TRUE;          // GCSPOPM
        when '0 01 011 0111 0111 010' return TRUE;          // GCSSS1
        when '1 01 011 0111 0111 011' return TRUE;          // GCSSS2
        when '0 01 000 0111 0111 110' return TRUE;          // GCSPOPX
        when '1 11 101 0010 0101 001' return FALSE;         // MRS: GCSPR_E
        when '1 11 000 0010 0101 010' return FALSE;         // MRS: GCSCREO
        when '1 11 000 0010 0101 000' return FALSE;         // MRS: GCSCR_E
        when '1 11 101 0010 0101 000' return FALSE;         // MRS: GCSCR_E

```

```

when '1 11 100 0010 0101 000' return FALSE;           // MRS: GCSCR_E
when '1 11 110 0010 0101 000' return FALSE;           // MRS: GCSCR_E
when '1 11 xxx 0xxx xxxx xxx' return TRUE;           // MRS: op0=3,
when '1 11 xxx 100x xxxx xxx' return TRUE;           // MRS: op0=3,
when '1 11 xxx 1010 xxxx xxx' return TRUE;           // MRS: op0=3,
when '1 11 000 1100 1x00 010' return TRUE;           // MRS: op0=3,
when '1 11 000 1100 1011 011' return TRUE;           // MRS: op0=3,
when '1 11 xxx 1101 xxxx xxx' return TRUE;           // MRS: op0=3,
when '1 11 xxx 1110 xxxx xxx' return TRUE;           // MRS: op0=3,
when '0 01 011 0111 0011 111' return TRUE;           // CPP RCTX
when '0 01 011 0111 0011 10x' return TRUE;           // CFP RCTX, DV
when 'x 11 xxx 1x11 xxxx xxx'
    return boolean IMPLEMENTATION_DEFINED;
otherwise return FALSE;                                // All other SY

```

### Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

```

// CommitTransactionalWrites()
// =====
// Makes all transactional writes to memory observable by other PEs and
// the transactional read and write sets.

CommitTransactionalWrites();

```

### Library pseudocode for aarch64/functions/tme/DiscardTransactionalWrites

```

// DiscardTransactionalWrites()
// =====
// Discards all transactional writes to memory and reset the transactional
// read and write sets.

DiscardTransactionalWrites();

```

### Library pseudocode for aarch64/functions/tme/FailTransaction

```

// FailTransaction()
// =====

FailTransaction(TMFailure cause, boolean retry)
    FailTransaction(cause, retry, FALSE, Zeros(15));
    return;

// FailTransaction()
// =====
// Exits Transactional state and discards transactional updates to registers
// and memory.

FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits
    assert !retry || !interrupt;

    if IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed(PSTATE.EI)
        BRBFCR_EL1.LASTFAILED = '1';

    DiscardTransactionalWrites();
    // For trivial implementation no transaction checkpoint was taken

```

```

if cause != TMFailure_TRIVIAL then
    RestoreTransactionCheckpoint();
ClearExclusiveLocal(ProcessorID());

bits(64) result = Zeros(64);

result<23> = if interrupt then '1' else '0';
result<15> = if retry && !interrupt then '1' else '0';
case cause of
    when TMFailure_TRIVIAL result<24> = '1';
    when TMFailure_DBG result<22> = '1';
    when TMFailure_NEST result<21> = '1';
    when TMFailure_SIZE result<20> = '1';
    when TMFailure_ERR result<19> = '1';
    when TMFailure_IMP result<18> = '1';
    when TMFailure_MEM result<17> = '1';
    when TMFailure_CNCL result<16> = '1'; result<14:0> = reason;

TSTATE.depth = 0;
X[TSTATE.Rt, 64] = result;
boolean branch_conditional = FALSE;
BranchTo(TSTATE.nPC, BranchType_TFAIL, branch_conditional);
EndOfInstruction();
return;

```

## Library pseudocode for aarch64/functions/tme/IsTMEEnabled

```

// IsTMEEnabled()
// =====
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise

boolean IsTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
        if SCR_EL3.TME == '0' then
            return FALSE;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR_EL2.TME == '0' then
            return FALSE;
    return TRUE;

```

## Library pseudocode for aarch64/functions/tme/MemHasTransactionalAccess

```

// MemHasTransactionalAccess()
// =====
// Function checks if transactional accesses are not supported for an address
// range or memory type.

boolean MemHasTransactionalAccess(MemoryAttributes memattrs)
    if ((memattrs.shareability == Shareability_ISH ||
        memattrs.shareability == Shareability_OSH) &&
        memattrs.memtype == MemType_Normal &&
        memattrs.inner.attrs == MemAttr_WB &&
        memattrs.inner.hints == MemHint_RWA &&
        memattrs.inner.transient == FALSE &&
        memattrs.outer.hints == MemHint_RWA &&
        memattrs.outer.attrs == MemAttr_WB &&
        memattrs.outer.transient == FALSE) then

```

```

        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Memory Region does not s
```

### Library pseudocode for aarch64/functions/tme/RestoreTransactionCheckpoint

```

// RestoreTransactionCheckpoint()
// =====
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
    SP[] = TSTATE.SP;
    ICC_PMR_EL1 = TSTATE.ICC_PMR_EL1;
    PSTATE.<N,Z,C,V> = TSTATE.nzcv;
    PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;

    for n = 0 to 30
        X[n, 64] = TSTATE.X[n];

    if IsFPEEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant integer VL = CurrentVL;
            constant integer PL = VL DIV 8;
            for n = 0 to 31
                Z[n, VL] = TSTATE.Z[n]<VL-1:0>;
            for n = 0 to 15
                P[n, PL] = TSTATE.P[n]<PL-1:0>;
                FFR[PL] = TSTATE.FFR<PL-1:0>;
        else
            for n = 0 to 31
                V[n, 128] = TSTATE.Z[n]<127:0>;
    FPCR = TSTATE.FPCR;
    FPSR = TSTATE.FPSR;

    if IsFeatureImplemented(FEAT_GCS) then
        case PSTATE.EL of
            when EL0 GCSR_EL0 = TSTATE.GCSR_ELx;
            when EL1 GCSR_EL1 = TSTATE.GCSR_ELx;
            when EL2 GCSR_EL2 = TSTATE.GCSR_ELx;
            when EL3 GCSR_EL3 = TSTATE.GCSR_ELx;

    return;
```

### Library pseudocode for aarch64/functions/tme/StartTrackingTransactionalReadsWrites

```

// StartTrackingTransactionalReadsWrites()
// =====
// Starts tracking transactional reads and writes to memory.

StartTrackingTransactionalReadsWrites();
```

### Library pseudocode for aarch64/functions/tme/TMFailure

```

// TMFailure
// ======
// Transactional failure causes

enumeration TMFailure {
    TMFailure_CNCL,           // Executed a TCANCEL instruction
    TMFailure_DBG,            // A debug event was generated
    TMFailure_ERR,            // A non-permissible operation was attempted
    TMFailure_NEST,           // The maximum transactional nesting level was e
    TMFailure_SIZE,           // The transactional read or write set limit was
    TMFailure_MEM,            // A transactional conflict occurred
    TMFailure_TRIVIAL,        // Only a TRIVIAL version of TM is available
    TMFailure_IMP             // Any other failure cause
};

```

### Library pseudocode for aarch64/functions/tme/TMState

```

// TMState
// ======
// Transactional execution state bits.
// There is no significance to the field order.

type TMState is (
    integer      depth,          // Transaction nesting depth
    integer      Rt,             // TSTART destination register
    bits(64)     nPC,            // Fallback instruction address
    array[0..30]  of bits(64)   X,  // General purpose registers
    array[0..31]  of bits(MAX_VL) Z, // Vector registers
    array[0..15]  of bits(MAX_PL) P, // Predicate registers
    bits(MAX_PL) FFR,           // First Fault Register
    bits(64)     SP,             // Stack Pointer at current EL
    bits(64)     FPCR,           // Floating-point Control Register
    bits(64)     FPSR,           // Floating-point Status Register
    bits(64)     ICC_PMR_EL1,   // Interrupt Controller Interrupt
    bits(64)     GCSPR_ELx,     // GCS pointer for current EL
    bits(4)      nzcv,           // Condition flags
    bits(1)      D,               // Debug mask bit
    bits(1)      A,               // SError interrupt mask bit
    bits(1)      I,               // IRQ mask bit
    bits(1)      F,               // FIQ mask bit
)

```

### Library pseudocode for aarch64/functions/tme/TSTATE

```
TMState TSTATE;
```

### Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

```

// TakeTransactionCheckpoint()
// =====
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
    TSTATE.SP          = SP[];
    TSTATE.ICC_PMR_EL1 = ICC_PMR_EL1;

```

```

TSTATE.nzcv          = PSTATE.<N,Z,C,V>;
TSTATE.<D,A,I,F>    = PSTATE.<D,A,I,F>;

for n = 0 to 30
    TSTATE.X[n] = X[n, 64];

if IsFPEEnabled(PSTATE.EL) then
    if IsSVEEnabled(PSTATE.EL) then
        constant integer VL = CurrentVL;
        constant integer PL = VL DIV 8;
        for n = 0 to 31
            TSTATE.Z[n]<VL-1:0> = Z[n, VL];
        for n = 0 to 15
            TSTATE.P[n]<PL-1:0> = P[n, PL];
            TSTATE.FFR<PL-1:0> = FFR[PL];
    else
        for n = 0 to 31
            TSTATE.Z[n]<127:0> = V[n, 128];
    TSTATE.FPCR = FPCR;
    TSTATE.FPSR = FPSR;

if IsFeatureImplemented(FEAT_GCS) then
    case PSTATE.EL of
        when EL0 TSTATE.GCSPR_ELx = GCSPR_EL0;
        when EL1 TSTATE.GCSPR_ELx = GCSPR_EL1;
        when EL2 TSTATE.GCSPR_ELx = GCSPR_EL2;
        when EL3 TSTATE.GCSPR_ELx = GCSPR_EL3;

return;

```

## Library pseudocode for aarch64/functions/tme/TransactionStartTrap

```

// TransactionStartTrap()
// =====
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
    bits(2) targetEL;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

except = ExceptionSyndrome(Exception_TSTARTAccessTrap);
except.syndrome<9:5> = dreg<4:0>

if UInt(PSTATE.EL) > UInt(EL1) then
    targetEL = PSTATE.EL;
elseif EL2Enabled() && HCR_EL2.TGE == '1' then
    targetEL = EL2;
else
    targetEL = EL1;
AArch64.TakeException(targetEL, except, preferred_exception_return,

```

## Library pseudocode for aarch64/functions/vbitop/VBitOp

```

// VBitOp
// =====
// Vector bit select instruction types.

```

```
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_
```

### Library pseudocode for aarch64/translation/attrs/AArch64.MAIRAttr

```
// AArch64.MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch64.MAIRAttr(integer index, MAIRType mair2, MAIRType mair)
    assert (index < 8 || (IsFeatureImplemented(FEAT_AIE) && (index < 16))
    if (index > 7) then
        return Elem[mair2, index-8, 8]; // Read from LSB at MAIR2
    else
        return Elem[mair, index, 8];
```

### Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =====
// Called before executing the instruction of length "size" bytes at "vaddress"
// in the "regime" translation regime, when either debug exceptions are enabled, or halting
// is allowed.

FaultRecord AArch64.CheckBreakpoint(FaultRecord fault_in, bits(64) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    FaultRecord fault = fault_in;
    boolean match = FALSE;
    boolean mismatch = TRUE;                                // Default assumption that all
                                                            // the range of all address matches
    boolean mismatch_bp = FALSE;                           // Has a breakpoint been configured

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, is_mismatch_i) = AArch64.BreakpointMatch(i, vaddress,
        if is_mismatch_i then
            mismatch_bp = TRUE;
            mismatch     = mismatch && !match_i;
        else
            match = match || match_i;

        if match || (mismatch && mismatch_bp) then
            fault.statuscode = Fault_Debug;
            if HaltOnBreakpointOrWatchpoint() then
                reason = DebugHalt_Breakpoint;
                Halt(reason);

    return fault;
```

### Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// =====
```

```

// Called on each access to check for a debug exception or entry to Debug
FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccessDescriptor accdesc)

    FaultRecord fault = NoFault(accdesc);
    boolean generate_exception;

    boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType\_NonMemory);
    boolean i_side = (accdesc.acctype == AccessType\_IFETCH);
    if accdesc.acctype == AccessType\_NV2 then
        mask = '0';
        ss = CurrentSecurityState();
        generate_exception = (AArch64.GenerateDebugExceptionsFrom(EL2, MDSCR_EL1.MDE == '1');
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(fault, vaddress, accdesc, size);
        elseif i_side then
            fault = AArch64.CheckBreakpoint(fault, vaddress, accdesc, size);

    return fault;

```

## Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```

// AArch64.CheckWatchpoint()
// =====
// Called before accessing the memory location of "size" bytes at "addr"
// when either debug exceptions are enabled for the access, or halting
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(FaultRecord fault_in, bits(64) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    FaultRecord fault      = fault_in;
    bits(64) vaddress       = vaddress_in;
    integer size            = size_in;
    boolean rounded_match   = FALSE;
    bits(64) original_vaddress = vaddress;
    integer original_size   = size;

    if accdesc.acctype == AccessType\_DC then
        if accdesc.cacheop != CacheOp\_Invalidate then
            return fault;
    elseif !IsDataAccess(accdesc.acctype) then
        return fault;

    // In case of set of contiguous memory accesses each call to this function
    // - the lowest accessed address is rounded down to the nearest multiple
    // - the highest accessed address is rounded up to the nearest multiple
    // Since the WPF field is set if the implementation does rounding, returning
    // false match, it would be acceptable to return TRUE for either/boundary
    // access.
    if IsSVEAccess(accdesc) || IsSMEAccess(accdesc) then
        integer upper_vaddress = UInt(original_vaddress) + original_size;

```

```

        if ConstrainUnpredictableBool(Unpredictable_16BYTEROUNDEDOWNAC
            vaddress = Align(vaddress, 16);
            rounded_match = TRUE;
        if ConstrainUnpredictableBool(Unpredictable_16BYTEROUNDEDUPACCE
            upper_vaddress = Align(upper_vaddress + 15, 16) ;
            rounded_match = TRUE;
            size = upper_vaddress - UInt(vaddress);

        for i = 0 to NumWatchpointsImplemented() - 1
            if AArch64.WatchpointMatch(i, vaddress, size, accdesc) then
                fault.maybe_false_match = rounded_match;
                fault.watchpt_num = i;
                fault.statuscode = Fault_Debug;
                if DBGWCR_EL1[i].LSC<0> == '1' && accdesc.read then
                    fault.write = FALSE;
                elsif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
                    fault.write = TRUE;
            if (fault.statuscode == Fault_Debug && HaltOnBreakpointOrWatchpoint
                !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first))
                reason = DebugHalt_Watchpoint;
                EDWAR = vaddress;
                is_async = FALSE;
                Halt(reason, is_async, fault);
            return fault;
    
```

### Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.IASize

```

// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);

```

### Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.LeafBase

```

// AArch64.LeafBase()
// =====
// Extract the address embedded in a block and page descriptor pointing
// base of a memory block

bits(56) AArch64.LeafBase(bits(N) descriptor, bit d128, bit ds,
                           TGx tgx, integer level)
    bits(56) leafbase = Zeros(56);

    granulebits = TGxGranuleBits(tgx);
    descsizeilog2 = if d128 == '1' then 4 else 3;
    constant integer stride = granulebits - descsizeilog2;
    constant integer leafsize = granulebits + stride * (FINAL_LEVEL - 1)

    leafbase<47:0> = Align(descriptor<47:0>, 1 << leafsize);

    if IsFeatureImplemented(FEAT_D128) && d128 == '1' then
        leafbase<55:48> = descriptor<55:48>;
        return leafbase;
    if IsFeatureImplemented(FEAT_LPA) && tgx == TGx_64KB then
        leafbase<51:48> = descriptor<15:12>;

```

```

        elseif ds == '1' then
            leafbase<51:48> = descriptor<9:8>:descriptor<49:48>;
        return leafbase;
    
```

## Library pseudocode for aarch64/translation/vmsa\_addrCalc/ AArch64.NextTableBase

```

// AArch64.NextTableBase()
// =====
// Extract the address embedded in a table descriptor pointing to the b
// the next level table of descriptors

bits(56) AArch64.NextTableBase(bits(N) descriptor, bit d128, bits(2) sk
    bits(56) tablebase = Zeros(56);
    integer granulebits = TGxGranuleBits(tgx);
    integer tablesiz
        if d128 == '1' then
            integer descsizeilog2 = 4;
            integer stride = granulebits - descsizeilog2;
            tablesiz
                else
                    tablesiz
            case tgx of
                when TGx_4KB tablebase<47:12> = descriptor<47:12>;
                when TGx_16KB tablebase<47:14> = descriptor<47:14>;
                when TGx_64KB tablebase<47:16> = descriptor<47:16>;
            tablebase = Align(tablebase, 1 << tablesiz
                if IsFeatureImplemented(FEAT_D128) && d128 == '1' then
                    tablebase<55:48> = descriptor<55:48>;
                    return tablebase;
                if IsFeatureImplemented(FEAT_LPA) && tgx == TGx_64KB then
                    tablebase<51:48> = descriptor<15:12>;
                    return tablebase;
                if ds == '1' then
                    tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;
                    return tablebase;
                return tablebase;
    
```

## Library pseudocode for aarch64/translation/vmsa\_addrCalc/ AArch64.PhysicalAddressSize

```

// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bit d128, bits(3) encoded_ps, TGx t
    integer ps;
    integer max_ps;

    case encoded_ps of
        when '000' ps = 32;
        when '001' ps = 36;
    
```

```

        when '010'  ps = 40;
        when '011'  ps = 42;
        when '100'  ps = 44;
        when '101'  ps = 48;
        when '110'  ps = 52;
        when '111'  ps = 56;
    if !IsFeatureImplemented(FEAT_D128) || d128 == '0' then
        if tgx != TGx_64KB && !IsFeatureImplemented(FEAT_LPA2) then
            max_ps = Min(48, AArch64.PAMax());
        elseif !IsFeatureImplemented(FEAT_LPA) then
            max_ps = Min(48, AArch64.PAMax());
        else
            max_ps = Min(52, AArch64.PAMax());
    else
        max_ps = AArch64.PAMax();

    return Min(ps, max_ps);

```

### Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.S1SLTTEEntryAddress

```

// AArch64.S1SLTTEEntryAddress()
// =====
// Compute the first stage 1 translation table descriptor address within
// table pointed to by the base at the start level

FullAddress AArch64.S1SLTTEEntryAddress(integer level, S1TTWParams walkp
                                         bits(64) ia, FullAddress tableba
                                         // Input Address size
                                         iasize      = AArch64.IASize(walkparams.txsz);
                                         granulebits = TGxGranuleBits(walkparams.tgx);
                                         descsizeilog2 = if walkparams.d128 == '1' then 4 else 3;
                                         stride      = granulebits - descsizeilog2;
                                         levels       = FINAL LEVEL - level;

                                         bits(56) index;
                                         constant integer lsb = levels*stride + granulebits;
                                         constant integer msb = iasize - 1;
                                         index = ZeroExtend(ia<msb:lsb>:Zeros(descsizeilog2), 56);

                                         FullAddress descaddress;
                                         descaddress.address = tablebase.address OR index;
                                         descaddress.paspace = tablebase.paspace;

                                         return descaddress;

```

### Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.S1StartLevel

```

// AArch64.S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translati
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);

```

```

granulebits = TGxGranuleBits(walkparams.tgx);
descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
stride = granulebits - descsize;
s1startlevel = FINAL LEVEL - (((iasize-1) - granulebits) DIV stride);
if walkparams.d128 == '1' then
    s1startlevel = s1startlevel + UInt(walkparams.skl);
return s1startlevel;

```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.S1TTBaseAddress

```

// AArch64.S1TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation
bits(56) AArch64.S1TTBaseAddress(S1TTWParams walkparams, Regime regime,
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsize = if walkparams.d128 == '1' then 4 else 3;
    stride = granulebits - descsize;
    startlevel = AArch64.S1StartLevel(walkparams);
    levels = FINAL LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table
    tsize = (iasize - (levels*stride + granulebits)) + descsize;
    if IsFeatureImplemented(FEAT_D128) && walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if regime == Regime EL3 then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>;ttbr<47:5>;
    elseif ((IsFeatureImplemented(FEAT_LPA) && walkparams.tgx == TGX_64K) ||
            (walkparams.ps == '110') || (walkparams.ds == '1')) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>;ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;

```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.S2SLTTEEntryAddress

```

// AArch64.S2SLTTEEntryAddress()
// =====
// Compute the first stage 2 translation table descriptor address within
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEEntryAddress(S2TTWParams walkparams, bits(56)
                                         FullAddress tablebase)
    startlevel = AArch64.S2StartLevel(walkparams);
    iasize = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);

```

```

descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
stride      = granulebits - descsizelog2;
levels      = FINAL LEVEL - startlevel;

bits(56) index;
constant integer lsb = levels*stride + granulebits;
constant integer msb = iasize - 1;
index = ZeroExtend(ipa<msb:lsb>:Zeros(descsizelog2), 56);

FullAddress descaddress;
descaddress.address = tablebase.address OR index;
descaddress.paspace = tablebase.paspace;

return descaddress;

```

### **Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.S2StartLevel**

```

// AArch64.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    if walkparams.d128 == '1' then
        iasize      = AArch64.IASize(walkparams.txsz);
        granulebits = TGxGranuleBits(walkparams.tgx);
        descsizelog2 = 4;
        stride      = granulebits - descsizelog2;
        s2startlevel = FINAL LEVEL - (((iasize-1) - granulebits) DIV stride);
        s2startlevel = s2startlevel + UInt(walkparams.skl);

    return s2startlevel;

case walkparams.tgx of
    when TGx 4KB
        case walkparams.s12:walkparams.s10 of
            when '000' return 2;
            when '001' return 1;
            when '010' return 0;
            when '011' return 3;
            when '100' return -1;
    when TGx 16KB
        case walkparams.s10 of
            when '00' return 3;
            when '01' return 2;
            when '10' return 1;
            when '11' return 0;
    when TGx 64KB
        case walkparams.s10 of
            when '00' return 3;
            when '01' return 2;
            when '10' return 1;

```

### **Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.S2TTBaseAddress**

```

// AArch64.S2TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation

bits(56) AArch64.S2TTBaseAddress(S2TTWParams walkparams, PASpace paspac
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASIZE(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    startlevel   = AArch64.S2StartLevel(walkparams);
    levels       = FINAL LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table
    tsize = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if IsFeatureImplemented(FEAT_D128) && walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if paspace == PAS_Secure then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>;ttbr<47:5>;
    elseif ((IsFeatureImplemented(FEAT_LPA) && walkparams.tgx == TGx_64K
        walkparams.ps == '110') || (walkparams.ds == '1')) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>;ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;

```

### Library pseudocode for aarch64/translation/vmsa\_addrcalc/ AArch64.TTEntryAddress

```

// AArch64.TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed
// to by the table base

FullAddress AArch64.TTEntryAddress(integer level, bit d128, bits(2) skl,
                                     bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize      = AArch64.IASIZE(txsz);
    granulebits = TGxGranuleBits(tgx);
    descsizelog2 = if d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    levels       = FINAL LEVEL - level;

    bits(56) index;

    constant integer lsb = levels*stride + granulebits;
    constant integer nstride = if d128 == '1' then UInt(skl) + 1 else 1;
    constant integer msb = (lsb + (stride * nstride)) - 1;
    index = ZeroExtend(ia<msb:lsb>;Zeros(descsizelog2), 56);

    FullAddress descaddress;

```

```

descaddress.address = tablebase.address OR index;
descaddress.paspace = tablebase.paspace;

return descaddress;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.AddrTop

```

// AArch64.AddrTop()
// =====
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccessType acctype, bit tbi)
    if tbid == '1' && acctype == AccessType_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.ContiguousBitFaults

```

// AArch64.ContiguousBitFaults()
// =====
// If contiguous bit is set, returns whether the translation size exceeds
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bit d128, bits(6) txsz, TGx tgx, in
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize = TranslationSize(d128, tgx, level) + ContiguousSize(d128, tsize);

    return (tsize > iasize &&
            boolean IMPLEMENTATION_DEFINED "Translation fault on mispredic"

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.IPAIsOutOfRange

```

// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(56) ipa, S2TTWParams walkparams)
    //Input Address size
    constant integer iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 56 then
        return !IsZero(ipa<55:iasize>);
    else
        return FALSE;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number

boolean AArch64.OAOutOfRange(bits(56) address, bit d128, bits(3) ps, TGX tGX)
    // Output Address size
    constant integer oasize = AArch64.PhysicalAddressSize(d128, ps, tGX);

    if oasize < 56 then
        return !IsZero(address<55:oasize>);
    else
        return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S1CheckPermissions

```
// AArch64.S1CheckPermissions()
// =====
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

FaultRecord AArch64.S1CheckPermissions(FaultRecord fault_in, Regime regime,
                                         S1TTWParams walkparams, AccessDesc accdesc,
                                         FaultRecord fault = fault_in;
                                         Permissions permissions = walkstate.permissions;
                                         S1AccessControls s1perms;

    s1perms = AArch64.S1ComputePermissions(regime, walkstate, walkparams);

    if accdesc.acctype == AccessType\_IFETCH then
        if s1perms.overlay && s1perms.ox == '0' then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif (walkstate.memattrs.memtype == MemType\_Device &&
                ConstrainUnpredictable\(Unpredictable\_INSTRDEVICE\) == ConstrainUnpredictable\(ConstrainUnpredictable\_INSTRDEVICE\))
            fault.statuscode = Fault\_Permission;
        elseif s1perms.x == '0' then
            fault.statuscode = Fault\_Permission;
    elseif accdesc.acctype == AccessType\_DC then
        if accdesc.cacheop == CacheOp\_Invalidate then
            if s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault\_Permission;
                fault.overlay = TRUE;
            elseif s1perms.w == '0' then
                fault.statuscode = Fault\_Permission;
        // DC from privileged context which clean cannot generate a Permission fault
    elseif accdesc.el == EL0 then
        if s1perms.overlay && s1perms.or == '0' then
            fault.statuscode = Fault\_Permission;
            fault.overlay = TRUE;
        elseif (walkparams.cmow == '1' &&
                accdesc.opscope == CacheOpScope\_PoC &&
                accdesc.cacheop == CacheOp\_CleanInvalidate &&
                s1perms.overlay && s1perms.ow == '0') then
```

```

        fault.statuscode = Fault_Permission;
        fault.overlay     = TRUE;
    elseif slperms.r == '0' then
        fault.statuscode = Fault_Permission;
    elseif (walkparams.cmow == '1' &&
            accdesc.opscope == CacheOpScope_PoC &&
            accdesc.cacheop == CacheOp_CleanInvalidate &&
            slperms.w == '0') then
        fault.statuscode = Fault_Permission;
    elseif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == ELO then
            if (slperms.overlay && slperms.or == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on E"
                fault.statuscode = Fault_Permission;
                fault.overlay     = TRUE;
            elseif walkparams.cmow == '1' && slperms.overlay && slperms.
                fault.statuscode = Fault_Permission;
                fault.overlay     = TRUE;
            elseif (slperms.r == '0' &&
                    boolean IMPLEMENTATION_DEFINED "Permission fault on E"
                    fault.statuscode = Fault_Permission;
            elseif walkparams.cmow == '1' && slperms.w == '0' then
                fault.statuscode = Fault_Permission;
            elseif IsFeatureImplemented(FEAT_GCS) && accdesc.acctype == AccessType_GCS
                if slperms.gcs == '0' then
                    fault.statuscode = Fault_Permission;
                elseif accdesc.write && walkparams.<ha,hd> != '11' && permission
                    fault.statuscode = Fault_Permission;
                    fault.dirtybit     = TRUE;
                    fault.write        = TRUE;
                elseif accdesc.read && slperms.overlay && slperms.or == '0' then
                    fault.statuscode = Fault_Permission;
                    fault.overlay     = TRUE;
                    fault.write        = FALSE;
                elseif accdesc.write && slperms.overlay && slperms.ow == '0' then
                    fault.statuscode = Fault_Permission;
                    fault.overlay     = TRUE;
                    fault.write        = TRUE;
                elseif accdesc.read && slperms.r == '0' then
                    fault.statuscode = Fault_Permission;
                    fault.write        = FALSE;
                elseif accdesc.write && slperms.w == '0' then
                    fault.statuscode = Fault_Permission;
                    fault.write        = TRUE;
                elseif (accdesc.write && accdesc.tagaccess &&
                        walkstate.memattrs.tags == MemTag_CanonicallyTagged) then
                    fault.statuscode     = Fault_Permission;
                    fault.write          = TRUE;
                    fault.s1tagnotdata = TRUE;
                elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.
                    permissions.ndirty == '1') then
                    fault.statuscode = Fault_Permission;
                    fault.dirtybit   = TRUE;
                    fault.write      = TRUE;
            return fault;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S1ComputePermissions

```
// AArch64.S1ComputePermissions()
// =====
// Computes the overall stage 1 permissions

S1AccessControls AArch64.S1ComputePermissions(Regime regime, TTWState
S1TTWParams walkparams, Permissions permissions)
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    if walkparams.pie == '1' then
        s1perms = AArch64.S1IndirectBasePermissions(regime, walkstate,
    else
        s1perms = AArch64.S1DirectBasePermissions(regime, walkstate, wa

    if accdesc.el == EL0 && !AArch64.S1E0POEnabled(regime, walkparams.r
        s1perms.overlay = FALSE;
    elseif accdesc.el != EL0 && !AArch64.S1POEnabled(regime) then
        s1perms.overlay = FALSE;

    if s1perms.overlay then
        sloverlay_perms = AArch64.S1OverlayPermissions(regime, walkstat
        s1perms.or = sloverlay_perms.or;
        s1perms.ow = sloverlay_perms.ow;
        s1perms.ox = sloverlay_perms.ox;

    // If wxn is set, overlay execute permissions is set to 0
    if s1perms.overlay && s1perms.wxn == '1' && s1perms.ox == '1' then
        s1perms.ow = '0';
    elseif s1perms.wxn == '1' then
        s1perms.x = '0';

    return s1perms;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S1DirectBasePermissions

```
// AArch64.S1DirectBasePermissions()
// =====
// Computes the stage 1 direct base permissions

S1AccessControls AArch64.S1DirectBasePermissions(Regime regime, TTWStat
S1TTWParams walkparams
    bit r, w, x;
    bit pr, pw, px;
    bit ur, uw, ux;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case permissions.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only,
```

```

        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

    // Apply hierarchical permissions
    case permissions.ap_table of
        when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
        when '01' (pr,pw,ur,uw) = ( pr, pw, '0','0'); // Privileged
        when '10' (pr,pw,ur,uw) = ( pr,'0', ur, '0'); // Read-only
        when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only

    // Locations writable by unprivileged cannot be executed by privileged
    px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
    ux = NOT(permissions.uxn OR permissions.uxn_table);

    if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime =
                    walkparams.nvl == '1')) then
        bit pan;
        if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN"
            accdesc.ss == SS_Secure &&
            walkstate.baseaddress.paspace == PAS_NonSecure &&
            walkparams.sif == '1') then
            ux = '0';

        if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN"
            accdesc.ss == SS_Realm && regime == Regime_EL20 &&
            walkstate.baseaddress.paspace != PAS_Realm) then
            ux = '0';

        pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (pr,pw) = ('1','1'); // No effect
            when '1' (pr,pw) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (pr,pw) = ( pr, pw); // No effect
            when '1' (pr,pw) = ( pr,'0'); // Read-only

        px = NOT(permissions.xn OR permissions.xn_table);

        (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

        // Compute WGN value
        wgn = walkparams.wgn AND w AND x;

        // Prevent execution from Non-secure space by PE in secure state if
        if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_Secure
            x = x AND NOT(walkparams.sif);
        // Prevent execution from non-Root space by Root
        if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root
            x = '0';
        // Prevent execution from non-Realm space by Realm EL2 and Realm EL20
        if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
            walkstate.baseaddress.paspace != PAS_Realm) then
            x = '0';

```

```

s1perms.r    = r;
s1perms.w    = w;
s1perms.x    = x;
s1perms.gcs = '0';
s1perms.wxn = wxn;
s1perms.overlay = TRUE;

return s1perms;

```

### **Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S1HasAlignmentFault**

```

// AArch64.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data access
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                    bit ntlsmd, MemoryAttributes memattr)
{
    if accdesc.acctype == AccessType IFETCH then
        return FALSE;
    elsif IsFeatureImplemented(FEAT_MTE) && accdesc.tagaccess && accdesc.acctype == AccessType DCZero
        return (memattr.memtype == MemType Device &&
                ConstrainUnpredictable\(Unpredictable DEVICETAGSTORE\) == TRUE);
    elseif accdesc.a32lsm && ntlsmd == '0' then
        return memattr.memtype == MemType Device && memattr.device != 0;
    elseif accdesc.acctype == AccessType DCZero
        return memattr.memtype == MemType Device;
    else
        return memattr.memtype == MemType Device && !aligned;
}

```

### **Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S1IndirectBasePermissions**

```

// AArch64.S1IndirectBasePermissions()
// =====
// Computes the stage 1 indirect base permissions

S1AccessControls AArch64.S1IndirectBasePermissions(Regime regime, TTWSTable ttwst,
                                                 S1TTWParams walkparams,
                                                 AccessDescriptor accdesc)

bit r, w, x, gcs, wxn, overlay;
bit pr, pw, px, pgcs, pwxn, p_overlay;
bit ur, uw, ux, ugcs, uwxn, u_overlay;
Permissions permissions = walkstate.permissions;
S1AccessControls s1perms;

// Apply privileged indirect permissions
case permissions.ppi of
    when '0000' (pr,pw,px,pgcs) = ('0','0','0','0'); // No access
    when '0001' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged
    when '0010' (pr,pw,px,pgcs) = ('0','0','1','0'); // Privileged
    when '0011' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged
    when '0100' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
    when '0101' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged
    when '0110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged

```

```

when '0111' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged
when '1000' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged
when '1001' (pr,pw,px,pgcs) = ('1','0','0','1'); // Privileged
when '1010' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged
when '1011' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
when '1100' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged
when '1101' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
when '1110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged
when '1111' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved

p_overlay = NOT(permissions.ppi[3]);
pwxn = if permissions.ppi == '0110' then '1' else '0';

if HasUnprivileged(regime) then
    // Apply unprivileged indirect permissions
    case permissions.upi of
        when '0000' (ur,uw,ux,ugcs) = ('0','0','0','0'); // No access
        when '0001' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged
        when '0010' (ur,uw,ux,ugcs) = ('0','0','1','0'); // Unprivileged
        when '0011' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged
        when '0100' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
        when '0101' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged
        when '0110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged
        when '0111' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged
        when '1000' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged
        when '1001' (ur,uw,ux,ugcs) = ('1','0','0','1'); // Unprivileged
        when '1010' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged
        when '1011' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
        when '1100' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged
        when '1101' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
        when '1110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged
        when '1111' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved

u_overlay = NOT(permissions.upi[3]);
uwxn = if permissions.upi == '0110' then '1' else '0';

// If the decoded permissions has either px or pgcs along with
// then all effective Stage 1 Base Permissions are set to 0
if ((px == '1' || pgcs == '1') && (uw == '1' || ugcs == '1')) then
    (pr,pw,px,pgcs) = ('0','0','0','0');
    (ur,uw,ux,ugcs) = ('0','0','0','0');

if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime =
    walkparams.nvl == '1')) then
    if PSTATE.PAN == '1' && (permissions.upi != '0000') then
        (pr,pw) = ('0','0');

if accdesc.el == EL0 then
    (r,w,x,gcs,wxn,overlay) = (ur,uw,ux,ugcs,uwxn,u_overlay);
else
    (r,w,x,gcs,wxn,overlay) = (pr,pw,px,pgcs,pwxn,p_overlay);

// Prevent execution from Non-secure space by PE in secure state if
if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_Secure
    x = x AND NOT(walkparams.sif);
    gcs = '0';
// Prevent execution from non-Root space by Root
if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root
    x = '0';
    gcs = '0';

```

```

// Prevent execution from non-Realm space by Realm EL2 and Realm EL1
if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &
    walkstate.baseaddress.paspace != PAS_Realm) then
    x = '0';
    gcs = '0';

s1perms.r      = r;
s1perms.w      = w;
s1perms.x      = x;
s1perms.gcs    = gcs;
s1perms.wxn    = wxn;
s1perms.overlay = overlay == '1';

return s1perms;

```

## Library pseudocode for aarch64/translation/vmsa\_faulsts/ AArch64.S1OverlayPermissions

```

// AArch64.S1OverlayPermissions()
// =====
// Computes the stage 1 overlay permissions

S1AccessControls AArch64.S1OverlayPermissions(Regime regime, TTWState w
                                              AccessDescriptor accdesc)

bit r, w, x;
bit pr, pw, px;
bit ur, uw, ux;
Permissions permissions = walkstate.permissions;
S1AccessControls sloveray_perms;

S1PORType por = AArch64.S1POR(regime);
constant integer bit_index = 4 * UInt(permissions.po_index);
bits(4) ppo = por<bit_index+3:bit_index>;

// Apply privileged overlay permissions
case ppo of
    when '0000' (pr,pw,px) = ('0','0','0'); // No access
    when '0001' (pr,pw,px) = ('1','0','0'); // Privileged read
    when '0010' (pr,pw,px) = ('0','0','1'); // Privileged execute
    when '0011' (pr,pw,px) = ('1','0','1'); // Privileged read and
    when '0100' (pr,pw,px) = ('0','1','0'); // Privileged write
    when '0101' (pr,pw,px) = ('1','1','0'); // Privileged read and
    when '0110' (pr,pw,px) = ('0','1','1'); // Privileged write and
    when '0111' (pr,pw,px) = ('1','1','1'); // Privileged read, write
    when '1xxx' (pr,pw,px) = ('0','0','0'); // Reserved

if HasUnprivileged(regime) then
    bits(4) upo = POR_EL0<bit_index+3:bit_index>;

// Apply unprivileged overlay permissions
case upo of
    when '0000' (ur,uw,ux) = ('0','0','0'); // No access
    when '0001' (ur,uw,ux) = ('1','0','0'); // Unprivileged read
    when '0010' (ur,uw,ux) = ('0','0','1'); // Unprivileged execute
    when '0011' (ur,uw,ux) = ('1','0','1'); // Unprivileged read and
    when '0100' (ur,uw,ux) = ('0','1','0'); // Unprivileged write
    when '0101' (ur,uw,ux) = ('1','1','0'); // Unprivileged read and

```

```

        when '0110' (ur,uw,ux) = ('0','1','1'); // Unprivileged write
        when '0111' (ur,uw,ux) = ('1','1','1'); // Unprivileged read
        when '1xxx' (ur,uw,ux) = ('0','0','0'); // Reserved

(r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

sloverlay_perms.or = r;
sloverlay_perms.ow = w;
sloverlay_perms.ox = x;

return sloverlay_perms;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S1TxSZFaults

```

// AArch64.S1TxSZFaults()
// =====
// Detect whether configuration of stage 1 TxSZ field generates a fault

boolean AArch64.S1TxSZFaults(Regime regime, S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(regime, walkparams.d128, walkparams.ds,
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (IsFeatureImplemented(FEAT_LVA) ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above";

    return FALSE;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2CheckPermissions

```

// AArch64.S2CheckPermissions()
// =====
// Verifies memory access with available permissions.

(FaultRecord, boolean) AArch64.S2CheckPermissions(FaultRecord fault_in,
                                                 S2TTWParams walkparam,
                                                 AccessDescriptor accd,
                                                 MemType memtype = walkstate.memattrs.memtype,
                                                 Permissions permissions = walkstate.permissions,
                                                 FaultRecord fault = fault_in,
                                                 S2AccessControls s2perms = AArch64.S2ComputePermissions(permissions)

bit r, w;
bit or, ow;

if accdesc.acctype == AccessType_TTW then
    r = s2perms.r_mmu;
    w = s2perms.w_mmu;
    or = s2perms.or_mmu;
    ow = s2perms.ow_mmu;
elseif accdesc.rcw then
    r = s2perms.r_rcw;
    w = s2perms.w_rcw;

```

```

        or = s2perms.or_rcw;
        ow = s2perms.ow_rcw;
    else
        r = s2perms.r;
        w = s2perms.w;
        or = s2perms.or;
        ow = s2perms.ow;

    if accdesc.acctype == AccessType_TTW then
        if (accdesc.toplevel && accdesc.varange == VARange_LOWER &&
            ((walkparams.tl0 == '1' && s2perms.toplevel0 == '0') ||
             (walkparams.tl1 == '1' && s2perms.<toplevel1,toplevel0>
              fault.statuscode = Fault_Permission;
              fault.toplevel = TRUE;
            elseif (accdesc.toplevel && accdesc.varange == VARange_UPPER &&
                    ((walkparams.tl1 == '1' && s2perms.toplevel1 == '0') ||
                     (walkparams.tl0 == '1' && s2perms.<toplevel1,toplevel0>
                      fault.statuscode = Fault_Permission;
                      fault.toplevel = TRUE;
// Stage 2 Permission fault due to AssuredOnly check
elseif (walkstate.s2assuredonly == '1' && !ipa.slassured) then
    fault.statuscode = Fault_Permission;
    fault.assuredonly = TRUE;

    elseif walkparams.ptw == '1' && memtype == MemType_Device then
        fault.statuscode = Fault_Permission;
    elseif s2perms.overlay && or == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay = TRUE;
    elseif accdesc.write && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay = TRUE;
// Prevent translation table walks in Non-secure space by Realm
elseif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != 0
    fault.statuscode = Fault_Permission;
elseif r == '0' then
    fault.statuscode = Fault_Permission;
elseif accdesc.write && w == '0' then
    fault.statuscode = Fault_Permission;
elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkpa
      permissions.s2dirty == '0') then
    fault.statuscode = Fault_Permission;
    fault.dirtybit = TRUE;

// Stage 2 Permission fault due to AssuredOnly check
elseif ((walkstate.s2assuredonly == '1' && !ipa.slassured) ||
        (walkstate.s2assuredonly != '1' && IsFeatureImplemented(FE
          VTCR_EL2.GCSH == '1' && accdesc.acctype == AccessType_GCS
        fault.statuscode = Fault_Permission;
        fault.assuredonly = TRUE;

    elseif accdesc.acctype == AccessType_IFETCH then
        if s2perms.overlay && s2perms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay = TRUE;
        elseif (memtype == MemType_Device &&
               ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Co
            fault.statuscode = Fault_Permission;

// Prevent execution from Non-secure space by Realm state

```

```

        elseif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace !
            fault.statuscode = Fault_Permission;
        elseif s2perms.x == '0' then
            fault.statuscode = Fault_Permission;

        elseif accdesc.acctype == AccessType_DC then
            if accdesc.cacheop == CacheOp_Invalidate then
                if !ELUsingAArch32(EL1) && s2perms.overlay && ow == '0' then
                    fault.statuscode = Fault_Permission;
                    fault.overlay = TRUE;
                if !ELUsingAArch32(EL1) && w == '0' then
                    fault.statuscode = Fault_Permission;
            elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif (walkparams.cmow == '1' &&
                    accdesc.opscope == CacheOpScope_PoC &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s2perms.overlay && ow == '0') then
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' then
                fault.statuscode = Fault_Permission;
            elseif (walkparams.cmow == '1' &&
                    accdesc.opscope == CacheOpScope_PoC &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    w == '0') then
                fault.statuscode = Fault_Permission;

        elseif accdesc.acctype == AccessType_IC then
            if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay)
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0";
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif walkparams.cmow == '1' && s2perms.overlay && ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif (!ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' &&
                    boolean IMPLEMENTATION_DEFINED "Permission fault on EL0");
                fault.statuscode = Fault_Permission;
            elseif walkparams.cmow == '1' && w == '0' then
                fault.statuscode = Fault_Permission;

        elseif accdesc.read && s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay = TRUE;
            fault.write = FALSE;
        elseif accdesc.write && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay = TRUE;
            fault.write = TRUE;
        elseif accdesc.read && r == '0' then
            fault.statuscode = Fault_Permission;
            fault.write = FALSE;
        elseif accdesc.write && w == '0' then
            fault.statuscode = Fault_Permission;
            fault.write = TRUE;
        elseif ((accdesc.tagaccess || accdesc.tagchecked) &&
                ipa.memattrs.tags == MemTag_AllocationTagged &&
                permissions.s2tag_na == '1' && S2DCacheEnabled()) then

```

```

        fault.statuscode = Fault_Permission;
        fault.tagaccess = TRUE;
        fault.write = accdesc.tagaccess && accdesc.write;
    elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams
            permissions.s2dirty == '0') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit = TRUE;
        fault.write = TRUE;

    // MRO* allows only RCW and MMU writes
    boolean mro;
    if s2perms.overlay then
        mro = (s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>)
    else
        mro = s2perms.<w,w_rcw,w_mmu> == '011';

    return (fault, mro);

```

## **Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2ComputePermissions**

```

// AArch64.S2ComputePermissions()
// =====
// Compute the overall stage 2 permissions.

S2AccessControls AArch64.S2ComputePermissions(Permissions permissions,
                                              AccessDescriptor accdesc)

S2AccessControls s2perms;

if walkparams.s2pie == '1' then
    s2perms = AArch64.S2IndirectBasePermissions(permissions, accdesc);
    s2perms.overlay = IsFeatureImplemented(FEAT_S2POE) && VTCR_EL2...
    if s2perms.overlay then
        s2overlay_perms = AArch64.S2OverlayPermissions(permissions,
                                                       s2perms.or      = s2overlay_perms.or;
                                                       s2perms.ow      = s2overlay_perms.ow;
                                                       s2perms.ox      = s2overlay_perms.ox;
                                                       s2perms.or_rcw  = s2overlay_perms.or_rcw;
                                                       s2perms.ow_rcw  = s2overlay_perms.ow_rcw;
                                                       s2perms.or_mmu  = s2overlay_perms.or_mmu;
                                                       s2perms.ow_mmu  = s2overlay_perms.ow_mmu;

        // Toplevel is applicable only when the effective S2 permission
        if ((s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>)
            s2perms.toplevel0 = s2perms.toplevel0 OR s2overlay_perms...
            s2perms.toplevel1 = s2perms.toplevel1 OR s2overlay_perms...

        else
            s2perms.toplevel0 = '0';
            s2perms.toplevel1 = '0';
    else
        s2perms = AArch64.S2DirectBasePermissions(permissions, accdesc);

    return s2perms;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2DirectBasePermissions

```
// AArch64.S2DirectBasePermissions()
// =====
// Computes the stage 2 direct base permissions.

S2AccessControls AArch64.S2DirectBasePermissions(Permissions permissions,
                                              AccessDescriptor accdesc)
{
    S2AccessControls s2perms;
    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;
    bit px, ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if accdesc.el == EL0 then ux else px;
    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r;
    s2perms.w_rcw = w;
    s2perms.r_mmu = r;
    s2perms.w_mmu = w;

    return s2perms;
}
```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2HasAlignmentFault

```
// AArch64.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data access
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccessDescriptor accdesc, boolean aligned)
{
    if accdesc.acctype == AccessType\_IFETCH then
        return FALSE;
    elsif IsFeatureImplemented(FEAT_MTE) && accdesc.tagaccess && accdesc.acctype == AccessType\_DCZero
        return (memattrs.memtype == MemType\_Device &&
                ConstrainUnpredictable\(Unpredictable\_DEVICETAGSTORE\) == TRUE);
    elsif accdesc.acctype == AccessType\_DCZero then
        return memattrs.memtype == MemType\_Device;
    else
        return memattrs.memtype == MemType\_Device && !aligned;
}
```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields
```

```

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel    = AArch64.S2StartLevel(walkparams);
    levels        = FINAL LEVEL - startlevel;
    granulebits   = TGxGranuleBits(walkparams.tgx);
    descsizeilog2 = 3;
    stride        = granulebits - descsizeilog2;

    // Input address size must at least be large enough to be resolved
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial
        + granulebits // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial

    // Can accomodate 1 more stride in the level + concatenation of up
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize         = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2IndirectBasePermissions

```

// AArch64.S2IndirectBasePermissions()
// =====
// Computes the stage 2 indirect base permissions.

S2AccessControls AArch64.S2IndirectBasePermissions(Permissions permissions,
                                                AccessDescriptor accdesc)
{
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2perms;

    bits(4) s2pi = permissions.s2pi;
    case s2pi of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0','0')
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0','0')
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0','0')
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0','0')
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0','0')
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','0','0')
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0','0')
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','1','0','0')
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','0','1','1')
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1','1')
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1','1')
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1','1')

    x = if accdesc.el == EL0 then ux else px;
}

```

```

// RCW and MMU read permissions.
(r_rcw, r_mmu) = (r, r);

// Stage 2 Top Level Permission Attributes.
case s2pi of
    when '0110' (toplevel0,toplevel1) = ('1','0');
    when '0011' (toplevel0,toplevel1) = ('0','1');
    when '0111' (toplevel0,toplevel1) = ('1','1');
    otherwise   (toplevel0,toplevel1) = ('0','0');

s2perms.r = r;
s2perms.w = w;
s2perms.x = x;
s2perms.r_rcw = r_rcw;
s2perms.r_mmu = r_mmu;
s2perms.w_rcw = w_rcw;
s2perms.w_mmu = w_mmu;
s2perms.toplevel0 = toplevel0;
s2perms.toplevel1 = toplevel1;

return s2perms;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2InvalidSL

```

// AArch64.S2InvalidSL()
// =====
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx\_4KB
            case walkparams.s12:walkparams.s10 of
                when '1x1' return TRUE;
                when '1lx' return TRUE;
                when '010' return AArch64.PAMax\(\) < 44;
                when '011' return !IsFeatureImplemented(FEAT_TTST);
                otherwise return FALSE;
        when TGx\_16KB
            case walkparams.s10 of
                when '11' return walkparams.ds == '0';
                when '10' return AArch64.PAMax\(\) < 42;
                otherwise return FALSE;
        when TGx\_64KB
            case walkparams.s10 of
                when '11' return TRUE;
                when '10' return AArch64.PAMax\(\) < 44;
                otherwise return FALSE;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2OverlayPermissions

```

// AArch64.S2OverlayPermissions()
// =====
// Computes the stage 2 overlay permissions.

S2AccessControls AArch64.S2OverlayPermissions(Permissions permissions,
    bit r, w;

```

```

bit r_rcw, w_rcw;
bit r_mmu, w_mmu;
bit px, ux;
bit toplevel0, toplevel1;
S2AccessControls s2overlay_perms;

constant integer index = 4 * UInt(permissions.s2po_index);
bits(4) s2po = S2POR_ELL<index+3 : index>;
case s2po of
    when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0','0')
    when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0','0')
    when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
    when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
    when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0','0')
    when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0','0')
    when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
    when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','1','1')
    when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0','0')
    when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','0','0')
    when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0','0')
    when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0','0')
    when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','0','1','1')
    when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1','1')
    when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','0','1','1')
    when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1','1')

x = if accdesc.el == ELO then ux else px;

// RCW and MMU read permissions.
(r_rcw, r_mmu) = (r, r);

// Stage 2 Top Level Permission Attributes.
case s2po of
    when '0110' (toplevel0,toplevel1) = ('1','0');
    when '0011' (toplevel0,toplevel1) = ('0','1');
    when '0111' (toplevel0,toplevel1) = ('1','1');
    otherwise (toplevel0,toplevel1) = ('0','0');

s2overlay_perms.or = r;
s2overlay_perms.ow = w;
s2overlay_perms.ox = x;
s2overlay_perms.or_rcw = r_rcw;
s2overlay_perms.ow_rcw = w_rcw;
s2overlay_perms.or_mmu = r_mmu;
s2overlay_perms.ow_mmu = w_mmu;
s2overlay_perms.toplevel0 = toplevel0;
s2overlay_perms.toplevel1 = toplevel1;

return s2overlay_perms;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.S2TxSZFaults

```

// AArch64.S2TxSZFaults()
// =====
// Detect whether configuration of stage 2 TxSZ field generates a fault

boolean AArch64.S2TxSZFaults(S2TTWParams walkparams, boolean s1aarch64)

```

```
mintxsz = AArch64.S2MinTxSz(walkparams.d128, walkparams.ds, walkpar  
maxtxsz = AArch64.MaxTxSz(walkparams.tgx);  
  
if UInt(walkparams.txsz) < mintxsz then  
    return (IsFeatureImplemented(FEAT_LPA) ||  
            boolean IMPLEMENTATION_DEFINED "Fault" on TxSz value bel  
if UInt(walkparams.txsz) > maxtxsz then  
    return boolean IMPLEMENTATION_DEFINED "Fault" on TxSz value abov  
  
return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/ AArch64.VAIsOutOfRange

```

// AArch64.VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted

boolean AArch64.VAIsOutOfRange(bits(64) va_in, AccessType acctype,
                               Regime regime, S1TTWParams walkparams)
bits(64) va = va_in;

constant integer addrtop = AArch64.AddrTop(walkparams.tbid, acctype);

// If the VA has a Logical Address Tag then the bits holding the Logical
// ignored when checking if the address is out of range.
if walkparams.mtx == '1' then
    va<59:56> = if AArch64.GetVARange(va) == VARange UPPER then '11111111111111111111111111111111
    else '00000000000000000000000000000000;

// Input Address size
constant integer iasize = AArch64.IASIZE(walkparams.txsz);

// The min value of TxSZ can be 8, with LVA3 implemented.
// If TxSZ is set to 8 iasize becomes 64 - 8 = 56
// If tbi is also set, addrtop becomes 55
// Then the return statements check va<56:55>
// The check here is to guard against this corner case.
if addrtop < iasize then
    return FALSE;

if HasUnprivileged(regime) then
    if AArch64.GetVARange(va) == VARange LOWER then
        return !IsZero(va<addrtop:iasize>);
    else
        return !IsOnes(va<addrtop:iasize>);
else
    return !IsZero(va<addrtop:iasize>);

```

## Library pseudocode for aarch64/translation/vmsa\_memattr/AArch64.S2ApplyFWBMemAttrs

```


MemoryAttributes memattrs;
s2_attr = descriptor<5:2>;
s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<5:2>;
s2_fnxs = descriptor<11>;

if s2_attr<2> == '0' then           // S2 Device, S1 any
    s2_device = DecodeDevice(s2_attr<1:0>);
    memattrs.memtype = MemType_Device;
    if s1_memattrs.memtype == MemType_Device then
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
    else
        memattrs.device = s2_device;

    memattrs.xs = s1_memattrs.xs;

elsif s2_attr<1:0> == '11' then      // S2 attr = S1 attr
    memattrs = s1_memattrs;

elsif s2_attr<1:0> == '10' then      // Force writeback
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = MemAttr_WB;
    memattrs.outer.attrs = MemAttr_WB;

    if (s1_memattrs.memtype == MemType_Normal &&
        s1_memattrs.inner.attrs != MemAttr_NC) then
        memattrs.inner.hints = s1_memattrs.inner.hints;
        memattrs.inner.transient = s1_memattrs.inner.transient;
    else
        memattrs.inner.hints = MemHint_RWA;
        memattrs.inner.transient = FALSE;

    if (s1_memattrs.memtype == MemType_Normal &&
        s1_memattrs.outer.attrs != MemAttr_NC) then
        memattrs.outer.hints = s1_memattrs.outer.hints;
        memattrs.outer.transient = s1_memattrs.outer.transient;
    else
        memattrs.outer.hints = MemHint_RWA;
        memattrs.outer.transient = FALSE;

    memattrs.xs = '0';

else                                // Non-cacheable unless S1 is device
    if s1_memattrs.memtype == MemType_Device then
        memattrs = s1_memattrs;
    else
        MemAttrHints cacheability_attr;
        cacheability_attr.attrs = MemAttr_NC;

        memattrs.memtype = MemType_Normal;
        memattrs.inner = cacheability_attr;
        memattrs.outer = cacheability_attr;

    memattrs.xs = s1_memattrs.xs;

s2_shareability = DecodeShareability(s2_sh);
memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability);
memattrs.tags     = S2MemTagType(memattrs, s1_memattrs.tags);
memattrs.notagaccess = (s2_attr<3:1> == '111' && memattrs.tags == 0);

if s2_fnxs == '1' then


```

```

        memattrs.xs = '0';

        memattrs.shareability = EffectiveShareability(memattrs);
        return memattrs;
    }
}

```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/ AArch64.GetS1TLBContext

```

// AArch64.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLE

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bit
TLBContext tlbcontext;

case regime of
    when Regime EL3 tlbcontext = AArch64.TLBContextEL3(ss, va, tg)
    when Regime EL2 tlbcontext = AArch64.TLBContextEL2(ss, va, tg)
    when Regime EL20 tlbcontext = AArch64.TLBContextEL20(ss, va, tg)
    when Regime EL10 tlbcontext = AArch64.TLBContextEL10(ss, va, tg)

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stale
    tlbcontext.includes_s2 = FALSE;
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
}

```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/ AArch64.GetS2TLBContext

```

// AArch64.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLE

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TLBContext
assert EL2Enabled());

    tlbcontext.tlbcontext;
    tlbcontext.ss = ss;
    tlbcontext.regime = Regime EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid = VMID[];;
    tlbcontext.tg = tg;
    tlbcontext.ia = ZeroExtend(ipa.address, 64);
    if IsFeatureImplemented(FEAT_TTCNP) then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2.CNP
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    // This may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
}

```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/ AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime to match a

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg,
TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID[];;
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else T
if TCR_EL1.AS == '0' then
    tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg       = tg;
    tlbcontext.ia       = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/ AArch64.TLBContextEL2

```
// AArch64.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match a

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg,
TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.tg       = tg;
    tlbcontext.ia       = va;
    tlbcontext.cnp     = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/ AArch64.TLBContextEL20

```
// AArch64.TLBContextEL20()
// =====
// Gather translation context for accesses under EL20 regime to match a

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg,
TLBContext tlbcontext;
```

```

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime_EL20;
    tlbcontext.asid   = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else T
    if TCR_EL2.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL2.CnP;
        else
            tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;

```

### **Library pseudocode for aarch64/translation/vmsa\_tlbcontext/ AArch64.TLBContextEL3**

```

// AArch64.TLBContextEL3()
// =====
// Gather translation context for accesses under EL3 regime to match ag

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime_EL3;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp     = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_1

    return tlbcontext;

```

### **Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.FullTranslate**

```

// AArch64.FullTranslate()
// =====
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccessDescriptor accdesc)
    Regime regime = TranslationRegime(accdesc.el);
    FaultRecord fault = NoFault(accdesc);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(va, fault);

    if accdesc.ss == SS_Realm then
        assert EL2Enabled();

```

```

if regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
    AddressDescriptor pa;
    (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, aligned);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(va, fault);
    else
        return pa;
else
    return ipa;

```

### Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.MemSwapTableDesc

```

// AArch64.MemSwapTableDesc()
// =====
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(N)) AArch64.MemSwapTableDesc(FaultRecord fault_in, b
bits(N) new_desc, bit e
AccessDescriptor descaccess,
AddressDescriptor descaddr);

FaultRecord fault = fault_in;
boolean iswrite;

if IsFeatureImplemented(FEAT_RME) then
    fault.gpcf = GranuleProtectionCheck(descpaddr, descaccess);
    if fault.gpcf.gpf != GPCF_None then
        fault.statuscode = Fault_GPCFOnWalk;
        fault.paddress = descpaddr.paddress;
        fault.gpcfs2walk = fault.secondstage;
        return (fault, bits(N) UNKNOWN);

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    bits(N) mem_desc;
    PhysMemRetStatus memstatus;
    (memstatus, mem_desc) = PhysMemRead(descpaddr, N DIV 8, descaccess);

    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if mem_desc == prev_desc then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc)
        memstatus = PhysMemWrite(descpaddr, N DIV 8, descaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, fault);

```

```

        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;

        return (fault, mem_desc);
    
```

## Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.S1DisabledOutput

```

// AArch64.S1DisabledOutput()
// =====
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord f
                                                               bits(64) va_i
                                                               boolean align)

bits(64) va = va_in;
walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);
FaultRecord fault = fault_in;

// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

// Output Address
FullAddress oa;
oa.address = va<55:0>;
case accdesc.ss of
    when SS_Secure      oa.paspace = PAS_Secure;
    when SS_NonSecure   oa.paspace = PAS_NonSecure;
    when SS_Root        oa.paspace = PAS_Root;
    when SS_Realm       oa.paspace = PAS_Realm;

MemoryAttributes memattrs;
if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
    MemAttrHints default_cacheability;
    default_cacheability.attrs      = MemAttr_WB;
    default_cacheability.hints     = MemHint_RWA;
    default_cacheability.transient = FALSE;

    memattrs.memtype      = MemType_Normal;
    memattrs.outer        = default_cacheability;
    memattrs.inner        = default_cacheability;
    memattrs.shareability = Shareability_NSH;
    if walkparams.dct == '1' then
        memattrs.tags      = MemTag_AllocationTagged;
    elseif walkparams.mtx == '1' then
        memattrs.tags      = MemTag_CanonicallyTagged;
    else
        memattrs.tags      = MemTag_Untagged;
    memattrs.xs           = '0';

elseif accdesc.acctype == AccessType_IFETCH then
    MemAttrHints i_cache_attr;
    if AArch64.S1ICacheEnabled(regime) then
        i_cache_attr.attrs      = MemAttr_WT;
        i_cache_attr.hints      = MemHint_RA;
    
```

```

        i_cache_attr.transient = FALSE;
    else
        i_cache_attr.attrs      = MemAttr_NC;

    memattrs.memtype       = MemType_Normal;
    memattrs.outer          = i_cache_attr;
    memattrs.inner          = i_cache_attr;
    memattrs.shareability   = Shareability_OSH;
    memattrs.tags            = MemTag_Untagged;
    memattrs.xs              = '1';

else
    memattrs.memtype       = MemType_Device;
    memattrs.device          = DeviceType_nGnRnE;
    memattrs.shareability   = Shareability_OSH;
    if walkparams.mtx == '1' then
        memattrs.tags = MemTag_CanonicallyTagged;
    else
        memattrs.tags = MemTag_Untagged;
    memattrs.xs              = '1';
memattrs.notagaccess = FALSE;

if walkparams.mtx == '1' && walkparams.tbi == '0' && accdesc.acctype
    // For the purpose of the checks in this function, the MTE tag
    va<59:56> = if HasUnprivileged(regime) then Replicate(va<55>, 4

fault.level = 0;
constant integer addrtop = AArch64.AddrTop(walkparams.tbid, accdesc);
constant integer pamax = AArch64.PAMax();

if !IsZero(va<addrtop:pamax>) then
    fault.statuscode = Fault_AddressSize;
elseif AArch64.S1HasAlignmentFault(accdesc, aligned, walkparams.ntls)
    fault.statuscode = Fault_Alignment;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);
else
    ipa = CreateAddressDescriptor(va_in, oa, memattrs);
    ipa.mecid = AArch64.S1DisabledOutputMECID(walkparams, regime, i
    return (fault, ipa);

```

## Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.S1Translate

```

// AArch64.S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_
                                         bits(64) va, boolean
                                         AccessDescriptor a
                                         FaultRecord fault = fault_in;
                                         // Prepare fault fields in case a fault is detected
                                         fault.secondstage = FALSE;
                                         fault.s2fs1walk = FALSE;

                                         if !AArch64.S1Enabled(regime, accdesc.acctype) then
                                             return AArch64.S1DisabledOutput(fault, regime, va, accdesc, ali

```

```

walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);

constant integer s1mintxsz = AArch64.S1MinTxSz(regime, walkparams.ds,
                                                walkparams.ds, walkparams);
constant integer s1maxtxsz = AArch64.MaxTxSz(walkparams.tgx);
if AArch64.S1TxSZFaults(regime, walkparams) then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);
elsif UInt(walkparams.txsz) < s1mintxsz then
    walkparams.txsz = s1mintxsz<5:0>;
elsif UInt(walkparams.txsz) > s1maxtxsz then
    walkparams.txsz = s1maxtxsz<5:0>;

if AArch64.VAIsOutOfRange(va, accdesc.acctype, regime, walkparams)
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

if accdesc.el == EL0 && walkparams.e0pd == '1' then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

if (IsFeatureImplemented(FEAT_TME) && accdesc.el == EL0 && walkparams.transactional) then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

if (IsFeatureImplemented(FEAT_SVE) && accdesc.el == EL0 && walkparams.nonfault && accdesc.contiguous) ||
   (accdesc.firstfault && !accdesc.first && !accdesc.contiguous))
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descipaddr;
TTWState walkstate;
bits(128) descriptor;
bits(128) new_desc;
bits(128) mem_desc;
repeat
    if walkparams.d128 == '1' then
        (fault, descipaddr, walkstate, descriptor) = AArch64.S1Walk
    else
        (fault, descipaddr, walkstate, descriptor<63:0>) = AArch64.

            descriptor<127:64> = Zeros(64);
if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

if accdesc.acctype == AccessType_IFETCH then
    // Flag the fetched instruction is from a guarded page
    SetInGuardedPage(walkstate.guardedpage == '1');

if AArch64.S1HasAlignmentFault(accdesc, aligned, walkparams.nth)

```

```

            walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;

    if fault.statuscode == Fault_None then
        fault = AArch64.S1CheckPermissions(fault, regime, walkstate);

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(f
        // Set descriptor AF bit
        new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permis
        // will already reflect a configuration permitting writes.
        // The update of the descriptor occurs only if the descriptor b
        // memory do not reflect that and the access instigates a write.

    if (AArch64.SettingDirtyStatePermitted(fault) &&
        walkparams.ha == '1' &&
        walkparams.hd == '1' &&
        (walkparams.pie == '1' || descriptor<51> == '1') &&
        accdesc.write &&
        !(accdesc.acctype IN {AccessType_AT, AccessType_IC, Acc
        // Clear descriptor AP[2]/nDirty bit permitting stage 1 wri
    new_desc<7> = '0';

    // Either the access flag was clear or AP[2]/nDirty is set
    if new_desc != descriptor then
        AddressDescriptor descpaddr;
        descaccess = CreateAccDescTTEUpdate(accdesc);
        if regime == Regime_EL10 && EL2Enabled() then
            FaultRecord s2fault;
            s1aarch64 = TRUE;
            s2aligned = TRUE;
            (s2fault, descpaddr) = AArch64.S2Translate(fault, desc
                descaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN);

        else
            descpaddr = descipaddr;
            if walkparams.d128 == '1' then
                (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, des
                    walkparams
            else
                (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(faul
                    new_
                    desc
                mem_desc<127:64> = Zeros(64);

            until new_desc == descriptor || mem_desc == new_desc;

            if fault.statuscode != Fault_None then
                return (fault, AddressDescriptor UNKNOWN);

            // Output Address
            oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
            MemoryAttributes memattrs;
            if (accdesc.acctype == AccessType_IFETCH &&
                (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICa

```

```

        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    elseif (accdesc.acctype != AccessType_IFETCH && !AArch64.S1DCacheEna
            walkstate.memattrs.memtype == MemType_Normal) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;

        // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
        // on the Tagged attribute
        if (IsFeatureImplemented(FEAT_MTE2) &&
            walkstate.memattrs.tags == MemTag_AllocationTagged &&
            !ConstrainUnpredictableBool(Unpredictable_S1CTAGGED)) then
            memattrs.tags = MemTag_Untagged;
        else
            memattrs = walkstate.memattrs;

        // Shareability value of stage 1 translation subject to stage 2 is
        // to be either effective value or descriptor value
        if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability"))
            memattrs.shareability = walkstate.memattrs.shareability;
        else
            memattrs.shareability = EffectiveShareability(memattrs);

        if accdesc.ls64 && memattrs.memtype == MemType_Normal then
            if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs !=
                fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

        ipa = CreateAddressDescriptor(va, oa, memattrs);
        ipa.slassured = walkstate.slassured;
        varange = AArch64.GetVARange(va);
        ipa.mecid = AArch64.S1OutputMECID(walkparams, regime, varange, ipa.
                                            descriptor);
        return (fault, ipa);
    
```

## Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.S2Translate

```

// AArch64.S2Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault,
                                                    boolean s1aarch64,
                                                    AccessDescriptor a

walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspac
FaultRecord fault = fault_in;
boolean s2fs1mro;

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None; // Ignore any faults from stage 1
fault.secondstage = TRUE;
fault.s2fs1walk = accdesc.acctype == AccessType_TTW;
fault.ipaddress = ipa.paddress;
    
```

```

if walkparams.vm != '1' then
    // Stage 2 translation is disabled
    return (fault, ipa);

constant integer s2mintxsz = AArch64.S2MinTxSz(walkparams.d128, walkparams.tgx, s1aarch64);
constant integer s2maxtxsz = AArch64.MaxTxSz(walkparams.tgx);
if AArch64.S2TxSZFaults(walkparams, s1aarch64) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);
elsif UInt(walkparams.txsz) < s2mintxsz then
    walkparams.txsz = s2mintxsz<5:0>;
elsif UInt(walkparams.txsz) > s2maxtxsz then
    walkparams.txsz = s2maxtxsz<5:0>;

if (walkparams.d128 == '0' &&
    (AArch64.S2InvalidSL(walkparams) || AArch64.S2InconsistentSL(walkparams))) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

if AArch64.IPAOutOfRange(ipa.paddress.address, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddr;
TTWState walkstate;
bits(128) descriptor;
bits(128) new_desc;
bits(128) mem_desc;
repeat
    if walkparams.d128 == '1' then
        (fault, descaddr, walkstate, descriptor) = AArch64.S2Walk(ipa.paddress.address, walkparams);
    else
        (fault, descaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(ipa.paddress.address, walkparams);

        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch64.S2HasAlignmentFault(accdesc, aligned, walkstate.memaddr) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode == Fault\_None then
        (fault, s2fslmro) = AArch64.S2CheckPermissions(fault, walkstate, accdesc);

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(accdesc) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permission
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor b

```

```

// memory do not reflect that and the access instigates a write.

if (AArch64.SettingDirtyStatePermitted(fault) &&
    walkparams.ha == '1' &&
    walkparams.hd == '1' &&
    (walkparams.s2pie == '1' || descriptor<51> == '1') &&
    accdesc.write &&
    !(accdesc.acctype IN {AccessType_AT, AccessType_IC, AccessType_TTW}) &&
    // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
    new_desc<7> = '1';

// Either the access flag was clear or S2AP[1]/Dirty is clear
if new_desc != descriptor then
    AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    if walkparams.d128 == '1' then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor,
                                                       walkparams,
                                                       descaddr);
    else
        (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault,
                                                               new_desc);

    mem_desc<127:64> = Zeros(64);

until new_desc == descriptor || mem_desc == new_desc;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

ipa_64 = ZeroExtend(ipa.paddress.address, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
MemoryAttributes s2_memattrs;
if ((accdesc.acctype == AccessType_TTW &&
    walkstate.memattrs.memtype == MemType_Device && walkparams.d128 == '1') ||
    (accdesc.acctype == AccessType_IFETCH &&
     (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.IDR == '1')) ||
    (accdesc.acctype != AccessType_IFETCH &&
     walkstate.memattrs.memtype == MemType_Normal && !S2DCacheEnabled)) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

if accdesc.ls64 && s2_memattrs.memtype == MemType_Normal then
    if s2_memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.attrs != MemAttr_NC
        fault.statuscode = Fault_Exclusive;
    return (fault, AddressDescriptor UNKNOWN);

s2aarch64 = TRUE;
MemoryAttributes memattrs;
if walkparams.fwb == '0' then
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
else
    memattrs = s2_memattrs;

pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
pa.s2fs1mro = s2fs1mro;

```

```
pa.mecid = AArch64.S2OutputMECID(walkparams, pa.paddress.paspace, c  
return (fault, pa);
```

### Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.SettingAccessFlagPermitted

```
// AArch64.SettingAccessFlagPermitted()  
// =====  
// Determine whether the access flag could be set by HW given the fault  
  
boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)  
    if fault.statuscode == Fault_None then  
        return TRUE;  
    elseif fault.statuscode IN {Fault_Alignment, Fault_Permission} then  
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);  
    else  
        return FALSE;
```

### Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.SettingDirtyStatePermitted

```
// AArch64.SettingDirtyStatePermitted()  
// =====  
// Determine whether the dirty state could be set by HW given the fault  
  
boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault)  
    if fault.statuscode == Fault_None then  
        return TRUE;  
    elseif fault.statuscode == Fault_Alignment then  
        return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);  
    else  
        return FALSE;
```

### Library pseudocode for aarch64/translation/vmsa\_translation/ AArch64.TranslateAddress

```
// AArch64.TranslateAddress()  
// =====  
// Main entry point for translating an address  
  
AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccessDescriptor  
                                         boolean aligned, integer size  
                                         if (SPESampleInFlight && !(accdesc.acctype IN {AccessType_IFETCH,  
                                         AccessType_SPE})) then  
                                         SPEStartCounter(SPECounterPosTranslationLatency);  
  
AddressDescriptor result = AArch64.FullTranslate(va, accdesc, align  
  
if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then  
    result.fault = AArch64.CheckDebug(va, accdesc, size);  
  
if (IsFeatureImplemented(FEAT_RME) && !IsFault(result) &&  
    (accdesc.acctype != AccessType_DC ||  
     boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations")  
    result.fault.gpcf = GranuleProtectionCheck(result, accdesc);
```

```

        if result.fault.gpcf.gpf != GPCF_None then
            result.fault.statuscode = Fault_GPCFOnOutput;
            result.fault.paddress    = result.paddress;

        if !IsFault(result) && accdesc.acctype == AccessType_IFETCH then
            result.fault = AArch64.CheckDebug(va, accdesc, size);

        if (SPESampleInFlight && !(accdesc.acctype IN {AccessType_IFETCH,
                                                       AccessType_SPE})) then
            SPEStopCounter(SPECounterPosTranslationLatency);

        // Update virtual address for abort functions
        result.vaddress = ZeroExtend(va, 64);

        return result;
    
```

### **Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.BlockDescSupported**

```

// AArch64.BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule
// and level

boolean AArch64.BlockDescSupported(bit d128, bit ds, TGx tgx, integer 1
case tgx of
    when TGx_4KB   return ((level == 0 && (ds == '1' || d128 == '1'))
                                level == 1 ||
                                level == 2);
    when TGx_16KB return ((level == 1 && (ds == '1' || d128 == '1'))
                                level == 2);
    when TGx_64KB return ((level == 1 && (d128 == '1' || AArch64.PA
                                level == 2));
return FALSE;
    
```

### **Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.BlocknTFaults**

```

// AArch64.BlocknTFaults()
// =====
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bit d128, bits(N) descriptor)
    bit nT;
    if !IsFeatureImplemented(FEAT_BBM) then
        return FALSE;
    nT = if d128 == '1' then descriptor<6> else descriptor<16>;
    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = (boolean IMPLEMENTATION_DEFINED
                    "BBM level 1 or 2 support nT bit causes Translation Fa
    return bbm_level IN {1, 2} && nT == '1' && nT_faults;
    
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =====
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, bit d128, integer level, bits(N) des
    if d128 == '1' then
        if (tgx == TGx_64KB && level == 1) || (tgx == TGx_4KB && level
            return '0'; // RES0
        else
            return descriptor<111>;
    // When using TGx 64KB and FEAT_LPA is implemented,
    // the Contiguous bit is RES0 for Block descriptors at level 1

    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0

    // When the effective value of TCR_ELx.DS is '1',
    // the Contiguous bit is RES0 for all the following:
    //     * For TGx 4KB, Block descriptors at level 0
    //     * For TGx 16KB, Block descriptors at level 1

    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0

    if tgx == TGx_4KB && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(N) descriptor, bit d128
                                              TGx tgx, integer level)
    if descriptor<0> == '0' then
        return DescriptorType_Invalid;
    elseif d128 == '1' then
        bits(2) skl = descriptor<110:109>;
        if tgx IN {TGx_16KB, TGx_64KB} && UInt(skl) == 3 then
            return DescriptorType_Invalid;

        integer effective_level = level + UInt(skl);
        if effective_level > FINAL_LEVEL then
            return DescriptorType_Invalid;
        elseif effective_level == FINAL_LEVEL then
            return DescriptorType_Leaf;
        else
            return DescriptorType_Table;
    else
        if descriptor<1> == '1' then
```

```

        if level == FINAL_LEVEL then
            return DescriptorType_Leaf;
        else
            return DescriptorType_Table;
    elseif descriptor<1> == '0' then
        if AArch64.BlockDescSupported(d128, ds, tgx, level) then
            return DescriptorType_Leaf;
        else
            return DescriptorType_Invalid;
    Unreachable();

```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.S1ApplyOutputPerms

```

// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits
                                         Regime regime, S1TTWParams walkp
                                         Permissions permissions = permissions_in;

bits (4) pi_index;
if walkparams.pie == '1' then
    if walkparams.d128 == '1' then
        pi_index = descriptor<118:115>;
    else
        pi_index = descriptor<54:53>;descriptor<51>;descriptor<6>;
permissions.ppi      = Elem[walkparams.pir, UInt(pi_index), 4];
permissions.upi      = Elem[walkparams.pire0, UInt(pi_index), 4];
permissions.ndirty   = descriptor<7>;
else
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 ==
        permissions.ap<2:1> = descriptor<7>:'0';
        permissions.pxn     = descriptor<54>;
    elseif HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn     = descriptor<54>;
        permissions.pxn     = descriptor<53>;
    else
        permissions.ap<2:1> = descriptor<7>:'1';
        permissions.xn      = descriptor<54>;
// Descriptors marked with DBM set have the effective value of A
// This implies no Permission faults caused by lack of write pe
// reported, and the Dirty bit can be set.
if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<5
    permissions.ap<2> = '0';

if IsFeatureImplemented(FEAT_S1POE) then
    if walkparams.d128 == '1' then
        permissions.po_index = descriptor<124:121>;
    else
        permissions.po_index = '0':descriptor<62:60>;

return permissions;

```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime\_EL10 && EL2Enabled() && walkparams.nvl == '1' then
        if walkparams.d128 == '1' then
            ap_table = descriptor<126>:'0';
            pxn_table = descriptor<124>;
        else
            ap_table = descriptor<62>:'0';
            pxn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elseif HasUnprivileged(regime) then
        if walkparams.d128 == '1' then
            ap_table = descriptor<126:125>;
            uxn_table = descriptor<124>;
            pxn_table = descriptor<123>;
        else
            ap_table = descriptor<62:61>;
            uxn_table = descriptor<60>;
            pxn_table = descriptor<59>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        if walkparams.d128 == '1' then
            ap_table = descriptor<126>:'0';
            xn_table = descriptor<124>;
        else
            ap_table = descriptor<62>:'0';
            xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/ AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(N) descriptor, S2TTWParams
    Permissions permissions;
```

```

bits(4) s2pi_index;
if walkparams.s2pie == '1' then
    if walkparams.d128 == '1' then
        s2pi_index = descriptor<118:115>;
    else
        s2pi_index = descriptor<54:53,51,6>;
permissions.s2pi = Elem[walkparams.s2pir, UInt(s2pi_index), 4];
permissions.s2dirty = descriptor<7>;
else
    permissions.s2ap = descriptor<7:6>;
    if walkparams.d128 == '1' then
        permissions.s2xn = descriptor<118>;
    else
        permissions.s2xn = descriptor<54>;
if IsFeatureImplemented(FEAT_XNX) then
    if walkparams.d128 == '1' then
        permissions.s2xnx = descriptor<117>;
    else
        permissions.s2xnx = descriptor<53>;
else
    permissions.s2xnx = '0';
// Descriptors marked with DBM set have the effective value of S
// This implies no Permission faults caused by lack of write permission
// reported, and the Dirty bit can be set.
bit desc_dbm;
if walkparams.d128 == '1' then
    desc_dbm = descriptor<115>;
else
    desc_dbm = descriptor<51>;
if walkparams.ha == '1' && walkparams.hd == '1' && desc_dbm ==
    permissions.s2ap<1> = '1';
if IsFeatureImplemented(FEAT_S2POE) then
    if walkparams.d128 == '1' then
        permissions.s2po_index = descriptor<124:121>;
    else
        permissions.s2po_index = descriptor<62:59>;
return permissions;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.S1InitialTTWState

```

// AArch64.S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va,
                                    SecurityState ss)
TTWState      walkstate;
FullAddress   tablebase;
Permissions   permissions;
bits(128)       ttbr;

ttbr           = AArch64.S1TTBR(regime, va);
case ss of
    when SS_Secure    tablebase.paspace = PAS_Secure;
    when SS_NonSecure  tablebase.paspace = PAS_NonSecure;

```

```

when SS_Root      tablebase.paspace = PAS_Root;
when SS_Realm     tablebase.paspace = PAS_Realm;

tablebase.address = AArch64.S1TTBaseAddress(walkparams, regime, ttb

permissions.ap_table = '00';
if HasUnprivileged(regime) then
    permissions.uxn_table = '0';
    permissions.pxn_table = '0';
else
    permissions.xn_table = '0';

walkstate.baseaddress = tablebase;
walkstate.level      = AArch64.S1StartLevel(walkparams);
walkstate.istable    = TRUE;
// In regimes that support global and non-global translations, trans
// table entries from lookup levels other than the final level of lo
// are treated as being non-global
walkstate.nG          = if HasUnprivileged(regime) then '1' else '0'
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgrn
walkstate.permissions = permissions;
if (regime == Regime_EL10 && EL2Enabled() && (HCR_EL2.VM == '1' ||
   if ((AArch64.GetVARange(va) == VARange_LOWER && VTCR_EL2.TL0 ==
        (AArch64.GetVARange(va) == VARange_UPPER && VTCR_EL2.TL1 ==
        walkstate.slassured = TRUE;
    else
        walkstate.slassured = FALSE;
else
    walkstate.slassured = FALSE;
walkstate.disch = walkparams.disch;

return walkstate;

```

### Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.S1NextWalkStateLeaf

```

// AArch64.S1NextWalkStateLeaf()
// =====
// Decode stage 1 page or block descriptor as output to this stage of t

TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate, boolean s2f
                                         SecurityState ss, S1TTWParams walk
                                         TTWState nextstate;
                                         FullAddress baseaddress;
                                         baseaddress.address = AArch64.LeafBase(descriptor, walkparams.d128,
                                         walkparams.ds,
                                         walkparams.tgx, currentstate

if currentstate.baseaddress.paspace == PAS_Secure then
    // Determine PA space of the block from NS bit
    bit ns;
    ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<11,127>;
    baseaddress.paspace = if ns == '0' then PAS_Secure else PAS_NonSecure;
elseif currentstate.baseaddress.paspace == PAS_Root then
    // Determine PA space of the block from NSE and NS bits
    bit nse;
    bit ns;
    <nse,ns> = if walkparams.d128 == '1' then descriptor<11,127> else descriptor<11,127>;
    baseaddress.paspace = if nse == '1' then PAS_Root else PAS_NonRoot;
else
    baseaddress.paspace = PAS_NonRoot;

```

```

baseaddress.paspace = DecodePASpace(nse, ns);

// If Secure state is not implemented, but RME is,
// force Secure space accesses to Non-secure space
if baseaddress.paspace == PAS\_Secure && !HaveSecureState() then
    baseaddress.paspace = PAS\_NonSecure;

elseif (currentstate.baseaddress.paspace == PAS\_Realm &&
       regime IN {Regime\_EL2, Regime\_EL20}) then
    // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
    bit ns;
    ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<128>;
    baseaddress.paspace = if ns == '0' then PAS\_Realm else PAS\_NonSecure;
elseif currentstate.baseaddress.paspace == PAS\_Realm then
    // Realm EL1&0 regime does not have a stage 1 NS bit
    baseaddress.paspace = PAS\_Realm;
else
    baseaddress.paspace = PAS\_NonSecure;

nextstate.istable      = FALSE;
nextstate.level        = currentstate.level;
nextstate.baseaddress = baseaddress;

bits(4) attrindx;
if walkparams.aie == '1' then
    if walkparams.d128 == '1' then
        attrindx = descriptor<5:2>;
    else
        attrindx = descriptor<59,4:2>;
else
    attrindx = '0':descriptor<4:2>;

bits(2) sh;
if walkparams.d128 == '1' then
    sh = descriptor<9:8>;
elseif walkparams.ds == '1' then
    sh = walkparams.sh;
else
    sh = descriptor<9:8>;
attr = AArch64.MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.slaarch64);
slaarch64 = TRUE;

nextstate.memattrs     = S1DecodeMemAttrs(attr, sh, slaarch64, walkparams.mair2);
nextstate.permissions  = AArch64.S1ApplyOutputPerms(currentstate.permissions, descriptor, regi
bit protectedbit;
if walkparams.d128 == '1' then
    protectedbit = descriptor<114>;
else
    protectedbit = if walkparams.pnch == '1' then descriptor<52> else descriptor<53>;
if (currentstate.slassured && s2fs1mro && protectedbit == '1') then
    nextstate.slassured = TRUE;
else
    nextstate.slassured = FALSE;

if walkparams.pnch == '1' || currentstate.disch == '1' then
    nextstate.contiguous = '0';
else
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, wa
currentstate.level

```

```

if !HasUnprivileged(regime) then
    nextstate.nG = '0';
elsif ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure
    // In Secure state, a translation must be treated as non-global
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table
    nextstate.nG = '1';
else
    nextstate.nG = descriptor<11>;
if walkparams.d128 == '1' then
    nextstate.guardedpage = descriptor<113>;
else
    nextstate.guardedpage = descriptor<50>;
return nextstate;

```

### Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.S1NextWalkStateTable

```

// AArch64.S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, boolean s2_is_secure,
                                         S1TTWParams walkparams, bits(N) currentstage_nstable)
{
    TTWState      nextstate;
    FullAddress   tablebase;
    bits(2)  skl = if walkparams.d128 == '1' then descriptor<110:109> else
                    descriptor<111:109>;
    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128,
                                                skl, walkparams.ds,
                                                walkparams.tgx);
    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        bit nstable;
        nstable = if walkparams.d128 == '1' then descriptor<127> else
                   descriptor<126>;
        tablebase.paspace = if nstable == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable      = TRUE;
    nextstate.nG            = currentstate.nG;
    if walkparams.d128 == '1' then
        nextstate.level    = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level    = currentstate.level + 1;
    nextstate.baseaddress  = tablebase;
    nextstate.memattrs     = currentstate.memattrs;
    if walkparams.hpd == '0' && walkparams.pie == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate,
                                                       regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;
    bit protectedbit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor<114>;
}

```

```

    else
        protectedbit = if walkparams.pnch == '1' then descriptor<52> el
    if (currentstate.slassured && s2fs1mro && protectedbit == '1') then
        nextstate.slassured = TRUE;
    else
        nextstate.slassured = FALSE;
    nextstate.disch = if walkparams.d128 == '1' then descriptor<112> el

    return nextstate;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S1Walk

```

// AArch64.S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S1Walk(Fault\_S1T1
bits
Access
integer
FaultRecord fault = fault_in;
boolean slaarch64;
boolean aligned;

if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, accde
constant integer startlevel = walkstate.level;

bits(N) descriptor;
AddressDescriptor walkaddress;
bits(2) skl = '00';
walkaddress.vaddress = va;
walkaddress.mecid = AArch64.TTWalkMECID(walkparams.emec, regime, ac

if !AArch64.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is
// to be either effective value or descriptor value
if (regime == Regime\_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability"))
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkad

boolean s2fs1mro = FALSE;

DescriptorType desctype;

```

```


FullAddress descaddress = AArch64.S1SLTEntryAddress(walkstate.level,
                                                    walkstate.base)

// Detect Address Size Fault by Descriptor Address
if AArch64.OAOutOfRange(descaddress.address, walkparams.d128,
                        walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N))

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    walkaddress.slassured = walkstate.slassured;

    boolean toplevel = walkstate.level == startlevel;
    VARange varange = AArch64.GetVARange(va);
    AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange);
    FaultRecord s2fault;
    AddressDescriptor s2walkaddress;
    if regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        aligned   = TRUE;
        (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaccess,
                                                       walkparams.ps,
                                                       walkparams.tgx);

    if s2fault.statuscode != Fault_None then
        return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    s2fs1mro = s2walkaddress.s2fs1mro;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress,
                                           fault, N);
else
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaccess,
                                           fault, N);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

bits(N) new_descriptor;
repeat
    new_descriptor = descriptor;
    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ps,
                                              walkparams.tgx, walkstate);
    case desctype of
        when DescriptorType_Table
            walkstate = AArch64.S1NextWalkStateTable(walkstate,
                                                      regime, walkstate.base);
            skl = if walkparams.d128 == '1' then descriptor<110>;
            descaddress = AArch64.TTEEntryAddress(walkstate.level,
                                                   walkparams.ps,
                                                   walkparams.tgx,
                                                   walkstate.base);

// Detect Address Size Fault by Descriptor Address
if AArch64.OAOutOfRange(descaddress.address, walkparams.d128,
                        walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault_AddressSize;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N))


```

```

bits(N) UNKNOWN);

    if walkparams.haft == '1' then
        new_descriptor<10> = '1';
    if (walkparams.d128 == '1' && skl != '00' &&
        AArch64.BlocknTFaults(walkparams.d128, descriptor)
        fault.statuscode = Fault Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState
            bits(N) UNKNOWN);
    when DescriptorType_Leaf
        walkstate = AArch64.S1NextWalkStateLeaf(walkstate,
            regime, acc
            descriptor)

    when DescriptorType_Invalid
        fault.statuscode = Fault Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState
            bits(N) UNKNOWN);
    otherwise
        Unreachable();

if new_descriptor != descriptor then
    AddressDescriptor descpaddr;
    AccessDescriptor descaccess = CreateAccDescTTEUpdate(ac
    if regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        aligned = TRUE;
        (s2fault, descpaddr) = AArch64.S2Translate(fault, w
            s1aarch64
            descaccess

        if s2fault.statuscode != Fault None then
            return (s2fault, AddressDescriptor UNKNOWN,
                TTWState UNKNOWN, bits(N) UNKNOWN);
    else
        descpaddr = walkaddress;

    (fault, descriptor) = AArch64.MemSwapTableDesc(fault, d
        walkparams
        descpaddr

    if fault.statuscode != Fault None then
        return (fault, AddressDescriptor UNKNOWN,
            TTWState UNKNOWN, bits(N) UNKNOWN);
    until new_descriptor == descriptor;
until desctype == DescriptorType_Leaf;

FullAddress oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate.level);

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkstate.level)) then
    fault.statuscode = Fault Translation;
elseif (desctype == DescriptorType_Leaf && walkstate.level < FINAL_LEVEL)
    AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
    fault.statuscode = Fault Translation;
elseif AArch64.S1AMECFault(walkparams, walkstate.baseaddress.paspace)
    fault.statuscode = Fault Translation;
// Detect Address Size Fault by final output
elseif AArch64.OAOutOfRange(oa.address, walkparams.d128,
    walkparams.ps, walkparams.tgx) then
    fault.statuscode = Fault AddressSize;

```

```

// Check descriptor AF bit
elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
      !accdesc.acctype IN {AccessType_DC, AccessType_IC} &&
      !boolean IMPLEMENTATION_DEFINED "Generate access flag fault
      fault.statuscode = Fault_AccessFlag;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bit)

return (fault, walkaddress, walkstate, descriptor);

```

### Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.S2InitialTTWState

```

// AArch64.S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkpa
    TTWState walkstate;
    FullAddress tablebase;
    bits(128) ttbr;

    ttbr = ZeroExtend(VTTBR_EL2, 128);
    case ss of
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Realm      tablebase.paspace = PAS_Realm;
    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.p

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs   = WalkMemAttrs(walkparams.sh, walkparams.irgr

    return walkstate;

```

### Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.S2NextWalkStateLeaf

```

// AArch64.S2NextWalkStateLeaf()
// =====
// Decode stage 2 page or block descriptor as output to this stage of t

TTWState AArch64.S2NextWalkStateLeaf(TTWState currentstate, SecuritySta
    S2TTWParams walkparams, AddressDes
    bits(N) descriptor)
    TTWState nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.
    elsif ss == SS_Realm then
        bit ns;
        ns = if walkparams.d128 == '1' then descriptor<127> else descri
        baseaddress.paspace = if ns == '1' then PAS_NonSecure else PAS_
    else
        baseaddress.paspace = PAS_NonSecure;

```

```

baseaddress.address      = AArch64.LeafBase(descriptor, walkparams.d128,
                                         walkparams.tgx, currentstate.level);

nextstate.istable        = FALSE;
nextstate.level           = currentstate.level;
nextstate.baseaddress     = baseaddress;
nextstate.permissions     = AArch64.S2ApplyOutputPerms(descriptor, walkparams.ds);

s2_attr = descriptor<5:2>;
s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor.s2_sh;
s2_fnxs = descriptor<11>;
if walkparams.fwb == '1' then
    nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, walkparams.ds);
    if s2_attr<3:1> == '111' then
        nextstate.permissions.s2tag_na = '1';
    else
        nextstate.permissions.s2tag_na = '0';
else
    s2aarch64 = TRUE;
    nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
    // FnXS is used later to mask the XS value from stage 1
    nextstate.memattrs.xs = NOT s2_fnxs;
    if s2_attr == '0100' then
        nextstate.permissions.s2tag_na = '1';
    else
        nextstate.permissions.s2tag_na = '0';
nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.ds,
                                         currentstate.level, descriptor.s2assuredonly);

if walkparams.d128 == '1' then
    nextstate.s2assuredonly = descriptor<114>;
else
    nextstate.s2assuredonly = if walkparams.assuredonly == '1' then
                                descriptor.s2assuredonly
                            else
                                descriptor.s2assuredonly;

return nextstate;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.S2NextWalkStateTable

```

// AArch64.S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParam bits(N) descriptor)
{
    TTWState      nextstate;
    FullAddress  tablebase;
    bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else
                  descriptor.s2assuredonly;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128,
                                              skl, walkparams.ds,
                                              walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable      = TRUE;
    if walkparams.d128 == '1' then
        nextstate.level    = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level    = currentstate.level + 1;
}

```

```

nextstate.baseaddress = tablebase;
nextstate.memattrs   = currentstate.memattrs;

return nextstate;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S2Walk

```

// AArch64.S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S2Walk(FaultRecord, AddressDescriptor, TTWState, bits(N))

FaultRecord fault = fault_in;
ipa_64 = ZeroExtend(ipa.paddress.address, 64);

TTWState walkstate;
if accdesc.ss == SS\_Secure then
    walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress);
else
    walkstate = AArch64.S2InitialTTWState(accdesc.ss, walkparams);

constant integer startlevel = walkstate.level;

bits(N) descriptor;
AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
AddressDescriptor walkaddress;
bits(2) skl = '00';

walkaddress.vaddress = ipa.vaddress;
walkaddress.mecid = AArch64.TTWalkMECID(walkparams.emec, Regime\_EL1);

if !S2DCacheEnabled() then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

walkaddress.memattrs.shareability = EffectiveShareability(walkaddress);

DescriptorType desctype;

// Initial lookup might index into concatenated tables
FullAddress descaddress = AArch64.S2SLTEntryAddress(walkparams, ipa);
walkstate.baseaddress = descaddress.address;

// Detect Address Size Fault by Descriptor Address
if AArch64.OAOutOfRange(descaddress.address, walkparams.d128, walkparams.tgx) then
    fault.statuscode = Fault\_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bit);

```

```

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress);

    if fault.statuscode != Fault None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN);

    bits(N) new_descriptor;
    repeat
        new_descriptor = descriptor;
        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams);
        walkparams.tgx, walkstate.base = walkparams.tgx, walkstate.base;

        case desctype of
            when DescriptorType_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate,
                    skl = if walkparams.d128 == '1' then descriptor<110>;
                    descaddress = AArch64.TTEEntryAddress(walkstate.level,
                        walkparams.tgx,
                        walkstate.base);

                // Detect Address Size Fault by table descriptor
                if AArch64.OAOutOfRange(descaddress.address, walkparams,
                    walkparams.tgx) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN);

                if walkparams.haft == '1' then
                    new_descriptor<10> = '1';

                if (walkparams.d128 == '1' && skl != '00' &&
                    AArch64.BlocknTFaults(walkparams.d128, descriptor));
                    fault.statuscode = Fault_Translation;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN);

            when DescriptorType_Leaf
                walkstate = AArch64.S2NextWalkStateLeaf(walkstate,
                    descriptor);

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN);

            otherwise
                Unreachable();

        if new_descriptor != descriptor then
            AccessDescriptor descaccess = CreateAccDescTTEUpdate(access,
                (fault, descriptor) = AArch64.MemSwapTableDesc(fault,
                    walkparams,
                    walkaddr);

            if fault.statuscode != Fault None then
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN);

            until new_descriptor == descriptor;
            until desctype == DescriptorType_Leaf;

            FullAddress oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx,
                walkparams);

            if (walkstate.contiguous == '1' &&

```

```

AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz,
                            walkstate.level)) then
    fault.statuscode = Fault Translation;
elseif (desctype == DescriptorType_Leaf && walkstate.level < FINAL_I
       AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
    fault.statuscode = Fault Translation;
// Detect Address Size Fault by final output
elseif AArch64.OAOutOfRange(oa.address, walkparams.d128, walkparams.
                            fault.statuscode = Fault AddressSize;
// Check descriptor AF bit
elseif (descriptor<10> == '0' && walkparams.ha == '0' &&
       !(accdesc.acctype IN {AccessType_DC, AccessType_IC} &&
       !boolean IMPLEMENTATION_DEFINED "Generate access flag fault")
       fault.statuscode = Fault AccessFlag;

return (fault, walkaddress, walkstate, descriptor);

```

### Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.SS2InitialTTWState

```

// AArch64.SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipa
                                    TTWState walkstate;
                                    FullAddress tablebase;
                                    bits(128) ttbr;

if ipaspace == PAS_Secure then
    ttbr = ZeroExtend(VSTTBR_EL2, 128);
else
    ttbr = ZeroExtend(VTTBRL2, 128);

if ipaspace == PAS_Secure then
    if walkparams.sw == '0' then
        tablebase.paspace = PAS_Secure;
    else
        tablebase.paspace = PAS_NonSecure;
else
    if walkparams.nsw == '0' then
        tablebase.paspace = PAS_Secure;
    else
        tablebase.paspace = PAS_NonSecure;

tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace);

walkstate.baseaddress = tablebase;
walkstate.level      = AArch64.S2StartLevel(walkparams);
walkstate.istable    = TRUE;
walkstate.memattrs   = WalkMemAttrs(walkparams.sh, walkparams.irgrn);

return walkstate;

```

### Library pseudocode for aarch64/translation/vmsa\_walk/ AArch64.SS2OutputPASpace

```

// AArch64.SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.BBMSupportLevel**

```

// AArch64.BBMSupportLevel()
// =====
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !IsFeatureImplemented(FEAT_BBM) then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level"

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.GetS1TTWParams**

```

// AArch64.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, SecurityState ss, bit
S1TTWParams walkparams;

varange = AArch64.GetVARange(va);

case regime of
    when Regime_EL3 walkparams = AArch64.S1TTWParamsEL3();
    when Regime_EL2 walkparams = AArch64.S1TTWParamsEL2(ss);
    when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(ss, varange);
    when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

return walkparams;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.GetS2TTWParams**

```

// AArch64.GetS2TTWParams()
// =====

```

```

// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.Gets2TTWParams(SecurityState ss, PASpace ipaspace,
S2TTWParams walkparams;

if ss == SS_NonSecure then
    walkparams = AArch64.NSS2TTWParams(s1aarch64);
elsif IsFeatureImplemented(FEAT_SEL2) && ss == SS_Secure then
    walkparams = AArch64.SS2TTWParams(ipaspace, s1aarch64);
elsif ss == SS_Realm then
    walkparams = AArch64.RLS2TTWParams(s1aarch64);
else
    Unreachable();

return walkparams;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.GetVRange**

```

// AArch64.GetVRange()
// =====
// Determines if the VA that is to be translated lies in LOWER or UPPER

VARange AArch64.GetVRange(bits(64) va)
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.HaveS1TG**

```

// AArch64.HaveS1TG()
// =====
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx_4KB return boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule";
        when TGx_16KB return boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule";
        when TGx_64KB return boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule";

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.HaveS2TG**

```

// AArch64.HaveS2TG()
// =====
// Determine whether the given translation granule is supported for stage 2

boolean AArch64.HaveS2TG(TGx tgx)
    assert HaveEL(EL2);

    if IsFeatureImplemented(FEAT_GTG) then
        case tgx of
            when TGx_4KB

```

```

        return boolean IMPLEMENTATION_DEFINED "Has Stage 2 4K T
when TGx_16KB
        return boolean IMPLEMENTATION_DEFINED "Has Stage 2 16K
when TGx_64KB
        return boolean IMPLEMENTATION_DEFINED "Has Stage 2 64K
else
    return AArch64.HaveS1TG(tgx);

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.MaxTxSZ**

```

// AArch64.MaxTxSZ()
// =====
// Retrieve the maximum value of TxSZ indicating minimum input address
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if IsFeatureImplemented(FEAT_TTST) then
        case tgx of
            when TGx_4KB      return 48;
            when TGx_16KB     return 48;
            when TGx_64KB     return 47;

    return 39;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.NSS2TTWParams**

```

// AArch64.NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean s1aarch64)
    S2TTWParams walkparams;

    walkparams.vm      = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx     = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz   = VTCR_EL2.T0SZ;
    walkparams.ps      = VTCR_EL2.PS;
    walkparams.irgn   = VTCR_EL2.IRGN0;
    walkparams.orgn   = VTCR_EL2.ORGNO;
    walkparams.sh      = VTCR_EL2.SH0;
    walkparams.ee      = SCTLR_EL2.EE;
    walkparams.d128   = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2.D128;
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;

    walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.PTW;
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB) then HCR_EL2.FWB;
    walkparams.ha  = if IsFeatureImplemented(FEAT_HAFDBS) then VTCR_EL2.HA;
    walkparams.hd  = if IsFeatureImplemented(FEAT_HAFDBS) then VTCR_EL2.HD;
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_HAFDBS)
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';

```

```

        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
        walkparams.cmow = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2E
                            else '0');
        if walkparams.d128 == '1' then
            walkparams.s2pie = '1';
        else
            walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2.S2PIE;
        walkparams.s2pir = if IsFeatureImplemented(FEAT_S2PIE) then S2PIR_E;
        if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
            walkparams.assuredonly = VTCR_EL2.AssuredOnly;
        else
            walkparams.assuredonly = '0';
        walkparams.tl0 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.TL0;
        walkparams.tl1 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.TL1;
        if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
            walkparams.haft = VTCR_EL2.HAFT;
        else
            walkparams.haft = '0';

    return walkparams;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.PAMax**

```

// AArch64.PAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED maximum number of bits capable of
// physical address for this processor

integer AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size"

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.RLS2TTWParams**

```

// AArch64.RLS2TTWParams()
// =====
// Gather walk parameters specific for Realm stage 2 translation

S2TTWParams AArch64.RLS2TTWParams(boolean s1aarch64)
    // Realm stage 2 walk parameters are similar to Non-secure
    S2TTWParams walkparams = AArch64.NSS2TTWParams(s1aarch64);
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                        IsSCTRL2EL2Enabled() then SCTRL2_EL2.EMEC else '0');

    return walkparams;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1DCacheEnabled**

```

// AArch64.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

```

```

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime EL3 return SCLTR_EL3.C == '1';
        when Regime EL2 return SCLTR_EL2.C == '1';
        when Regime EL20 return SCLTR_EL2.C == '1';
        when Regime EL10 return SCLTR_EL1.C == '1';

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1DecodeTG0**

```

// AArch64.S1DecodeTG0()
// =====
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size"

    case tg0 of
        when '00'    tgx = TGx 4KB;
        when '01'    tgx = TGx 64KB;
        when '10'    tgx = TGx 16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size"
            when '00'    tgx = TGx 4KB;
            when '01'    tgx = TGx 64KB;
            when '10'    tgx = TGx 16KB;

    return tgx;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1DecodeTG1**

```

// AArch64.S1DecodeTG1()
// =====
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tg1_in)
    bits(2) tg1 = tg1_in;
    TGx tgx;

    if tg1 == '00' then
        tg1 = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size"

    case tg1 of
        when '10'    tgx = TGx 4KB;
        when '11'    tgx = TGx 64KB;
        when '01'    tgx = TGx 16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size"
            when '10'    tgx = TGx 4KB;

```

```

        when '11'    tgx = TGx_64KB;
        when '01'    tgx = TGx_16KB;

    return tgx;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1E0POEnabled**

```

// AArch64.S1E0POEnabled()
// =====
// Determine whether stage 1 unprivileged permission overlay is enabled

boolean AArch64.S1E0POEnabled(Regime regime, bit nv1)
    assert HasUnprivileged(regime);

    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;

    case regime of
        when Regime_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.E0POE == 1;
        when Regime_EL10 return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1.E0POE == 1;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1EPD**

```

// AArch64.S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the access type

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.M == 1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.M == 1;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1Enabled**

```

// AArch64.S1Enabled()
// =====
// Determine if stage 1 is enabled for the access type for this translation

boolean AArch64.S1Enabled(Regime regime, AccessType acctype)
    case regime of
        when Regime_EL3  return SCTRLR_EL3.M == '1';
        when Regime_EL2  return SCTRLR_EL2.M == '1';
        when Regime_EL20 return SCTRLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled()) || HCR_EL2.<DC,TGE> == 1;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1ICacheEnabled**

```

// AArch64.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime\_EL3 return SCTRLR_EL3.I == '1';
        when Regime\_EL2 return SCTRLR_EL2.I == '1';
        when Regime\_EL20 return SCTRLR_EL2.I == '1';
        when Regime\_EL10 return SCTRLR_EL1.I == '1';

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1MinTxSZ**

```

// AArch64.S1MinTxSZ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address

integer AArch64.S1MinTxSZ(Regime regime, bit d128, bit ds, TGx tgx)
    if IsFeatureImplemented(FEAT_LVA3) && d128 == '1' then
        if HasUnprivileged(regime) then
            return 9;
        else
            return 8;
    if (IsFeatureImplemented(FEAT_LVA) && tgx == TGx\_64KB) || ds == '1'
        return 12;

    return 16;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1POEnabled**

```

// AArch64.S1POEnabled()
// =====
// Determine whether stage 1 privileged permission overlay is enabled

boolean AArch64.S1POEnabled(Regime regime)
    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;

    case regime of
        when Regime\_EL3 return TCR_EL3.POE == '1';
        when Regime\_EL2 return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime\_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime\_EL10 return IsTCR2EL1Enabled() && TCR2_EL1.POE == '1';

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1POR**

```

// AArch64.S1POR()
// =====
// Identify stage 1 permissions overlay register for the acting translation

S1PORType AArch64.S1POR(Regime regime)
    case regime of

```

```

when Regime_EL3    return POR_EL3;
when Regime_EL2    return POR_EL2;
when Regime_EL20   return POR_EL2;
when Regime_EL10   return POR_EL1;

```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1TTBR

```

// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regi

bits(128) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3    return ZeroExtend(TTBR0_EL3, 128);
        when Regime_EL2    return ZeroExtend(TTBR0_EL2, 128);
        when Regime_EL20
            if varange == VARange_LOWER then
                return ZeroExtend(TTBR0_EL2, 128);
            else
                return ZeroExtend(TTBR1_EL2, 128);
        when Regime_EL10
            if varange == VARange_LOWER then
                return ZeroExtend(TTBR0_EL1, 128);
            else
                return ZeroExtend(TTBR1_EL1, 128);

```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1TTWParamsEL10

```

// AArch64.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)
    S1TTWParams walkparams;

    if IsFeatureImplemented(FEAT_D128) && IstCR2EL1Enabled() then
        walkparams.d128 = TCR2_EL1.D128;
    else
        walkparams.d128 = '0';
    if varange == VARange_LOWER then
        walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz = TCR_EL1.T0SZ;
        walkparams.irgn = TCR_EL1.IRGN0;
        walkparams.orgn = TCR_EL1.ORGN0;
        walkparams.sh   = TCR_EL1.SH0;
        walkparams.tbi  = TCR_EL1.TBI0;

        walkparams.nfd  = (if IsFeatureImplemented(FEAT_SVE) || IsFeat
                           then TCR_EL1.NFD0 else '0');
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAUTH) then TCR
                           walkparams.e0pd = if IsFeatureImplemented(FEAT_EOPD) then TCR
                           walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR

```

```

        walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL1.MTX;
        walkparams.skl = if walkparams.d128 == '1' then TTBR0_EL1.SKL;
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DISCH;
    else
        walkparams.tgx = AArch64.S1DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz = TCR_EL1.T1SZ;
        walkparams.irgn = TCR_EL1.IRGN1;
        walkparams.orgn = TCR_EL1.ORGN1;
        walkparams.sh = TCR_EL1.SH1;
        walkparams.tbi = TCR_EL1.TBI1;

        walkparams.nfd = (if IsFeatureImplemented(FEAT_SVE) || IsFeatureImplemented(FEAT_MTE4) then TCR_EL1.NFD1 else '0');
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID;
        walkparams.e0pd = if IsFeatureImplemented(FEAT_EOPD) then TCR_EL1.E0PD;
        walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL1.HPDS;
        walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL1.MTX;
        walkparams.skl = if walkparams.d128 == '1' then TTBR1_EL1.SKL;
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DISCH;

    walkparams.mair = MAIR_EL1;
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL1;
    walkparams.aie = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL1Enabled() then TCR2_EL1.AIE else '0');
    walkparams.wxn = SCTLR_EL1.WXN;
    walkparams.ps = TCR_EL1.IPS;
    walkparams.ee = SCTLR_EL1.EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_CMOW))) || IsFeatureImplemented(FEAT_MTE2) then HCR_EL2.SIF;
    else
        walkparams.sif = '0';

    if EL2Enabled() then
        walkparams.dc = HCR_EL2.DC;
        walkparams.dct = if IsFeatureImplemented(FEAT_MTE2) then HCR_EL2.DCT else '0';
    end;

    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = SCTLR_EL1.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case ConstrainUnpredictable(Unpredictable NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';
            else
                walkparams.nv1 = HCR_EL2.NV1;
        else
            walkparams.nv1 = '0';
    end;

    walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW) then SCTLR_EL1.CMOW;
    walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL1.HA;
    walkparams.hd = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL1.HD;
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_CMOW) then
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '0';

```

```

if walkparams.d128 == '1' then
    walkparams.pie = '1';
else
    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
                      IsTCR2EL1Enabled() then TCR2_EL1.PIE else '0')
if IsFeatureImplemented(FEAT_S1PIE) then
    walkparams.pir = PIR_EL1;
    if walkparams.nv1 != '1' then
        walkparams.pire0 = PIRE0_EL1;
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCLTR_EL1.EPAN
else
    walkparams.epan = '0';
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL1Enabled()
    walkparams.pnch = TCR2_EL1.PnCH;
else
    walkparams.pnch = '0';
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL1Enabled()
    walkparams.haft = TCR2_EL1.HAFT;
else
    walkparams.haft = '0';
walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                     IsSCLTR2EL2Enabled() then SCLTR2_EL2.EMEC else '0')

return walkparams;

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1TTWParamsEL2

```

// AArch64.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2(SecurityState ss)
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGN0;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2;
    walkparams.aie = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled()
                      else '0');
    walkparams.wxn = SCLTR_EL2.WXN;
    walkparams.ee = SCLTR_EL2.EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_HAFDBS))
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2.TBID
    walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL2.HPD
    walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HA

```

```

walkparams.hd = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.DS;
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_HAFDBS)
    walkparams.ds = TCR_EL2.DS;
else
    walkparams.ds = '0';
walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) && IsTCR2EL2Enabled()
                  else '0');
if IsFeatureImplemented(FEAT_S1PIE) then
    walkparams.pir = PIR_EL2;
walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL2.MTX;
walkparams.bnch = (if IsFeatureImplemented(FEAT_THE) && IsTCR2EL2Enabled()
                   else '0');
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled()
    walkparams.haft = TCR2_EL2.HAFT;
else
    walkparams.haft = '0';
walkparams.emec = (if IsFeatureImplemented(FEAT_MECC) && IsSCLR2EL2Enabled() then SCLR2_EL2.EMEC else '0');
if IsFeatureImplemented(FEAT_MECC) && ss == SS_Realm && IsTCR2EL2Enabled()
    walkparams.amec = TCR2_EL2.AMEC0;
else
    walkparams.amec = '0';

return walkparams;

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1TTWParamsEL20

```

// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(SecurityState ss, VARange varange)
S1TTWParams walkparams;

if IsFeatureImplemented(FEAT_D128) && IsTCR2EL2Enabled() then
    walkparams.d128 = TCR2_EL2.D128;
else
    walkparams.d128 = '0';
if varange == VARange LOWER then
    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGN0;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI0;

    walkparams.nfd = (if IsFeatureImplemented(FEAT_SVE) ||
                      IsFeatureImplemented(FEAT_TME) then TCR_EL2.NFD);
    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2.TBID;
    walkparams.e0pd = if IsFeatureImplemented(FEAT_EOPD) then TCR_EL2.E0PD;
    walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL2.HPDS;
    walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL2.MTX;
    walkparams.skl = if walkparams.d128 == '1' then TTBR0_EL2.SKL;
    walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DISCH;
else
    walkparams.tgx = AArch64.S1DecodeTG1(TCR_EL2.TG1);
    walkparams.txsz = TCR_EL2.T1SZ;

```

```

walkparams.irgn = TCR_EL2.IRGN1;
walkparams.orgn = TCR_EL2.ORGN1;
walkparams.sh = TCR_EL2.SH1;
walkparams.tbi = TCR_EL2.TBI1;

walkparams.nfd = (if IsFeatureImplemented(FEAT_SVE) || IsFeat
                  then TCR_EL2.NFD1 else '0');
walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR
walkparams.e0pd = if IsFeatureImplemented(FEAT_EOPD) then TCR
walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR
walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR
walkparams.skl = if walkparams.d128 == '1' then TTBR1_EL2.SKI
walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisC
walkparams.mair = MAIR_EL2;
if IsFeatureImplemented(FEAT_AIE) then
    walkparams.mair2 = MAIR2_EL2;
walkparams.aie = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled()
                  else '0');
walkparams.wxn = SCTLR_EL2.WXN;
walkparams.ps = TCR_EL2.IPS;
walkparams.ee = SCTLR_EL2.EE;
if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImpe
    walkparams.sif = SCR_EL3.SIF;
else
    walkparams.sif = '0';

if IsFeatureImplemented(FEAT_LSMIAOC) then
    walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
else
    walkparams.ntlsmd = '1';

walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW) then SCTLR_E
walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HA
walkparams.hd = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HD
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_CMOW)
    walkparams.ds = TCR_EL2.DS;
else
    walkparams.ds = '0';
if walkparams.d128 == '1' then
    walkparams.pie = '1';
else
    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) && IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0');
if IsFeatureImplemented(FEAT_S1PIE) then
    walkparams.pir = PIR_EL2;
    walkparams.pire0 = PIRE0_EL2;
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL2.EPAN
else
    walkparams.epan = '0';
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL2Enabled()
    walkparams.pnch = TCR2_EL2.PnCH;
else
    walkparams.pnch = '0';
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled()
    walkparams.haft = TCR2_EL2.HAFT;
else
    walkparams.haft = '0';
walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) && IsSCTLR2EL2Enabled()
                  then SCTLR2_EL2.EMEC else '0');

```

```

    if IsFeatureImplemented(FEAT_MEC) && ss == SS_Realm && IsTCR2EL2Enabled()
        walkparams.amec = if varange == VARANGE_LOWER then TCR2_EL2.AMEC
    else
        walkparams.amec = '0';

    return walkparams;
}

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S1TTWParamsEL3

```

// AArch64.S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGNO;
    walkparams.sh = TCR_EL3.SH0;
    walkparams.tbi = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL3;
    walkparams.aie = if IsFeatureImplemented(FEAT_AIE) then TCR_EL3.AIE else '0';
    walkparams.wxn = SCTLR_EL3.WXN;
    walkparams.ee = SCTLR_EL3.EE;
    walkparams.sif = (if !IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SIF)
                      then SCR_EL3.SIF else '0');

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL3.TBID;
    walkparams.hpd = if IsFeatureImplemented(FEAT_HPDS) then TCR_EL3.HPD;
    walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL3.HA;
    walkparams.hd = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL3.HD;
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_TGx)
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';
    walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then TCR_EL3.D128;
    walkparams.skl = if walkparams.d128 == '1' then TTBR0_EL3.SKEL else '0';
    walkparams.disch = if walkparams.d128 == '1' then TCR_EL3.DisCH0 else '0';
    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if IsFeatureImplemented(FEAT_S1PIE) then TCR_EL3.PIE else '0';
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL3;
    walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL3.MTX;
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' then
        walkparams.pnch = TCR_EL3.PnCH;
    else
        walkparams.pnch = '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3.HAFT;
    else
        walkparams.haft = '0';
}

```

```

        walkparams.haft = '0';
walkparams.emec = if IsFeatureImplemented(FEAT_MEC) then SCLTR2_EL3
return walkparams;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S2DecodeTG0**

```

// AArch64.S2DecodeTG0 ()
// =====
// Decode stage 2 granule size configuration bits TG0

TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
bits(2) tg0 = tg0_in;
TGx tgx;

if tg0 == '11' then
    tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size"

case tg0 of
    when '00'    tgx = TGx 4KB;
    when '01'    tgx = TGx 64KB;
    when '10'    tgx = TGx 16KB;

if !AArch64.HaveS2TG(tgx) then
    case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size"
        when '00'    tgx = TGx 4KB;
        when '01'    tgx = TGx 64KB;
        when '10'    tgx = TGx 16KB;

return tgx;

```

### **Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.S2MinTxSZ**

```

// AArch64.S2MinTxSZ ()
// =====
// Retrieve the minimum value of TxSZ indicating maximum input address

integer AArch64.S2MinTxSZ(bit d128, bit ds, TGx tgx, boolean s1aarch64)
ips = AArch64.PAMax();

if d128 == '0' then
    if IsFeatureImplemented(FEAT_LPA) && tgx != TGx 64KB && ds == '0'
        ips = Min(48, AArch64.PAMax());
    else
        ips = Min(52, AArch64.PAMax());
min_txsz = 64 - ips;
if !s1aarch64 then
    // EL1 is AArch32
    min_txsz = Min(min_txsz, 24);

return min_txsz;

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/ AArch64.SS2TTWParams

```
// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean s1aarch64)
S2TTWParams walkparams;

walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2.D128;
if ipaspace == PAS\_Secure then
    walkparams.tgx = AArch64.S2DecodeTG0(VSTCR_EL2.TG0);
    walkparams.txsz = VSTCR_EL2.T0SZ;
    if walkparams.d128 == '1' then
        walkparams.skl = VSTTBR_EL2.SK1;
    else
        walkparams.s10 = VSTCR_EL2.SL0;
    if walkparams.tgx == TGx\_4KB && IsFeatureImplemented(FEAT_LPA2)
        walkparams.s12 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.s12 = '0';
elsif ipaspace == PAS\_NonSecure then
    walkparams.tgx = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2.SK1;
    else
        walkparams.s10 = VTCR_EL2.SL0;
    if walkparams.tgx == TGx\_4KB && IsFeatureImplemented(FEAT_LPA2)
        walkparams.s12 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.s12 = '0';
else
    Unreachable();

walkparams.sw      = VSTCR_EL2.SW;
walkparams.nsw     = VTCR_EL2.NSW;
walkparams.sa      = VSTCR_EL2.SA;
walkparams.nsa     = VTCR_EL2.NSA;
walkparams.vm      = HCR_EL2.VM OR HCR_EL2.DC;
walkparams.ps      = VTCR_EL2.PS;
walkparams.irgn    = VTCR_EL2.IRGN0;
walkparams.orgn    = VTCR_EL2.ORGNO;
walkparams.sh      = VTCR_EL2.SH0;
walkparams.ee      = SCTLR_EL2.EE;

walkparams.ptw = if HCR_EL2.TGE == '0' then HCR_EL2.TGE;
walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB) then HCR_EL2.FWB;
walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then VTCR_EL2.HAFDBS;
walkparams.hd   = if IsFeatureImplemented(FEAT_HAFDBS) then VTCR_EL2.HAFDBS;
if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_CMOW)
    walkparams.ds = VTCR_EL2.DS;
else
    walkparams.ds = '0';
walkparams.cmow = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2
                    else '0');
if walkparams.d128 == '1' then
    walkparams.s2pie = '1';
```

```

    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2.S2PIE;
        walkparams.s2pir = if IsFeatureImplemented(FEAT_S2PIE) then S2PIR_E;
        if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
            walkparams.assuredonly = VTCR_EL2.AssuredOnly;
        else
            walkparams.assuredonly = '0';
        walkparams.t10 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.T10;
        walkparams.t11 = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.T11;
        if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
            walkparams.haft = VTCR_EL2.HAFT;
        else
            walkparams.haft = '0';
        walkparams.emec = '0';

    return walkparams;

```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/ S2DCacheEnabled

```

// S2DCacheEnabled()
// =====
// Returns TRUE if Stage 2 Data access cacheability is enabled

boolean S2DCacheEnabled()
    return HCR_EL2.CD == '0';

```

### Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```

// ClearStickyErrors()
// =====

ClearStickyErrors()
    EDSCR.TXU = '0';                                // Clear TX underrun flag
    EDSCR.RXO = '0';                                // Clear RX overrun flag

    if Halted() then                            // in Debug state
        EDSCR.ITO = '0';                            // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether
    // The UNPREDICTABLE behavior also affects the instructions in flight
    // in the pseudocode.
    if (Halted() && EDSCR.ITE == '0' &&
        ConstrainUnpredictableBool(Unpredictable CLEARERRITEZERO)) then
        return;
    EDSCR.ERR = '0';                                // Clear cumulative error flag

return;

```

### Library pseudocode for shared/debug/DebugTarget/DebugTarget

```

// DebugTarget()
// =====
// Returns the debug exception target Exception level

```

```

bits(2) DebugTarget()
    ss = CurrentSecurityState\(\);
    return DebugTargetFrom\(ss\);

```

### Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```

// DebugTargetFrom()
// =====

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    if HaveEL\(EL2\) && (from_state != SS\_Secure ||
        (IsFeatureImplemented(FEAT_SEL2) && (!HaveEL\(EL3\) || SCR_EL3.EE))
        if ELUsingAArch32\(EL2\) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elseif HaveEL\(EL3\) && !HaveAArch64\(\) && from_state == SS\_Secure then
        target = EL3;
    else
        target = EL1;

    return target;

```

### Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```

// DoubleLockStatus()
// =====
// Returns the state of the OS Double Lock.
//     FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the
//     TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the

boolean DoubleLockStatus()
    if !IsFeatureImplemented(FEAT_DoubleLock) then
        return FALSE;
    elseif ELUsingAArch32\(EL1\) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halt;
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' &&

```

### Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```

// OSLockStatus()
// =====
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32\(EL1\) then DBGOSLSR.OSLK else OSLSR_EL1.OSL

```

## **Library pseudocode for shared/debug/SoftwareLockStatus/Component**

```
// Component
// =====
// Component Types.

enumeration Component {
    Component_PMU,
    Component_Debug,
    Component_CTI
};
```

## **Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent**

```
// GetAccessComponent()
// =====
// Returns the accessed component.

Component GetAccessComponent();
```

## **Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus**

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
    otherwise
        Unreachable();
```

## **Library pseudocode for shared/debug/authentication/AccessState**

```
// AccessState()
// =====
// Returns the Security state of the access.

SecurityState AccessState();
```

## **Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess**

```

// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalDebugAccess\(AccessState\(\)\);

// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<EDADE, EDAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if access_state == SS\_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled\(\) then return FALSE;
        if ExternalSecureInvasiveDebugEnabled\(\) then return TRUE;

    if HaveEL\(EL3\) then
        EDAD_bit = if ELUsingAArch32\(EL3\) then SDCR.EDAD else MDCR_EL3.
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation\(\);

```

## Library pseudocode for shared/debug/authentication/ AllowExternalPMSSAccess

```

// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed, FALSE otherwise.

boolean AllowExternalPMSSAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMSSAccess\(AccessState\(\)\);

// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMSSAccess(SecurityState access_state)
    assert IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64\(\);
    assert IsFeatureImplemented(FEAT_Debugv8p4); // Required when PMU

    // The access may also be subject to the OS Double Lock, power-down
    bits(2) epmssad = if HaveEL\(EL3\) then MDCR_EL3.EPMSSAD else '11';

```

```

// Check for reserved values
if !IsFeatureImplemented(FEAT_RME) && epmssad IN {'01','10'} then
    Constraint c;
    (c, epmssad) = ConstrainUnpredictableBits(Unpredictable_RESEPMSSAD);
    // The value returned by ConstrainUnpredictableBits() must be a
    // non-reserved value

SecurityState allowed_state = (if IsFeatureImplemented(FEAT_RME) th

```

case epmssad of

- when '00' return access\_state == allowed\_state;
- when '01' return IsFeatureImplemented(FEAT\_RME) && access\_state == allowed\_state;
- when '10' return IsFeatureImplemented(FEAT\_RME) && access\_state == allowed\_state;
- when '11' return TRUE;

### **Library pseudocode for shared/debug/authentication/ AllowExternalPMUAccess**

```

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU register
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess\(AccessState\(\)\);

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU register
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<EPMADE,EPMAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

        if IsFeatureImplemented(FEAT_Debugv8p4) then
            if access_state == SS\_Secure then return TRUE;
        else
            if !ExternalInvasiveDebugEnabled\(\) then return FALSE;
            if ExternalSecureInvasiveDebugEnabled\(\) then return TRUE;

        if HaveEL\(EL3\) then
            EPMAD_bit = if ELUsingAArch32\(EL3\) then SDCR.EPMAD else MDCR_EI
            return EPMAD_bit == '0';
        else
            return NonSecureOnlyImplementation\(\);
    end if;
end if;

```

### **Library pseudocode for shared/debug/authentication/ AllowExternalTraceAccess**

```

// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is a
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess()
    if !IsFeatureImplemented(FEAT_TRBE) then
        return TRUE;
    else
        return AllowExternalTraceAccess\(AccessState\(\)\);

// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is a
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(SecurityState access_state)
    // The access may also be subject to OS lock, power-down, etc.
    if !IsFeatureImplemented(FEAT_TRBE) then return TRUE;
    assert IsFeatureImplemented(FEAT_Debugv8p4);
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<ETADE, ETAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

        if access_state == SS\_Secure then return TRUE;
        if HaveEL\(EL3\) then
            // External Trace access is not supported for EL3 using AArch32
            assert !ELUsingAArch32\(EL3\);
            return MDCR_EL3.ETAD == '0';
        else
            return NonSecureOnlyImplementation\(\);

```

## Library pseudocode for shared/debug/authentication/Debug\_authentication

```

Signal DBGEN;
Signal NIDEN;
Signal SPIDEN;
Signal SPNIDEN;
Signal RLPIDEN;
Signal RTPIDEN;

```

## Library pseudocode for shared/debug/authentication/ ExternalInvasiveDebugEnabled

```

// ExternalInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// signal DBGEN.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == Signal\_High;

```

## **Library pseudocode for shared/debug/authentication/ ExternalNoninvasiveDebugAllowed**

```
// ExternalNoninvasiveDebugAllowed()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugEnabled(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =====

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;

    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure      return ExternalSecureNoninvasiveDebugEnabled();
        when SS_Realm        return ExternalRealmNoninvasiveDebugEnabled();
        when SS_Root         return ExternalRootNoninvasiveDebugEnabled();
```

## **Library pseudocode for shared/debug/authentication/ ExternalNoninvasiveDebugEnabled**

```
// ExternalNoninvasiveDebugEnabled()
// =====
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return (IsFeatureImplemented(FEAT_Debugv8p4) || ExternalInvasiveDebugEnabled() &&
            NIDEN == Signal\_High);
```

## **Library pseudocode for shared/debug/authentication/ ExternalRealmInvasiveDebugEnabled**

```
// ExternalRealmInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmInvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == Signal\_High;
```

### **Library pseudocode for shared/debug/authentication/ ExternalRealmNoninvasiveDebugEnabled**

```
// ExternalRealmNoninvasiveDebugEnabled()  
// =====  
// The definition of this function is IMPLEMENTATION DEFINED.  
// In the recommended interface, this function returns the state of the  
// (DBGEN AND RLPIDEN) signal.  
  
boolean ExternalRealmNoninvasiveDebugEnabled()  
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;  
    return ExternalRealmInvasiveDebugEnabled\(\);
```

### **Library pseudocode for shared/debug/authentication/ ExternalRootInvasiveDebugEnabled**

```
// ExternalRootInvasiveDebugEnabled()  
// =====  
// The definition of this function is IMPLEMENTATION DEFINED.  
// In the recommended interface, this function returns the state of the  
// (DBGEN AND RLPIDEN AND RTPIDEN AND SPIDEN) signal when FEAT_SEL2 is  
// and the (DBGEN AND RLPIDEN AND RTPIDEN) signal when FEAT_SEL2 is not  
  
boolean ExternalRootInvasiveDebugEnabled()  
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;  
    return (ExternalInvasiveDebugEnabled\(\) &&  
            (!IsFeatureImplemented(FEAT_SEL2) || ExternalSecureInvasiveDebugEnabled\(\) &&  
             RTPIDEN == Signal\_High);
```

### **Library pseudocode for shared/debug/authentication/ ExternalRootNoninvasiveDebugEnabled**

```
// ExternalRootNoninvasiveDebugEnabled()  
// =====  
// The definition of this function is IMPLEMENTATION DEFINED.  
// In the recommended interface, this function returns the state of the  
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.  
  
boolean ExternalRootNoninvasiveDebugEnabled()  
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;  
    return ExternalRootInvasiveDebugEnabled\(\);
```

### **Library pseudocode for shared/debug/authentication/ ExternalSecureInvasiveDebugEnabled**

```
// ExternalSecureInvasiveDebugEnabled()
// =====
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == Signal_High;
```

## **Library pseudocode for shared/debug/authentication/ ExternalSecureNoninvasiveDebugEnabled**

```
// ExternalSecureNoninvasiveDebugEnabled()
// =====
// This function returns the value of ExternalSecureInvasiveDebugEnabled
// is implemented. Otherwise, the definition of this function is IMPLEMENTED.
// In the recommended interface, this function returns the state of the
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if !IsFeatureImplemented(FEAT_Debugv8p4) then
        return (ExternalNoninvasiveDebugEnabled() &&
                (SPIDEN == Signal_High || SPNIDEN == Signal_High));
    else
        return ExternalSecureInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// IsAccessSecure()  
// =====  
// Returns TRUE when an access is Secure  
  
boolean IsAccessSecure();
```

## Library pseudocode for shared/debug/authentication/IsCorePowered

```
// IsCorePowered()
// =====
// Returns TRUE if the Core power domain is powered on, FALSE otherwise

boolean IsCorePowered();
```

## Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```

        boolean i
if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0';
boolean reserved = FALSE;
bits(2) ssc = ssc_in;
bit ssce      = ssce_in;
bit hmc       = hmc_in;
bits(2) pxc = pxc_in;

// Values that are not allocated in any architecture version
case hmc:ssce:ssc:pxc of
    when '0 0 11 10' reserved = TRUE;
    when '0 0 1x xx' reserved = !HaveSecureState();
    when '1 0 00 x0' reserved = TRUE;
    when '1 0 01 10' reserved = TRUE;
    when '1 0 1x 10' reserved = TRUE;
    when 'x 1 xx xx' reserved = ssc != '01' || (hmc:pxc) IN {'000',
otherwise             reserved = FALSE;

// Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
if (!isbreakpt || !HaveAArch32EL(EL1)) && hmc:pxc == '000' && ssc
reserved = TRUE;

// Both EL3 and EL2 are not implemented
if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
reserved = TRUE;

// EL3 is not implemented
if !HaveEL(EL3) && ssc IN {'01','10'} && hmc:ssc:pxc != '10100' then
reserved = TRUE;

// EL3 using AArch64 only
if (!HaveEL(EL3) || !HaveAArch64()) && hmc:ssc:pxc == '11000' then
reserved = TRUE;

// EL2 is not implemented
if !HaveEL(EL2) && hmc:ssc:pxc == '11100' then
reserved = TRUE;

// Secure EL2 is not implemented
if !IsFeatureImplemented(FEAT_SEL2) && (hmc:ssc:pxc) IN {'01100', '11100'}
reserved = TRUE;

if reserved then
    // If parameters are set to a reserved type, behaves as either
    Constraint c;
    (c, <hmc,ssc,ssce,pxc>) = ConstrainUnpredictableBits(Unpredictable);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then
        return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
    // Otherwise the value returned by ConstrainUnpredictableBits must be
    // the same as the input
return (Constraint_NONE, ssc, ssce, hmc, pxc);

```

## Library pseudocode for shared/debug/breakpoint/ ContextMatchingBreakpointRange

```

// ContextMatchingBreakpointRange()
// =====

```

```

// Returns two numbers indicating the index of the first and last context-aware breakpoints implemented.

(integer, integer) ContextMatchingBreakpointRange()
    integer b = NumBreakpointsImplemented\(\);
    integer c = NumContextAwareBreakpointsImplemented\(\);

    if b <= 16 then
        return (b - c, b - 1);
    elseif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);

```

### **Library pseudocode for shared/debug/breakpoint/ IsContextMatchingBreakpoint**

```

// IsContextMatchingBreakpoint()
// =====
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

boolean IsContextMatchingBreakpoint(integer n)
    (lower, upper) = ContextMatchingBreakpointRange\(\);
    return n >= lower && n <= upper;

```

### **Library pseudocode for shared/debug/breakpoint/ NumBreakpointsImplemented**

```

// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";

```

### **Library pseudocode for shared/debug/breakpoint/ NumContextAwareBreakpointsImplemented**

```

// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";

```

### **Library pseudocode for shared/debug/breakpoint/ NumWatchpointsImplemented**

```

// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";

```

## **Library pseudocode for shared/debug/cti/CTI\_ProcessEvent**

```
// CTI_ProcessEvent()  
// =====  
// Process a discrete event on a Cross Trigger output event trigger.  
  
CTI_ProcessEvent(CrossTriggerOut id);
```

## **Library pseudocode for shared/debug/cti/CTI\_SetEventLevel**

```
// CTI_SetEventLevel()  
// =====  
// Set a Cross Trigger multi-cycle input event trigger to the specified  
  
CTI_SetEventLevel(CrossTriggerIn id, Signal level);
```

## **Library pseudocode for shared/debug/cti/CTI\_SignalEvent**

```
// CTI_SignalEvent()  
// =====  
// Signal a discrete event on a Cross Trigger input event trigger.  
  
CTI_SignalEvent(CrossTriggerIn id);
```

## **Library pseudocode for shared/debug/cti/CrossTrigger**

```
// CrossTrigger  
// =====  
  
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest,  
                             CrossTriggerOut IRQ,  
                             CrossTriggerOut_TraceExtIn0,  
                             CrossTriggerOut_TraceExtIn2,  
  
enumeration CrossTriggerIn {CrossTriggerIn_CrossHalt,  
                           CrossTriggerIn_RSVD2,  
                           CrossTriggerIn_TraceExtOut0,  
                           CrossTriggerIn_TraceExtOut2,
```

CrossTrigger  
CrossTrigger  
CrossTrigger  
CrossTrigger

CrossTrigger  
CrossTrigger  
CrossTrigger  
CrossTrigger

## **Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts**

```
// CheckForDCCInterrupts()  
// =====  
  
CheckForDCCInterrupts()  
    commrx = (EDSCR.RXfull == '1');  
    commtx = (EDSCR.TXfull == '0');  
  
    // COMMRX and COMMTRX support is optional and not recommended for ne  
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH  
    // SetInterruptRequestLevel(InterruptID_COMMTRX, if commtx then HIGH
```

```

// The value to be driven onto the common COMMIRQ signal.
boolean commirq;
if ELUsingAArch32(EL1) then
    commirq = ((commrx && DBGDCCINT.RX == '1') ||
                (commtx && DBGDCCINT.TX == '1'));
else
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                (commtx && MDCCINT_EL1.TX == '1'));
SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then Signal
return;

```

### Library pseudocode for shared/debug/dccanditr/DTR

```

bits(32) DTRRX;
bits(32) DTRTX;

```

### Library pseudocode for shared/debug/dccanditr/Read\_DBGDTRRX\_ELO

```

// Read_DBGDTRRX_ELO()
// =====
// Called on reads of debug register 0x080.

bits(32) Read_DBGDTRRX_ELO(boolean memory_mapped)
    return DTRRX;

```

### Library pseudocode for shared/debug/dccanditr/Read\_DBGDTRTX\_ELO

```

// Read_DBGDTRTX_ELO()
// =====
// Called on reads of debug register 0x08C.

bits(32) Read_DBGDTRTX_ELO(boolean memory_mapped)
    underrun = EDSCR.TXfull == '0' || (Halted()) && EDSCR.MA == '1' && EDSCR.IE == '0';
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value; // Error flag seen

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1'; // Underrun condition
        return value; // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0'; // See comments

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>); // A64 "LDR W1,
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);
    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set
    if EDSCR.ERR == '1' then
        EDSCR.TXfull = bit UNKNOWN;
        DBGDTRTX_ELO = bits(64) UNKNOWN;

```

```

        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>); // A64 "MSR DBG
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);
                // "MSR DBGDTRTX_EL0,X1" calls Write_DBGDTR_EL0() which set
                assert EDSCR.TXfull == '1';
            if !UsingAArch32() then
                X[1, 64] = bits(64) UNKNOWN;
            else
                R[1] = bits(32) UNKNOWN;
            EDSCR.ITE = '1'; // See comments

        return value;
    
```

## Library pseudocode for shared/debug/dccanditr/Read\_DBGDTR\_EL0

```

// Read_DBGDTR_EL0()
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGD
bits(N) Read_DBGDTR_EL0(integer N)
    // For MRS <Rt>,DBGDTRTX_EL0 N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0 N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to wr
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;

```

## Library pseudocode for shared/debug/dccanditr/Write\_DBGDTRRX\_EL0

```

// Write_DBGDTRRX_EL0()
// =====
// Called on writes to debug register 0x080.

Write_DBGDTRRX_EL0(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return; // Error flag se

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '1') // Overrun conc
        EDSCR.RXO = '1'; EDSCR.ERR = '1';
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0'; // See comments
        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,
            ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,

```

```

        X[1, 64] = bits(64) UNKNOWN;
    else
        ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // ...
        ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // ...
        R[1] = bits(32) UNKNOWN;
    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to '1'.
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(64) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears EDSCR.RXfull.
        assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1'; // See comments below
    return;

```

### Library pseudocode for shared/debug/dccanditr/**Write\_DBGDTRTX\_EL0**

```

// Write_DBGDTRTX_EL0()
// =====
// Called on writes to debug register 0x08C.

Write_DBGDTRTX_EL0(boolean memory_mapped, bits(32) value)
    DTRTX = value;
    return;

```

### Library pseudocode for shared/debug/dccanditr/**Write\_DBGDTR\_EL0**

```

// Write_DBGDTR_EL0()
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRX_EL0 (AArch32)

Write_DBGDTR_EL0(bits(N) value_in)
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt> N=32, value=X[t]<31:0>, X[t]<63:32> is undefined
    // For MSR DBGDTR_EL0,<Xt> N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>; // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

```

### Library pseudocode for shared/debug/dccanditr/**Write\_EDITR**

```

// Write_EDITR()
// =====
// Called on writes to debug register 0x084.

Write_EDITR(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return; // Error flag set
    if !Halted() then return; // Non-debug state

```

```

if EDSCR.ITE == '0' || EDSCR.MA == '1' then
    EDSCR.ITO = '1';   EDSCR.ERR = '1';           // Overrun condition
    return;

// ITE indicates whether the processor is ready to accept another instruction.
// It may support multiple outstanding instructions. Unlike the "InstrCompl"
// is no indication that the pipeline is empty (all instructions have been
// pseudocode, the assumption is that only one instruction can be executed).
// meaning ITE acts like "InstrCompl".
EDSCR.ITE = '0';

if !_UsingAArch32() then
    ExecuteA64(value);
else
    ExecuteT32(value<15:0>//*hw1*/, value<31:16>//*hw2*/);

EDSCR.ITE = '1';

return;

```

## Library pseudocode for shared/debug/halting/DCPSInstruction

```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !_UsingAArch32())
                handle_el = PSTATE.EL;
            elseif EL2Enabled() && HCR_EL2.TGE == '1' then
                UNDEFINED;
            else
                handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then
                UNDEFINED;
            elseif PSTATE.EL == EL3 && !_UsingAArch32() then
                handle_el = EL3;
            elseif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure
                UNDEFINED;
            else
                handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then
                UNDEFINED;
            else
                handle_el = EL3;
        otherwise
            Unreachable();
    from_secure = CurrentSecurityState() == SS_Secure;

```

```

if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    assert UsingAArch32(); // Cannot move from AArch32 to AArch64
    case handle_el of
        when EL1
            AArch32.WriteMode(M32_Svc);
            if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0'
                PSTATE.PAN = '1';
        when EL2 AArch32.WriteMode(M32_Hyp);
        when EL3
            AArch32.WriteMode(M32_Monitor);
            if IsFeatureImplemented(FEAT_PAN) then
                if !from_secure then
                    PSTATE.PAN = '0';
                elseif SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            if handle_el == EL2 then
                ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
            else
                LR = bits(32) UNKNOWN;
                SPSR_curr[] = bits(32) UNKNOWN;
                PSTATE.E = SCTLR_ELx[].EE;
                DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

        else // Targeting AArch64
            from_32 = UsingAArch32();
            if from_32 then AArch64.MaybeZeroRegisterUppers();
            if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1'
                ResetSVEState();
            else
                MaybeZeroSVEUppers(target_el);
            PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
            if IsFeatureImplemented(FEAT_PAN) && ((handle_el == EL1 && SCTLR.SPAN == '0')
                (handle_el == EL2 && HCR_EL2.TGE == '1' && SCTLR.SPAN == '1')
                PSTATE.PAN = '1';
            ELR_ELx[] = bits(64) UNKNOWN; SPSR_ELx[] = bits(64) UNKNOWN;
            DLR_ELO = bits(64) UNKNOWN; DSPSR_ELO = bits(64) UNKNOWN;
            if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
            if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
            if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = '0';

            UpdateEDSCRFields(); // Update EDSCR PE state
            sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_ELx[].IESB == '1';
            if IsFeatureImplemented(FEAT_DoubleFault) && !UsingAArch32() then
                sync_errors = (sync_errors ||
                    (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL1));
            // SCTLR_ELx[].IESB might be ignored in Debug state.
            if !ConstrainUnpredictableBool(Unpredictable IESBinDebug) then
                sync_errors = FALSE;
            if sync_errors then
                SynchronizeErrors();
            return;

```

## Library pseudocode for shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====

```

```

// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_ELx[].IESB =
    if IsFeatureImplemented(FEAT_DoubleFault) && !UsingAArch32() then
        sync_errors = (sync_errors ||
                        (EffectiveEA()) == '1' && SCR_EL3.NMEA == '1' && P
    // SCTLR_ELx[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SynchronizeContext();

    DebugRestorePSR();

    return;

```

## Library pseudocode for shared/debug/halting/DebugHalt

```

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRO          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive   = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch       = '100111';
constant bits(6) DebugHalt_Watchpoint       = '101011';
constant bits(6) DebugHalt_HaltInstruction  = '101111';
constant bits(6) DebugHalt_SoftwareAccess   = '110011';
constant bits(6) DebugHalt_ExceptionCatch   = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome   = '111011';

```

## Library pseudocode for shared/debug/halting/DebugRestorePSR

```

// DebugRestorePSR()
// =====

DebugRestorePSR()
    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored
    // behave as if UNKNOWN.
    if UsingAArch32() then
        bits(32) spsr = SPSR_curr[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000'; PSTATE.T = '1';           // PSTATE.J is
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        bits(64) spsr = SPSR_ELx[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
        UpdateEDSCRFIELDS();           // Update EDSCRFIELDS

```

## **Library pseudocode for shared/debug/halting/ DisableITRAndResumeInstructionPrefetch**

```
// DisableITRAndResumeInstructionPrefetch()  
// =====  
  
DisableITRAndResumeInstructionPrefetch();
```

## **Library pseudocode for shared/debug/halting/ExecuteA64**

```
// ExecuteA64()  
// =====  
// Execute an A64 instruction in Debug state.  
  
ExecuteA64(bits(32) instr);
```

## **Library pseudocode for shared/debug/halting/ExecuteT32**

```
// ExecuteT32()  
// =====  
// Execute a T32 instruction in Debug state.  
  
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

## **Library pseudocode for shared/debug/halting/ExitDebugState**

```
// ExitDebugState()  
// =====  
  
ExitDebugState()  
    assert Halted();  
    SynchronizeContext();  
  
    // Although EDSCR.STATUS signals that the PE is restarting, debugger  
    // detect that the PE has restarted.  
    EDSCR.STATUS = '000001';  
    // Signal restart  
    // Clear any pending Halting debug events  
    if IsFeatureImplemented(FEAT_Debugv8p8) then  
        EDESR<3:0> = '0000';  
    else  
        EDESR<2:0> = '000';  
  
    bits(64) new_pc;  
    bits(64) spsr;  
  
    if UsingAArch32() then  
        new_pc = ZeroExtend(DLR, 64);  
        if IsFeatureImplemented(FEAT_Debugv8p9) then  
            spsr = DSPSR2 : DSPSR;  
        else  
            spsr = ZeroExtend(DSPSR, 64);  
    else  
        new_pc = DLR_EL0;  
        spsr = DSPSR_EL0;
```

```

boolean illegal_psr_state = IllegalExceptionReturn(spsr);
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE
SetPSTATEFromPSR(spsr); // Can update pr

boolean branch_conditional = FALSE;
if UsingAArch32() then
    if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
else
    // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN
    if illegal_psr_state && spsr<4> == '1' then
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEDebugStateExit(new_pc);
    // A type of branch that is never predicted
    BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1'); // Atomically set
EDPRSR.HALTED = '0';
UpdateEDSCRFields(); // Stop signalling
DisableITRAndResumeInstructionPrefetch();

return;

```

## Library pseudocode for shared/debug/halting/Halt

```

// Halt()
// =====

Halt(bits(6) reason)
    boolean is_async = FALSE;
    FaultRecord fault = NoFault();
    Halt(reason, is_async, fault);

// Halt()
// =====

Halt(bits(6) reason, boolean is_async, FaultRecord fault)
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction(TMFailure_DBG, FALSE);

        CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores

    bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(64) spsr = GetPSRFromPSTATE(DebugState, 64);

    if (IsFeatureImplemented(FEAT_BTI) && !is_async &&
        !(reason IN {DebugHalt Step Normal, DebugHalt Step Exclusive,
                      DebugHalt Step NoSyndrome, DebugHalt Breakpoint, DebugHalt
                      ConstrainUnpredictableBool(Unpredictable_ZEROBTYP)})) then
        spsr<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr<31:0>;

```

```

        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DSPSR2 = spsr<63:32>;
        else
            DLR_ELO = preferred_restart_address;
            DSPSR_ELO = spsr;

            EDSCR.ITE = '1';
            EDSCR.ITO = '0';
            if IsFeatureImplemented(FEAT_RME) then
                if PSTATE.EL == EL3 then
                    EDSCR.SDD = '0';
                else
                    EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0'
            elseif CurrentSecurityState() == SS_Secure then
                EDSCR.SDD = '0'; // If entered in Secure
            elseif HaveEL(EL3) then
                EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else
            else
                EDSCR.SDD = '1'; // Otherwise EDSCR.SDD
            EDSCR.MA = '0';

            // In Debug state:
            // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behavior
            // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable
            //     are not changed on exception entry, this function also leaves them
            // * PSTATE.{IT,T} are ignored.
            // * PSTATE.IL is ignored and behave-as-if 0.
            // * PSTATE.BTYPE is ignored and behave-as-if 0.
            // * PSTATE.TCO is set 1.
            // * PSTATE.{UAO,PAN} are observable and not changed on entry into

            if UsingAArch32() then
                PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
            else
                PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;

                PSTATE.TCO = '1';
                PSTATE.BTYPE = '00';
                PSTATE.IL = '0';
                StopInstructionPrefetchAndEnableITR();
                (EDSCR.STATUS,EDPRSR.HALTED) = (reason, '1');
                UpdateEDSCRFields(); // Update EDSCR PE state
            if IsFeatureImplemented(FEAT_EDHSR) then
                UpdateEDHSR(reason, fault); // Update EDHSR fields.
            if !is_async then EndOfInstruction();
            return;

```

## Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```

// HaltOnBreakpointOrWatchpoint()
// =====
// Returns TRUE if the Breakpoint and Watchpoint debug events should be
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0'

```

## Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// =====

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});
```

## Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is pr

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure return ExternalSecureInvasiveDebugEnabled();
        when SS_Root return ExternalRootInvasiveDebugEnabled();
        when SS_Realm return ExternalRealmInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/halting/Restarting

```
// Restarting()
// =====

boolean Restarting()
    return EDSCR.STATUS == '000001';
```

## Library pseudocode for shared/debug/halting/ StopInstructionPrefetchAndEnableITR

```
// StopInstructionPrefetchAndEnableITR()
// =====

StopInstructionPrefetchAndEnableITR();
```

## Library pseudocode for shared/debug/halting/UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====
// Provides information about the state of the
// implementation defined authentication interface for debug.

UpdateDbgAuthStatus()
    bits(2) nsid, nsnid;
    bits(2) sid, snid;
    bits(32) regVal = Zeros(32);
    if HaveEL(EL3) then
```

```

        if ExternalInvasiveDebugEnabled() then
            nsid = '11';           // Non-Secure Invasive debug implemen
        else
            nsid = '10';           // Non-Secure Invasive debug implemen
        if IsFeatureImplemented(FEAT_Debugv8p4) || ExternalNoninvasiveD
            nsnid = '11';           // Non-Secure Non-Invasive debug impl
        else
            nsnid = '10';           // Non-Secure Non-Invasive debug impl
        if ExternalSecureInvasiveDebugEnabled() then
            sid = '11';           // Secure Invasive debug implemented
        else
            sid = '10';           // Secure Invasive debug implemented
        if IsFeatureImplemented(FEAT_Debugv8p4) || ExternalSecureNoninva
            snid = '11';           // Field has the same value as DBGAU
        else
            snid = '10';           // Secure Non-Invasive debug implemen
        else
            sid   = '00';
            snid  = '00';
            nsid  = '00';
            nsnid = '00';

DBGAUTHSTATUS_EL1.NSID    = nsid;
DBGAUTHSTATUS_EL1.NSNID   = nsnid;
DBGAUTHSTATUS_EL1.SID     = sid;
DBGAUTHSTATUS_EL1.SNID    = snid;
return;

```

### Library pseudocode for shared/debug/halting/UpdateEDHSR

```

// UpdateEDHSR()
// =====
// Update EDHSR watchpoint related fields.

UpdateEDHSR(bits(6) reason, FaultRecord fault)
    bits(64) syndrome = Zeros(64);
    if reason == DebugHalt Watchpoint then
        if IsFeatureImplemented(FEAT_GCS) && fault.accessdesc.acctype =
            syndrome<40> = '1'; // GCS
            syndrome<23:0> = WatchpointRelatedSyndrome(fault, EDWAR);
    else
        syndrome = bits(64) UNKNOWN;
    EDHSR = syndrome;

```

### Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```

// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        if IsFeatureImplemented(FEAT_RME) then
            // SDD bit.
            EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0'

```

```

        EDSCR.<NSE,NS> = bits(2) UNKNOWN;
    else
        // SDD bit.
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0'
        EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        // SError Pending.
        if EL2Enabled() && HCR_EL2.<AMO,TGE> == '10' && PSTATE.EL IN {0,1}
            EDSCR.A = if IsVirtualSErrorPending() then '1' else '0';
        else
            EDSCR.A = if IsPhysicalSErrorPending() then '1' else '0';

        ss = CurrentSecurityState();
        if IsFeatureImplemented(FEAT_RME) then
            case ss of
                when SS_Secure      EDSCR.<NSE,NS> = '00';
                when SS_NonSecure   EDSCR.<NSE,NS> = '01';
                when SS_Root        EDSCR.<NSE,NS> = '10';
                when SS_Realm       EDSCR.<NSE,NS> = '11';
            else
                EDSCR.NS = if ss == SS_Secure then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_curr[].NS == '0' && !IsS
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elseif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elseif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;

```

## Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```

// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current FEAT

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the
    // and PSTATE.EL are correct for the exception target. When FEAT_Debug
    // is not implemented, this function might also be called at any time.
    ss = SecurityStateAtEL(PSTATE.EL);

```

```

integer base;

case ss of
    when SS_Secure      base = 0;
    when SS_NonSecure   base = 4;
    when SS_Realm       base = 16;
    when SS_Root        base = 0;
if HaltingAllowed() then
    boolean halt;
    if IsFeatureImplemented(FEAT_Debugv8p2) then
        exception_exit = !exception_entry;
        increment = if ss == SS_Realm then 4 else 8;
        ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UI
    case ctrl of
        when '00'  halt = FALSE;
        when '01'  halt = TRUE;
        when '10'  halt = (exception_exit == TRUE);
        when '11'  halt = (exception_entry == TRUE);
    else
        halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

    if halt then
        if IsFeatureImplemented(FEAT_Debugv8p8) && exception_entry
            EDESR.EC = '1';
    else
        Halt(DebugHalt_ExceptionCatch);

```

### Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```

// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_pending = step_enabled && EDESR.SS == '1';
    if active_pending then
        if HaltingStep_DidNotStep() then
            FaultRecord fault = NoFault();
            Halt(DebugHalt_Step_NoSyndrome, is_async, fault);
        elseif HaltingStep_SteppedEX() then
            FaultRecord fault = NoFault();
            Halt(DebugHalt_Step_Exclusive, is_async, fault);
        else
            FaultRecord fault = NoFault();
            Halt(DebugHalt_Step_Normal, is_async, fault);
    if step_enabled then ShouldAdvanceHS = TRUE;
    return;

```

### Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```

// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug eve

CheckOSUnlockCatch()
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.OSUCE == '1') ||

```

```
( !IsFeatureImplemented(FEAT_DoPD) && EDECR.OSUCE == '1' ) ) then  
if !Halted() then EDESR.OSUC = '1';
```

### Library pseudocode for shared/debug/haltingevents/ CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()  
// =====  
// Check whether EDESR.EC has been set by an Exception Catch debug event  
  
CheckPendingExceptionCatch(boolean is_async)  
    if IsFeatureImplemented(FEAT_Debugv8p8) && HaltingAllowed() && EDES  
        FaultRecord fault = NoFault();  
        Halt(DebugHalt_ExceptionCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/ CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()  
// =====  
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event  
  
CheckPendingOSUnlockCatch()  
    if HaltingAllowed() && EDESR.OSUC == '1' then  
        boolean is_async = TRUE;  
        FaultRecord fault = NoFault();  
        Halt(DebugHalt_OSUnlockCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()  
// =====  
// Check whether EDESR.RC has been set by a Reset Catch debug event  
  
CheckPendingResetCatch()  
    if HaltingAllowed() && EDESR.RC == '1' then  
        boolean is_async = TRUE;  
        FaultRecord fault = NoFault();  
        Halt(DebugHalt_ResetCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()  
// =====  
// Called after reset  
  
CheckResetCatch()  
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.RCE == '1') ||  
        (!IsFeatureImplemented(FEAT_DoPD) && EDECR.RCE == '1')) then  
        EDESR.RC = '1';  
        // If halting is allowed then halt immediately  
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

## **Library pseudocode for shared/debug/haltingevents/ CheckSoftwareAccessToDebugRegisters**

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if EL1UsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1)
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

## **Library pseudocode for shared/debug/haltingevents/CheckTRBEHalt**

```
// CheckTRBEHalt()
// =====

CheckTRBEHalt()
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_EDECR)
        return;

    if (HaltingAllowed() && TraceBufferEnabled() &&
        TRBSR_EL1.IRQ == '1' && EDECR.TRBE == '1') then
        Halt(DebugHalt_EDBGRQ);
```

## **Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest**

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed() then
        boolean is_async = TRUE;
        FaultRecord fault = NoFault();
        Halt(DebugHalt_EDBGRQ, is_async, fault);
    // Otherwise the CTI continues to assert the debug request until it
```

## **Library pseudocode for shared/debug/haltingevents/HAdvance**

```
// HAdvance()
// =====
// Advance the Halting Step State Machine

HAdvance()
    if !ShouldAdvanceHS then return;
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_not_pending = step_enabled && EDESR.SS == '0';
    if active_not_pending then EDESR.SS = '1'; // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
```

## **Library pseudocode for shared/debug/haltingevents/HaltingStep\_DidNotStep**

```

// HaltingStep_DidNotStep()
// =====
// Returns TRUE if the previously executed instruction was executed in
// if it was not itself stepped.

boolean HaltingStep_DidNotStep();

```

### **Library pseudocode for shared/debug/haltingevents/HaltingStep\_SteppedEX**

```

// HaltingStep_SteppedEX()
// =====
// Returns TRUE if the previously executed instruction was a Load-Exclusive
// executed in the active-not-pending state.

boolean HaltingStep_SteppedEX();

```

### **Library pseudocode for shared/debug/interrupts/ ExternalDebugInterruptsDisabled**

```

// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    SecurityState ss = SecurityStateAtEL(target);
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if EDSCR.INTdis[0] == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled
                when SS_Secure int_dis = ExternalSecureInvasiveDebugEnabled
                when SS_Realm int_dis = ExternalRealmInvasiveDebugEnabled
                when SS_Root int_dis = ExternalRootInvasiveDebugEnabled
            else
                int_dis = FALSE;
        else
            case target of
                when EL3
                    int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled)
                when EL2
                    int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled)
                when EL1
                    if ss == SS_Secure then
                        int_dis = (EDSCR.INTdis IN {'1x'}) && ExternalSecureInvasiveDebugEnabled
                    else
                        int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled)
            end
    end
    return int_dis;

```

### **Library pseudocode for shared/debug/pmu**

```

array integer PMUEventAccumulator[0..30]; // Accumulates PMU events for each event type

array boolean PMULastThresholdValue[0..30]; // A record of the threshold value for each event type

```

## Library pseudocode for shared/debug/pmu/CYCLE\_COUNTER\_ID

```
constant integer CYCLE_COUNTER_ID = 31;
```

## Library pseudocode for shared/debug/pmu/CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events.
// Called before each instruction is executed.

CheckForPMUOverflow()
    boolean check_cnten = FALSE;
    boolean check_e = TRUE;
    boolean check_inten = TRUE;
    boolean include_lo = TRUE;
    boolean include_hi = TRUE;
    boolean exclude_cyc = FALSE;
    boolean exclude_sync = FALSE;

    boolean enabled = PMUInterruptEnabled\(\);
    boolean pmuirq = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                         include_hi, include_lo,
                                         exclude_cyc, exclude_sync);

    SetInterruptRequestLevel(InterruptID\_PMUIRQ,
                                if enabled && pmuirq then Signal High else
                                CTI\_SetEventLevel(CrossTriggerIn\_PMUOverflow, if pmuirq then Signal

    // The request remains set until the condition is cleared.
    // For example, an interrupt handler or cross-triggered event handler
    // the overflow status flag by writing to PMOVSCLR_EL0.

    if IsFeatureImplemented(FEAT_PMUV3p9) && IsFeatureImplemented(FEAT...)
        if pmuirq && HaltingAllowed\(\) && EDECR.PME == '1' then
            Halt(DebugHalt\_EDBGRO);

    if ShouldBRBEFreeze\(\) then
        BRBEFreeze();

    return;
```

## Library pseudocode for shared/debug/pmu/CountPMUEvents

```
// CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (32).
// For the instruction counter, idx == INSTRUCTION_COUNTER_ID (33).

boolean CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters\(\);
    assert (idx == CYCLE\_COUNTER\_ID || idx < num_counters ||
           (idx == INSTRUCTION\_COUNTER\_ID && IsFeatureImplemented(FEAT...))

    boolean debug;
```

```

boolean enabled;
boolean prohibited;
boolean filtered;
boolean frozen;
boolean resvd_for_el2;
bit E;

// Event counting is disabled in Debug state
debug = Halted();

// Software can reserve some counters for EL2
resvd_for_el2 = PMUCounterIsHyp(idx);
ss = CurrentSecurityState();

// Main enable controls
case idx of
    when INSTRUCTION COUNTER ID
        assert HaveAArch64();
        enabled = PMCR_ELO.E == '1' && PMCNTENSET_ELO.F0 == '1';
    when CYCLE COUNTER ID
        if HaveAArch64() then
            enabled = PMCR_ELO.E == '1' && PMCNTENSET_ELO.C == '1';
        else
            enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
    otherwise
        if resvd_for_el2 then
            E = if HaveAArch64() then MDCR_EL2.HPME else HDCR.HPME;
        else
            E = if HaveAArch64() then PMCR_ELO.E else PMCR.E;

        if HaveAArch64() then
            enabled = E == '1' && PMCNTENSET_ELO<idx> == '1';
        else
            enabled = E == '1' && PMCNTENSET<idx> == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;

// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * One of the following is true:
//   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
//     - FEAT_PMUv3p7 is not implemented
//     - MDCR_EL3.MPMX == 0
//   - EL3 is using AArch32 and SDCR.SPME == 0
// * Executing at EL0 using AArch32 and one of the following is true:
//   - EL3 is using AArch32 and SDER.SUNIDEN == 0
//   - EL3 is using AArch64, EL1 is using AArch32, and SDER32_EL3
if HaveEL(EL3) && ss == SS Secure then
    if !ELUsingAArch32(EL3) then
        prohibited = (MDCR_EL3.SPME == '0' && IsFeatureImplemented(
                        MDCR_EL3.MPMX == '0'));
    else
        prohibited = SDCR.SPME == '0';

    if prohibited && PSTATE.EL == EL0 then
        if ELUsingAArch32(EL3) then
            prohibited = SDER.SUNIDEN == '0';
        elseif ELUsingAArch32(EL1) then
            prohibited = SDER32_EL3.SUNIDEN == '0';

```

```

// Event counting at EL3 is prohibited if all of:
// * FEAT_PMUv3p7 is implemented
// * EL3 is using AArch64
// * One of the following is true:
//   - MDCR_EL3.SPME == 0
//   - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
if !prohibited && IsFeatureImplemented(FEAT_PMUv3p7) && PSTATE.EL =
    prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !
                                              prohibited);

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * EL2 is using AArch64 and MDCR_EL2.HPMD == 1 or EL2 is using AA
if !prohibited && PSTATE.EL == EL2 && IsFeatureImplemented(FEAT_PMUv3p7)
    hpmd = if HaveAArch64() then MDCR_EL2.HPMD else HDCR.HPMD;
    prohibited = hpmd == '1';

// The IMPLEMENTATION DEFINED authentication interface might overri
if prohibited && !IsFeatureImplemented(FEAT_Debugv8p2) then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if IsFeatureImplemented(FEAT_PMUv3p7) then
    bit FZ;
    if resvd_for_el2 then
        FZ = if HaveAArch64() then MDCR_EL2.HPMFZO else HDCR.HPMFZO;
    else
        FZ = if HaveAArch64() then PMCR_EL0.FZO else PMCR.FZO;

    frozen = (FZ == '1') && HiLoPMUOverflow(resvd_for_el2);

    frozen = frozen || SPEFreezeOnEvent(idx);

// PMCR_EL0.DP or PMCR.DP disables the cycle counter when event cou
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    dp = if HaveAArch64() then PMCR_EL0.DP else PMCR.DP;
    enabled = enabled && dp == '0';
    // Otherwise whether event counting is prohibited does not affect
    prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited
// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p5) && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && ss == SS_Secure then
        sccd = if HaveAArch64() then MDCR_EL3.SCCD else SDCR.SCCD;
        if sccd == '1' then
            prohibited = TRUE;

    if PSTATE.EL == EL2 then
        hccd = if HaveAArch64() then MDCR_EL2.HCCD else HDCR.HCCD;
        if hccd == '1' then
            prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting can be prohibited

```

```

// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p7) && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && HaveAArch64() && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH}
bits(32) filter;
case idx of
    when INSTRUCTION_COUNTER_ID
        filter = PMICFILTR_EL0<31:0>;
    when CYCLE_COUNTER_ID
        filter = if HaveAArch64() then PMCCFILTR_EL0<31:0> else PMCCFILTR_EL0<31:0>;
    otherwise
        filter = if HaveAArch64() then PMEVTYPE_R_EL0[idx]<31:0> else PMEVTYPE_R_EL0<31:0>;

P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
M = if HaveEL(EL3) && HaveAArch64() then filter<26> else '0';
SH = if HaveEL(EL3) && IsFeatureImplemented(FEAT_SEL2) then filter<25> else '0';
RLK = if IsFeatureImplemented(FEAT_RME) then filter<22> else '0';
RLU = if IsFeatureImplemented(FEAT_RME) then filter<21> else '0';
RLH = if IsFeatureImplemented(FEAT_RME) then filter<20> else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0
        case ss of
            when SS_NonSecure filtered = U != NSU;
            when SS_Secure filtered = U == '1';
            when SS_Realm filtered = U != RLU;
    when EL1
        case ss of
            when SS_NonSecure filtered = P != NSK;
            when SS_Secure filtered = P == '1';
            when SS_Realm filtered = P != RLK;
    when EL2
        case ss of
            when SS_NonSecure filtered = NSH == '0';
            when SS_Secure filtered = NSH == SH;
            when SS_Realm filtered = NSH == RLH;
    when EL3
        if HaveAArch64() then
            filtered = M != P;
        else
            filtered = P == '1';

return !debug && enabled && !prohibited && !filtered && !frozen;

```

## Library pseudocode for shared/debug/pmu/GetNumEventCounters

```

// GetNumEventCounters()
// =====
// Returns the number of event counters implemented. This is indicated
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in
// AArch64 state.

```

```
integer GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";
```

### Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// HasElapsed64Cycles()
// =====
// Returns TRUE if 64 cycles have elapsed between the last count, and E
boolean HasElapsed64Cycles();
```

### Library pseudocode for shared/debug/pmu/HiLoPMUOverflow

```
// HiLoPMUOverflow()
// =====

boolean HiLoPMUOverflow(boolean resvd_for_el2)
    boolean check_cnten = FALSE;
    boolean check_e     = FALSE;
    boolean check_inten = FALSE;
    boolean include_lo  = !resvd_for_el2;
    boolean include_hi  = resvd_for_el2;
    boolean exclude_cyc = FALSE;
    boolean exclude_sync = FALSE;

    boolean overflow = PMUOverflowCondition(check_e, check_cnten, check_i
                                         include_hi, include_lo,
                                         exclude_cyc, exclude_sync);
    return overflow;
```

### Library pseudocode for shared/debug/pmu/INSTRUCTION\_COUNTER\_ID

```
constant integer INSTRUCTION_COUNTER_ID = 32;
```

### Library pseudocode for shared/debug/pmu/IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =====
// Increment the instruction counter and possibly set overflow bits.

IncrementInstructionCounter(integer increment)
    if CountPMUEvents(INSTRUCTION\_COUNTER\_ID) then
        integer old_value = UInt(PMICNTR_EL0);
        integer new_value = old_value + increment;
        PMICNTR_EL0      = new_value<63:0>;

        // The effective value of PMCR_EL0.LP is '1' for the instruction
        if old_value<64> != new_value<64> then
            PMOVSET_EL0.F0 = '1';
            PMOVSCLR_EL0.F0 = '1';

    return;
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEvent

```
// PMUCaptureEvent()
// =====
// If permitted and enabled, generate a PMU snapshot Capture event.

PMUCaptureEvent()
    assert HaveEL(EL3) && IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAP();
    boolean debug_state = Halted();

    if !PMUCaptureEventAllowed() then
        // Indicate a Capture event in progress
        PMSSCR_EL1.<NC,SS> = '10';
        return;

    for idx = 0 to GetNumEventCounters() - 1
        PMEVCNTSVR_EL1[idx] = PMEVCNTR_EL0[idx];

    PMCCNTSVR_EL1 = PMCCNTR_EL0;

    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        PMICNTSVR_EL1 = PMICNTR_EL0;

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSCTL.SS == '1' then
        if pc_sample.valid && !debug_state then
            SetPCSample();
        else
            SetPCSRUnknown();

    if (IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed(PSTATE.E)
        BRBCR_EL1.FZPSS == '1' && (!HaveEL(EL2) || BRBCR_EL2.FZPSS == BRBEFreeze());
        // Indicate a successful Capture event
        PMSSCR_EL1.<NC,SS> = '00';
        if !debug_state || ConstrainUnpredictableBool(Unpredictable_PMUSNAF
            PMUEvent(PMU_EVENT_PMU_SNAPSHOT);

    return;
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEventAllowed

```
// PMUCaptureEventAllowed()
// =====
// Returns TRUE if PMU Capture events are allowed, and FALSE otherwise.

boolean PMUCaptureEventAllowed()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || OSLockStatus() || !HaveA
        return FALSE;
    if HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE == '11';
    elseif EL2Enabled() && ELUsingAArch32(EL2) then
        return FALSE;
    elseif EL2Enabled() && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE == '11';
    elseif ELUsingAArch32(EL1) then
        return FALSE;
    else
```

```

bits(2) pmsse_ell = PMECR_EL1.SSE;
if pmsse_ell == '01' then                                // Reserved value
    Constraint c;
    (c, pmsse_ell) = ConstrainUnpredictableBits(Unpredictable R);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then pmsse_ell = '00';
    // Otherwise the value returned by ConstrainUnpredictableBi
    // a non-reserved value
return pmsse_ell == '11';

```

## Library pseudocode for shared/debug/pmu/PMUCaptureEventEnabled

```

// PMUCaptureEventEnabled()
// =====
// Returns TRUE if PMU Capture events are enabled, and FALSE otherwise.

boolean PMUCaptureEventEnabled()
    if !IsFeatureImplemented(FEAT_PMUV3_SS) || !HaveAArch64() then
        return FALSE;
    if HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE IN {'1x'};
    elseif EL2Enabled() && ELUsingAArch32(EL2) then
        return FALSE;
    elseif EL2Enabled() && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE IN {'1x'};
    elseif ELUsingAArch32(EL1) then
        return FALSE;
    else
        bits(2) pmsse_ell = PMECR_EL1.SSE;
        if pmsse_ell == '01' then                                // Reserved value
            Constraint c;
            (c, pmsse_ell) = ConstrainUnpredictableBits(Unpredictable R);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_ell = '00';
            // Otherwise the value returned by ConstrainUnpredictableBi
            // a non-reserved value
        return pmsse_ell IN {'1x'};

```

## Library pseudocode for shared/debug/pmu/PMUCountValue

```

// PMUCountValue()
// =====
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by.
// 'Vb' is the base value of the event that event counter 'n' is configu

integer PMUCountValue(integer n, integer Vb)
    if !IsFeatureImplemented(FEAT_PMUV3_TH) || !HaveAArch64() then
        return Vb;

    integer T = UInt(PMEVTYPER_EL0[n].TH);
    boolean Vc;

    case PMEVYPER_EL0[n].TC<2:1> of
        when '00' Vc = (Vb != T);           // Disabled or not-equal
        when '01' Vc = (Vb == T);          // Equals
        when '10' Vc = (Vb >= T);         // Greater-than-or-equal

```

```

        when '11' Vc = (Vb < T);           // Less-than

        integer Vt;
        if PMEVTYPEPER_EL0[n].TC<0> == '0' then
            Vt = (if Vc then Vb else 0);      // Count values
        else
            Vt = (if Vc then 1 else 0);      // Count matches

        integer v;
        if IsFeatureImplemented(FEAT_PMUV3_EDGE) && PMEVTYPEPER_EL0[n].TE ==
            Vp = PMULastThresholdValue[n];

        tc = PMEVTYPEPER_EL0[n].TC<1:0>;
        // Check for reserved case
        if tc == '00' then
            Constraint c;
            (c, tc) = ConstrainUnpredictableBits(Unpredictable_RESTC, 2);
            if c == Constraint_DISABLED then tc = '00';
            // Otherwise the value returned by ConstrainUnpredictableBi
            // must be a not-reserved value.

        case tc of
            when '00' v = Vt;                  // Reserved - t
            when '10' v = (if Vp != Vc then 1 else 0); // Both edges
            when 'x1' v = (if !Vp && Vc then 1 else 0); // Single edge
        else
            v = Vt;

        PMULastThresholdValue[n] = Vc;

        return v;

```

## Library pseudocode for shared/debug/pmu/PMUCounterIsHyp

```

// PMUCounterIsHyp()
// =====
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise

boolean PMUCounterIsHyp(integer n)
    if n == INSTRUCTION_COUNTER_ID then return FALSE;
    if n == CYCLE_COUNTER_ID then return FALSE;

    boolean resvd_for_el2;
    if HaveEL(EL2) then          // Software can reserve some event counters
        bits(5) hpmn_bits = if HaveAArch64() then MDCR_EL2.HPMN else HD
        resvd_for_el2 = n >= UInt(hpmn_bits);
        if (UInt(hpmn_bits) > GetNumEventCounters() ||
            (!IsFeatureImplemented(FEAT_HPMN0) && IsZero(hpmn_bits)))
            resvd_for_el2 = ConstrainUnpredictableBool(Unpredictable_C
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;

```

## Library pseudocode for shared/debug/pmu/PMUCounterMask

```

// PMUCounterMask()
// =====
// Return bitmask of accessible PMU counters.

bits(64) PMUCounterMask()
    integer n;
    if UsingAArch32() then
        n = AArch32.GetNumEventCountersAccessible();
    else
        n = AArch64.GetNumEventCountersAccessible();

    mask = ZeroExtend(Ones(n), 64);
    mask<CYCLE_COUNTER_ID> = '1';
    if HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        mask<INSTRUCTION_COUNTER_ID> = '1';
    return mask;

```

## Library pseudocode for shared/debug/pmu/PMUEvent

```

constant bits(16) PMU_EVENT_PMU_SNAPSHOT = 0x8127<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_BR = 0x812A<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_LD = 0x812B<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_ST = 0x812C<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_OP = 0x812D<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_EVENT = 0x812E<15:0>;
constant bits(16) PMU_EVENT_SAMPLE_FEED_LAT = 0x812F<15:0>;
constant bits(16) PMU_EVENT_DSNP_HIT_RD = 0x8194<15:0>;
constant bits(16) PMU_EVENT_L1D_CACHE_HITM_RD = 0x8214<15:0>;
constant bits(16) PMU_EVENT_L2D_CACHE_HITM_RD = 0x8215<15:0>;
constant bits(16) PMU_EVENT_L3D_CACHE_HITM_RD = 0x8216<15:0>;
constant bits(16) PMU_EVENT_LL_CACHE_HITM_RD = 0x8217<15:0>;
constant bits(16) PMU_EVENT_L1D_LFB_HIT_RD = 0x8244<15:0>;
constant bits(16) PMU_EVENT_L2D_LFB_HIT_RD = 0x8245<15:0>;
constant bits(16) PMU_EVENT_L3D_LFB_HIT_RD = 0x8246<15:0>;
constant bits(16) PMU_EVENT_LL_LFB_HIT_RD = 0x8247<15:0>;

// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) pmuevent)
    PMUEvent(pmuevent, 1);

// PMUEvent()
// =====
// Accumulate a PMU Event.

PMUEvent(bits(16) pmuevent, integer increment)
    if SPESampleInFlight then
        SPEEvent(pmuevent);
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(pmuevent, increment, idx);

    if (HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
        pmuevent == PMU_EVENT_INST_RETIRED) then
        IncrementInstructionCounter(increment);

```

```

// PMUEvent()
// =====
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) pmuevent, integer increment, integer idx)
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;

    if UsingAArch32\(\) then
        if PMEVTPER[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTPER_EL0[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;

```

## Library pseudocode for shared/debug/pmu/PMUOverflowCondition

```

// PMUOverflowCondition()
// =====
// Checks for PMU overflow under certain parameter conditions
// If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and PMCR_EL1.E
// If 'check_cnten' is TRUE, then check the applicable PMCNTENCLR_EL0 and PMCNTENCLR_EL1
// If 'check_inten' is TRUE, then check the applicable PMINTENCLR_EL1
// If 'include_lo' is TRUE, then check counters in the set [0..(HPMN-1)]
// and ICNTR, unless excluded by other flags.
// If 'include_hi' is TRUE, then check counters in the set [HPMN..(N-1)]
// If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
// If 'exclude_sync' is TRUE, then counters in synchronous mode are NOT checked.

boolean PMUOverflowCondition(boolean check_e, boolean check_cnten,
                               boolean check_inten,
                               boolean include_hi, boolean include_lo,
                               boolean exclude_cyc, boolean exclude_sync)
constant integer counters = GetNumEventCounters\(\);

bits(64) ovsf;

if HaveAArch64\(\) then
    ovsf = PMOVSCLR_EL0;

    // Remove unimplemented counters - these fields are RES0
    ovsf<63:33> = Zeros(31);

    if !IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        ovsf<INSTRUCTION COUNTER ID> = '0';
    else
        ovsf = ZeroExtend(PMOVSR, 64);

    if counters < 31 then
        ovsf<30:counters> = Zeros(31-counters);

    for idx = 0 to counters - 1
        bit E;

        boolean is_hyp = PMUCounterIsHyp(idx);
        if HaveAArch64\(\) then
            E = (if is_hyp then MDCR_EL2.HPME else PMCR_EL0.E);

```

```

        if exclude_sync then
            bit sync = (PMCNTENCLR_EL0<idx> AND PMEVTYPE_R_EL0[idx].
            ovsf<idx> = ovsf<idx> AND NOT sync;
        else
            E = (if is_hyp then HDCR.HPME else PMCR.E);

        if check_e then
            ovsf<idx> = ovsf<idx> AND E;

        if (!is_hyp && !include_lo) || (is_hyp && !include_hi) then
            ovsf<idx> = '0';

        // Cycle counter
        if exclude_cyc || !include_lo then
            ovsf<CYCLE_COUNTER_ID> = '0';

        if check_e then
            ovsf<CYCLE_COUNTER_ID> = ovsf<CYCLE_COUNTER_ID> AND PMCR_EL0.E;

        // Instruction counter
        if HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
            if !include_lo then
                ovsf<INSTRUCTION_COUNTER_ID> = '0';
            if exclude_sync then
                bit sync = (PMCNTENCLR_EL0.F0 AND PMICFILTR_EL0.SYNC);
                ovsf<INSTRUCTION_COUNTER_ID> = ovsf<INSTRUCTION_COUNTER_ID>

            if check_e then
                ovsf<INSTRUCTION_COUNTER_ID> = ovsf<INSTRUCTION_COUNTER_ID>

        if check_cnten then
            bits(64) cnten = if HaveAArch64() then PMCNTENCLR_EL0 else Zero;
            ovsf = ovsf AND cnten;

        if check_inten then
            bits(64) inten = if HaveAArch64() then PMINTENCLR_EL1 else Zero;
            ovsf = ovsf AND inten;

        return !IsZero(ovsf);
    
```

## Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```

// CreatePCSample()
// =====

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample
    // executes an instruction that can be sampled. An implementation
    // reads of EDPCSR1o return the current values of PC, etc.

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PCSRSuspended() then retu
    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr(64);
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else

```

```

pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

if pc_sample.has_el2 then
    if ELUsingAArch32(EL2) then
        pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
    elseif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0'
        pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        pc_sample.vmid = VTTBR_EL2.VMID;
    if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_VA))
        !ELUsingAArch32(EL2)) then
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
    else
        pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
return;

```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSRSuspended

```

// PCSRSuspended()
// =====
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

boolean PCSRSuspended()
    if PMPCSCTL.IMP == '1' then
        return PMPCSCTL.EN == '0';
    else
        return boolean IMPLEMENTATION_DEFINED "PCSR is suspended";

```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```

PCSample pc_sample;

// PCSample
// =====

type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)

```

## Library pseudocode for shared/debug/samplebasedprofiling/Read\_EDPCSRlo

```

// Read_EDPCSRlo()
// =====

bits(32) Read_EDPCSRlo(boolean memory_mapped)

```

```

if EDPRSR<6:5,0> != '001' then                                // Check DLK, C
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0';                  // Software loc
bits(32) sample;
if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        if IsFeatureImplemented(FEAT_VHE) && EDSCR.SC2 == '1' then
            EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) el
            EDPCSRhi.EL = pc_sample.el;
            EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' el
        else
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else
EDCIDSR = pc_sample.contextidr;
        if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(
            EDSCR.SC2 == '1') then
            EDVIDSR = (if pc_sample.has_el2 then pc_sample.contexti
                else bits(32) UNKNOWN);
        else
            EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN
                then pc_sample.vmid else Zeros(16));
            EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' el
            EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
            EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0');
            // The conditions for setting HV are not specified if PC
            // An example implementation may be "pc_sample.rw".
            EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1'
                else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

return sample;

```

## Library pseudocode for shared/debug/samplebasedprofiling/Read\_PMPCSR

```

// Read_PMPCSR()
// =====

bits(64) Read_PMPCSR(boolean memory_mapped)
if EDPRSR<6:5,0> != '001' then
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(64) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0';                  // Software loc
if IsFeatureImplemented(FEAT_PCSRv8p9) && update then
    if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSCTL.SS == '1' th
        update = FALSE;
    elseif PMPCSCTL.<IMP,EN> == '10' || (PMPCSCTL.IMP == '0' && PCSF

```

```

        pc_sample.valid = FALSE;
        SetPCSRActive\(\);

        if pc_sample.valid then
            if update then SetPCSample\(\);
            return PMPCSR;
        else
            if update then SetPCSRUnknown\(\);
            return (bits(32) UNKNOWN : Ones(32));
    
```

### Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRActive

```

// SetPCSRActive()
// =====
// Sets PC Sample-based Profiling to active state.

SetPCSRActive()
    if PMPCSCTL.IMP == '1' then
        PMPCSCTL.EN = '1';
    // If PMPCSCTL.IMP reads as `0b0`, then PMPCSCTL.EN is RES0, and it
    // IMPLEMENTATION DEFINED whether PSCR is suspended or active at re

```

### Library pseudocode for shared/debug/samplebasedprofiling/ SetPCSRUnknown

```

// SetPCSRUnknown()
// =====
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

SetPCSRUnknown()
    PMPCSR<31:0> = Ones(32);
    PMPCSR<55:32> = bits(24) UNKNOWN;
    PMPCSR.EL = bits(2) UNKNOWN;
    PMPCSR.NS = bit UNKNOWN;

    PMCID1SR = bits(32) UNKNOWN;
    PMCID2SR = bits(32) UNKNOWN;

    PMVIDSR.VMID = bits(16) UNKNOWN;

    return;

```

### Library pseudocode for shared/debug/samplebasedprofiling/SetPCSample

```

// SetPCSample()
// =====
// Sets the PC sample registers to the appropriate sample values.

SetPCSample()
    PMPCSR<31:0> = pc_sample.pc<31:0>;
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_samp
    PMPCSR.EL = pc_sample.el;
    if IsFeatureImplemented(FEAT_RME) then
        case pc_sample.ss of

```

```

        when SS_Secure
            PMPCSR.NSE = '0'; PMPCSR.NS = '0';
        when SS_NonSecure
            PMPCSR.NSE = '0'; PMPCSR.NS = '1';
        when SS_Root
            PMPCSR.NSE = '1'; PMPCSR.NS = '0';
        when SS_Realm
            PMPCSR.NSE = '1'; PMPCSR.NS = '1';
    else
        PMPCSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');

    PMCID1SR = pc_sample.contextidr;
    PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else

    PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} &
                     then pc_sample.vmid else bits(16) UNKNOWN);

    return;

```

## Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```

// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

// Other self-hosted debug functions will call AArch32.GenerateDebugExceptionReturnSS()
// AArch32 state. However, because Software Step is only active when the
// level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptionReturnSS()
step_enabled = (!EL1UsingAArch32(DebugTarget())) && AArch64.GenerateDebugExceptionReturnSS()
                MDSCR_EL1.SS == '1');
active_pending = step_enabled && PSTATE.SS == '0'; // active-pending
if active_pending then
    AArch64.SoftwareStepException();
ShouldAdvanceSS = TRUE;
return;

```

## Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```

// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug Exception Return SS

bit DebugExceptionReturnSS(bits(N) spsr)
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elseif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    boolean valid;
    bits(2) dest_el;

```

```

if IllegalExceptionReturn(spsr) then
    dest_el = PSTATE.EL;
else
    (valid, dest_el) = ELFromSPSR(spsr); assert valid;

dest_ss = SecurityStateAtEL(dest_el);
bit mask;
boolean enabled_at_dest;
dest_using_32 = (if dest_el == EL0 then spsr<4> == '1' else ELUsinc);
if dest_using_32 then
    enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el,
else
    mask = spsr<9>;
    enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el,

ELd = DebugTargetFrom(dest_ss);
bit SS_bit;
if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source
    SS_bit = spsr<21>;
else
    SS_bit = '0';

return SS_bit;

```

### Library pseudocode for shared/debug/softwarestep/SSAdvance

```

// SSAdvance()
// =====
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS
    // current Software Step state machine. However, this check is made
    // processor only needs to consider advancing the state machine from
    // state.
    if !ShouldAdvanceSS then return;
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    return;

```

### Library pseudocode for shared/debug/softwarestep/SoftwareStep\_DidNotStep

```

// SoftwareStep_DidNotStep()
// =====
// Returns TRUE if the previously executed instruction was executed in
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception. Returns FALSE otherwise
// indicating that the previously executed instruction was executed in
// active-not-pending state, that is, the instruction was stepped.

boolean SoftwareStep_DidNotStep();

```

## Library pseudocode for shared/debug/softwarestep/SoftwareStep\_SteppedEX

```
// SoftwareStep_SteppedEX()
// =====
// Returns a value that describes the previously executed instruction.
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or
// that failed its condition code test. Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();
```

## Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then                                // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its
            // can be presented either with COND set to 0xE, the value
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpre
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else
            // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION
            // * CV set to 0 and COND is set to an UNKNOWN value
            // * CV set to 1 and COND is set to the condition code for
            //     applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trap"
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
        else
            syndrome<4> = '1';
            syndrome<3:0> = '1110';

    return syndrome;
```

## Library pseudocode for shared/exceptions/exceptions/Exception

```
// Exception
// =====
// Classes of exception.
```

```

enumeration Exception {
    Exception_Uncategorized,                                // Uncategorized or unknown reason
    Exception_WFxTrap,                                    // Trapped WFI or WFE instruction
    Exception_CP15RTTTrap,                               // Trapped AArch32 MCR or MRC access
    Exception_CP15RRTTrap,                               // Trapped AArch32 MCRR or MRRC access
    Exception_CP14RTTTrap,                               // Trapped AArch32 MCR or MRC access
    Exception_CP14DTTTrap,                               // Trapped AArch32 LDC or STC access
    Exception_CP14RRTTrap,                               // Trapped AArch32 MRRC access
    Exception_AdvSIMDFPAccessTrap,                      // HCPTR-trapped access to SIMD or FP registers
    Exception_FPIDTrap,                                 // Trapped access to SIMD or FP registers
    Exception_LDST64BTrap,                             // Trapped access to ST64BV, ST64DV, or ST64BVW
    // Trapped BXJ instruction not supported in Armv8
    Exception_PACTrap,                                // Trapped invalid PAC use
    Exception_IllegalState,                            // Illegal Execution state
    Exception_SupervisorCall,                          // Supervisor Call
    Exception_HypervisorCall,                          // Hypervisor Call
    Exception_MonitorCall,                            // Monitor Call or Trapped SMC
    Exception_SystemRegisterTrap,                     // Trapped MRS or MSR System register
    Exception_ERetTrap,                               // Trapped invalid ERET use
    Exception_InstructionAbort,                      // Instruction Abort or Prefetch Abort
    Exception_PCAAlignment,                           // PC alignment fault
    Exception_DataAbort,                             // Data Abort
    Exception_NV2DataAbort,                           // Data abort at EL1 reported by SMC
    Exception_PACFail,                               // PAC Authentication failure
    Exception_SPAlignment,                           // SP alignment fault
    Exception_FPTrappedException,                   // IEEE trapped FP exception
    Exception_SError,                                // SError interrupt
    Exception_Breakpoint,                            // (Hardware) Breakpoint
    Exception_SoftwareStep,                          // Software Step
    Exception_Watchpoint,                           // Watchpoint
    Exception_NV2Watchpoint,                         // Watchpoint at EL1 reported by SMC
    Exception_SoftwareBreakpoint,                   // Software Breakpoint Instruction
    Exception_VectorCatch,                           // AArch32 Vector Catch
    Exception_IRQ,                                  // IRQ interrupt
    Exception_SVEAccessTrap,                         // HCPTR trapped access to SVE registers
    Exception_SMEAccessTrap,                         // HCPTR trapped access to SME registers
    Exception_TSTARTAccessTrap,                     // Trapped TSTART access
    Exception_GPC,                                  // Granule protection check
    Exception_BranchTarget,                          // Branch Target Identification
    Exception_MemCpyMemSet,                           // Exception from a CPY* or SEV* instruction
    Exception_GCSFail,                             // GCS Exceptions
    Exception_PMU,                                 // PMU exception
    Exception_SystemRegister128Trap,                // Trapped MRRS or MSRR System register
    Exception_FIQ};                                // FIQ interrupt

```

## Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```

// ExceptionRecord
// =====

type ExceptionRecord is (
    Exception      exceptype,           // Exception class
    bits(25)        syndrome,          // Syndrome record
    bits(24)        syndrome2,         // Syndrome record
    FullAddress   paddress,          // Physical fault address
    bits(64)        vaddress,          // Virtual fault address
    boolean         ipavvalid,         // Validity of Intermediate Physical Address

```

```

boolean      pavalid,           // Validity of Physical fault address
bit          NS,                // Intermediate Physical fault address
bits(56)     ipaddress,        // Intermediate Physical fault address
boolean      trappedsyscallinst // Trapped SVC or SMC instruction

```

## Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```

// ExceptionSyndrome()
// =====
// Return a blank exception syndrome record for an exception of the given type

ExceptionRecord ExceptionSyndrome(Exception exceptype)

ExceptionRecord r;

r.exceptype = exceptype;

// Initialize all other fields
r.syndrome = Zeros(25);
r.syndrome2 = Zeros(24);
r.vaddress = Zeros(64);
r.ipavlid = FALSE;
r.NS = '0';
r.ipaddress = Zeros(56);
r.paddress.paspace = PASpace UNKNOWN;
r.paddress.address = bits(56) UNKNOWN;
r.trappedsyscallinst = FALSE;
return r;

```

## Library pseudocode for shared/exceptions/traps/Undefined

```

// Undefined()
// =====

Undefined()
  if UsingAArch32() then
    AArch32.Undefined();
  else
    AArch64.Undefined();

```

## Library pseudocode for shared/functions/aborts/EncodeLDFSC

```

// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
  bits(6) result;

  // 128-bit descriptors will start from level -2 for 4KB to resolve
  if level == -2 then
    assert IsFeatureImplemented(FEAT_D128);
    case statuscode of
      when Fault\_AddressSize                  result = '101100';
      when Fault\_Translation                 result = '101010';

```

```

        when Fault_SyncExternalOnWalk      result = '010010';
        when Fault_SyncParityOnWalk
            result = '011010';
            assert !IsFeatureImplemented(FEAT_RAS);
        when Fault_GPCFOnWalk                  result = '100010';
        otherwise                                Unreachable();
    return result;

if level == -1 then
    assert IsFeatureImplemented(FEAT_LPA2);
    case statuscode of
        when Fault_AddressSize           result = '101001';
        when Fault_Translation          result = '101011';
        when Fault_SyncExternalOnWalk   result = '010011';
        when Fault_SyncParityOnWalk
            result = '011011';
            assert !IsFeatureImplemented(FEAT_RAS);
        when Fault_GPCFOnWalk          result = '100011';
        otherwise                          Unreachable();

    return result;
case statuscode of
    when Fault_AddressSize           result = '0000':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_AccessFlag           result = '0010':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_Permission          result = '0011':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_Translation         result = '0001':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_SyncExternal        result = '010000';
    when Fault_SyncExternalOnWalk   result = '0101':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_SyncParity          result = '011000';
    when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_AsyncParity         result = '011001';
    when Fault_AsyncExternal       result = '010001'; assert UsingA != FEAT_RAS;
    when Fault_TagCheck           result = '010001'; assert IsFeatureImplemented(FEAT_RAS);
    when Fault_Alignment          result = '100001';
    when Fault_Debug              result = '100010';
    when Fault_GPCFOnWalk         result = '1001':level<1:0>; assert !IsFeatureImplemented(FEAT_RAS);
    when Fault_GPCFOnOutput       result = '101000';
    when Fault_TLBConflict        result = '110000';
    when Fault_HWUpdateAccessFlag result = '110001';
    when Fault_Lockdown           result = '110100'; // IMPLEMENTED;
    when Fault_Exclusive          result = '110101'; // IMPLEMENTED;
    otherwise                          Unreachable();

return result;

```

## Library pseudocode for shared/functions/aborts/IPAValid

```

// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.gpcf.gpf != GPCF_None then
        return fault.secondstage;
    elsif fault.s2fs1walk then
        return fault.statuscode IN {

```

```

        Fault AccessFlag,
        Fault Permission,
        Fault Translation,
        Fault AddressSize
    };
    elseif fault.secondstage then
        return fault.statuscode IN {
            Fault AccessFlag,
            Fault Translation,
            Fault AddressSize
        };
    else
        return FALSE;
}

```

### **Library pseudocode for shared/functions/aborts/IsAsyncAbort**

```

// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault None;

    return (statuscode IN {Fault AsyncExternal, Fault AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);

```

### **Library pseudocode for shared/functions/aborts/IsDebugException**

```

// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault None;
    return fault.statuscode == Fault Debug;

```

### **Library pseudocode for shared/functions/aborts/IsExternalAbort**

```

// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault None;

    return (statuscode IN {
        Fault SyncExternal,
        Fault SyncParity,
        Fault SyncExternalOnWalk,
        Fault SyncParityOnWalk,
    });
}

```

```

        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_

```

### Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```

// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPF_

```

### Library pseudocode for shared/functions/aborts/IsFault

```

// IsFault()
// ======
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// ======
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// ======
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;

```

## Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// =====

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

## Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =====
// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is valid only for certain
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.

bits(11) LSInstructionSyndrome();
```

## Library pseudocode for shared/functions/aborts/ReportAsGPCException

```
// ReportAsGPCException()
// =====
// Determine whether the given GPCF is reported as a Granule Protection
// rather than a Data or Instruction Abort

boolean ReportAsGPCException(FaultRecord fault)
    assert IsFeatureImplemented(FEAT_RME);
    assert fault.statuscode IN {Fault_GPCFOnWalk, Fault_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF_None;
```

```

        case fault.gpcf.gpf of
            when GPCF_Walk          return TRUE;
            when GPCF_AddressSize   return TRUE;
            when GPCF_EABT          return TRUE;
            when GPCF_Fail          return SCR_EL3.GPF == '1' && PSTATE.EL != 1;

```

### **Library pseudocode for shared/functions/cache/CACHE\_OP**

```

// CACHE_OP ()
// =====
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP (CacheRecord cache)
    IMPLEMENTATION_DEFINED;

```

### **Library pseudocode for shared/functions/cache/CPASAtPAS**

```

// CPASAtPAS ()
// =====
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS (PASpace pas)
    case pas of
        when PAS_NonSecure
            return CPAS_NonSecure;
        when PAS_Secure
            return CPAS_Secure;
        when PAS_Root
            return CPAS_Root;
        when PAS_Realm
            return CPAS_Realm;

```

### **Library pseudocode for shared/functions/cache/CPASAtSecurityState**

```

// CPASAtSecurityState ()
// =====
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState (SecurityState ss)
    case ss of
        when SS_NonSecure
            return CPAS_NonSecure;
        when SS_Secure
            return CPAS_SecureNonSecure;
        when SS_Root
            return CPAS_Any;
        when SS_Realm
            return CPAS_RealmNonSecure;

```

### **Library pseudocode for shared/functions/cache/CacheRecord**

```

// CacheRecord
// =====

```

```

// Details related to a cache operation.

type CacheRecord is (
    AccessType          acctype,           // Access type
    CacheOp            cacheop,          // Cache operation
    CacheOpScope       opscope,          // Cache operation type
    CacheType          cachetype,        // Cache type
    bits(64)              regval,
    FullAddress        paddress,         // For VA operations
    bits(64)              vaddress,         // For SW operations
    integer               setnum,           // For SW operations
    integer               waynum,           // For SW operations
    integer               level,             // For SW operations
    Shareability       shareability,
    boolean               translated,
    boolean               is_vmid_valid,   // is vmid valid for current co
    bits(16)              vmid,
    boolean               is_asid_valid,   // is asid valid for current co
    bits(16)              asid,
    SecurityState      security,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    CachePASpace       cpas
)

```

### **Library pseudocode for shared/functions/cache/DCInstNeedsTranslation**

```

// DCInstNeedsTranslation()
// =====
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if opscope == CacheOpScope_PoE then
        return FALSE;

    if opscope == CacheOpScope_PoPA then
        return FALSE;

    if CLIDR_EL1.LoC == '000' then
        return !(boolean IMPLEMENTATION_DEFINED
                  "No fault generated for DC operations if PoC is before

```

the instruction")

```

    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
        return !(boolean IMPLEMENTATION_DEFINED
                  "No fault generated for DC operations if PoU is before

```

the instruction")

```

    return TRUE;

```

### **Library pseudocode for shared/functions/cache/DecodeSW**

```

// DecodeSW()
// =====
// Decode input value into setnum, waynum and level for SW instructions

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype
    level = UInt(regval[3:1]);
    (setnum, waynum, linesize) = GetCacheInfo(level, cachetype);
    return (setnum, waynum, level);

```

### **Library pseudocode for shared/functions/cache/GetCacheInfo**

```

// GetCacheInfo()
// =====
// Returns numsets, associativity & linesize.

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype

```

### **Library pseudocode for shared/functions/cache/ICInstNeedsTranslation**

```

// ICInstNeedsTranslation()
// =====
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs transla

```

### **Library pseudocode for shared/functions/common/ASR**

```

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR\_C(x, shift);
    return result;

```

### **Library pseudocode for shared/functions/common/ASR\_C**

```

// ASR_C()
// =====

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

```

## **Library pseudocode for shared/functions/common/Abs**

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

## **Library pseudocode for shared/functions/common/Align**

```
// Align()
// =====

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =====

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

## **Library pseudocode for shared/functions/common/BitCount**

```
// BitCount()
// =====

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

## **Library pseudocode for shared/functions/common/CountLeadingSignBits**

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

## **Library pseudocode for shared/functions/common/CountLeadingZeroBits**

```
// CountLeadingZeroBits()
// =====
```

```

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);

```

### **Library pseudocode for shared/functions/common/Elem**

```

// Elem[] - non-assignment form
// =====

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<(e*size+size)-1 : e*size>;

// Elem[] - assignment form
// =====

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

```

### **Library pseudocode for shared/functions/common/Extend**

```

// Extend()
// =====

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

```

### **Library pseudocode for shared/functions/common/HighestSetBit**

```

// HighestSetBit()
// =====

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;

```

### **Library pseudocode for shared/functions/common/Int**

```

// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UIInt(x) else SInt(x);
    return result;

```

### **Library pseudocode for shared/functions/common/IsAligned**

```
// IsAligned()
// =====

boolean IsAligned(bits(N) x, integer y)
    return x == Align(x, y);
```

### Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

### Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

### Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

### Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/LSL\_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
```

```

extended_x = x : Zeros(shift);
result = extended_x<N-1:0>;
carry_out = extended_x<N>;
return (result, carry_out);

```

### **Library pseudocode for shared/functions/common/LSR**

```

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

```

### **Library pseudocode for shared/functions/common/LSR\_C**

```

// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

```

### **Library pseudocode for shared/functions/common/LowestSetBit**

```

// LowestSetBit()
// =====

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;

```

### **Library pseudocode for shared/functions/common/Max**

```

// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;

```

### Library pseudocode for shared/functions/common/Min

```

// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;

```

### Library pseudocode for shared/functions/common/Ones

```

// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1', N);

```

### Library pseudocode for shared/functions/common/ROR

```

// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

```

### Library pseudocode for shared/functions/common/ROR\_C

```

// ROR_C()
// =====

```

```

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);

```

### **Library pseudocode for shared/functions/common/RShr**

```

// RShr()
// =====
// Shift integer value right with rounding

integer RShr(integer value, integer shift, boolean round)
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;

```

### **Library pseudocode for shared/functions/common/Replicate**

```

// Replicate()
// =====
bits(M*N) Replicate(bits(M) x, integer N);

```

### **Library pseudocode for shared/functions/common/Reverse**

```

// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, M] = Elem[word, s, M];
    return result;

```

### **Library pseudocode for shared/functions/common/RoundDown**

```

// RoundDown()
// =====

integer RoundDown(real x);

```

### **Library pseudocode for shared/functions/common/RoundTowardsZero**

```
// RoundTowardsZero()
// =====
integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

### **Library pseudocode for shared/functions/common/RoundUp**

```
// RoundUp()
// =====
integer RoundUp(real x);
```

### **Library pseudocode for shared/functions/common/SInt**

```
// SInt()
// =====
integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
        if x<N-1> == '1' then result = result - 2^N;
    return result;
```

### **Library pseudocode for shared/functions/common/SignExtend**

```
// SignExtend()
// =====
bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;
```

### **Library pseudocode for shared/functions/common/Signal**

```
// Signal
// =====
// Available signal types

enumeration Signal {Signal_Low, Signal_High};
```

### **Library pseudocode for shared/functions/common/Split**

```
// Split()
// =====
(bits(M-N), bits(N)) Split(bits(M) value, integer N)
    assert M > N;
    return (value<M-1:N>, value<N-1:0>);
```

## Library pseudocode for shared/functions/common/UInt

```
// UInt()
// =====

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

## Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

## Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0', N);
```

## Library pseudocode for shared/functions/counters/ AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// =====
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();

    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S,
                                         CNTP_CTL_S.IMASK, InterruptID
                                         CNTP_CTL_S.ISTATUS = if status then '1' else '0';

        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS,
                                         CNTP_CTL_NS.IMASK, InterruptID
                                         CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL,
```

```

        CNTP_CTL.IMASK, InterruptID_CNTP)
CNTP_CTL.ISTATUS = if status then '1' else '0';

if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
    status = IsTimerConditionMet(offset, CNTHP_CVAL,
                                CNTHP_CTL.IMASK, InterruptID_CNTHP)
    CNTHP_CTL.ISTATUS = if status then '1' else '0';

if CNTV_CTL_EL0.ENABLE == '1' then
    status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                CNTV_CTL_EL0.IMASK, InterruptID_CNTV)
    CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

return;

```

## Library pseudocode for shared/functions/counters/ AArch64.CheckTimerConditions

```

// AArch64.CheckTimerConditions()
// =====
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    SecurityState ss = CurrentSecurityState();
    boolean ecv = FALSE;
    if IsFeatureImplemented(FEAT_ECV) then
        ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEN == '1' && EL2Enabled
    if ecv then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);
    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS_Root, SS_Realm})
            CNTHCTL_EL2.CNTPMASK == '1') then
            imask = '1';
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                      imask, InterruptID_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if ((HaveEL(EL3) || (HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))
        CNTHP_CTL_EL2.ENABLE == '1')) then
        status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                                      CNTHP_CTL_EL2.IMASK, InterruptID_CNTH)
        CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && CNTHPS_CTL_EL2.ENABLE == '1'
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                      CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS)
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTPS_CTL_EL1.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1,
                                      CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS)
        CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then

```

```

imask = CNTV_CTL_EL0.IMASK;
if (IsFeatureImplemented(FEAT_RME) && ss IN {SS_Root, SS_Realm})
    CNTHCTL_EL2.CNTVMASK == '1') then
    imask = '1';
status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                             imask, InterruptID_CNTV);
CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

if ((IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || !IsFeatureIm-
    CNTHV_CTL_EL2.ENABLE == '1') then
status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                           CNTHV_CTL_EL2.IMASK, InterruptID_C-
CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';

if ((IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_V-
    CNTHVS_CTL_EL2.ENABLE == '1') then
status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                           CNTHVS_CTL_EL2.IMASK, InterruptID_C-
CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
return;

```

## Library pseudocode for shared/functions/counters/GenericCounterTick

```

// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
bits(64) prev_physical_count;
if CNTCR.EN == '0' then
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    return;
prev_physical_count = PhysicalCountInt();
if IsFeatureImplemented(FEAT_CNTSC) && CNTCR.SCEN == '1' then
    PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
else
    PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
if !HaveAArch64() then
    AArch32.CheckTimerConditions();
else
    AArch64.CheckTimerConditions();
TestEventCNP(prev_physical_count, PhysicalCountInt());
TestEventCNTV(prev_physical_count, PhysicalCountInt());
return;

```

## Library pseudocode for shared/functions/counters/IsTimerConditionMet

```

// IsTimerConditionMet()
// =====

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                           bits(1) imask, InterruptID intid)
boolean condition_met;
Signal level;

```

```

        condition_met = (UInt(PhysicalCountInt()) - offset) -
                        UInt(compare_value)) >= 0;
        level = if condition_met && imask == '0' then Signal_High else Signal_Low;
        SetInterruptRequestLevel(intid, level);
        return condition_met;
    
```

### Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

### Library pseudocode for shared/functions/counters/SetEventRegister

```

// SetEventRegister()
// =====
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;

```

### Library pseudocode for shared/functions/counters/TestEventCNTP

```

// TestEventCNTP()
// =====
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count,
              bits(64) offset;
              bits(1) samplebit, previousbit;
              if CNTHCTL_EL2.EVNTE == '1' then
                  n = UInt(CNTHCTL_EL2.EVNTI);
                  if IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2.EVNTIS == '1'
                      n = n + 8;
                  boolean ecv = FALSE;
                  if IsFeatureImplemented(FEAT_ECV) then
                      ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
                             SCR_EL3.ECVEN == '1');
                      offset = if ecv then CNTPOFF_EL2 else Zeros(64);
                      samplebit = (current_physical_count - offset)<n>;
                      previousbit = (prev_physical_count - offset)<n>;
                      if CNTHCTL_EL2.EVNTDIR == '0' then
                          if previousbit == '0' && samplebit == '1' then SetEventRegister();
                      else
                          if previousbit == '1' && samplebit == '0' then SetEventRegister();
              return;

```

### Library pseudocode for shared/functions/counters/TestEventCNTV

```

// TestEventCNTV()
// =====
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count,
              bits(64) offset;
              bits(1) samplebit, previousbit;
              if CNTHCTL_EL2.EVNTDIR == '1' then
                  if previousbit == '0' && samplebit == '1' then SetEventRegister();
              else
                  if previousbit == '1' && samplebit == '0' then SetEventRegister();
              return;

```

```

bits(64) offset;
bits(1) samplebit, previousbit;
if (!IsFeatureImplemented(FEAT_VHE) && HCR_EL2.<E2H,TGE> == '11'
    CNTKCTL_EL1.EVNTEN == '1') then
    n = UInt(CNTKCTL_EL1.EVNTI);
    if IsFeatureImplemented(FEAT_ECV) && CNTKCTL_EL1.EVNTIS == '1'
        n = n + 8;
    if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11'
        offset = CNTVOFF_EL2;
    else
        offset = Zeros(64);
samplebit = (current_physical_count - offset)<n>;
previousbit = (prev_physical_count - offset)<n>;
if CNTKCTL_EL1.EVNTDIR == '0' then
    if previousbit == '0' && samplebit == '1' then SetEventRegi
else
    if previousbit == '1' && samplebit == '0' then SetEventRegi
return;

```

### Library pseudocode for shared/functions/crc/BitReverse

```

// BitReverse()
// =====

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;

```

### Library pseudocode for shared/functions/crc/Poly32Mod2

```

// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;

```

### Library pseudocode for shared/functions/crypto/AESInvMixColumns

```

// AESInvMixColumns()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns()

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op< 0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op< 8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;

```

```

bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;
bits(4*8) out0;
bits(4*8) out1;
bits(4*8) out2;
bits(4*8) out3;

for c = 0 to 3
    out0<c*8+:8> = (FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>)
                           FFmul09(in3<c*8+:8>));
    out1<c*8+:8> = (FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>)
                           FFmul0D(in3<c*8+:8>));
    out2<c*8+:8> = (FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>)
                           FFmul0B(in3<c*8+:8>));
    out3<c*8+:8> = (FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>)
                           FFmul0E(in3<c*8+:8>));

return (
    out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
    out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
    out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
    out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
);

```

## Library pseudocode for shared/functions/crypto/AESInvShiftRows

```

// AESInvShiftRows()
// =====
// Transformation in the Inverse Cipher that is inverse of AESShiftRows

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
        op< 95: 88> : op<119:112> : op< 15: 8> : op< 39: 32> :
        op< 63: 56> : op< 87: 80> : op<111:104> : op< 7: 0>
    );

```

## Library pseudocode for shared/functions/crypto/AESInvSubBytes

```

// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv =
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0          */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :

```

```

/*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
/*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
/*4*/ 0x92b6655dcc5ca4d4169868864f6f872<127:0> :
/*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
/*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
/*1*/ 0xcbefdec444438e3487ff2f9b8239e37c<127:0> :
/*0*/ 0xfb7f3819ea340bf38a53630d56a0952<127:0>

};

bits(128) out;
for i = 0 to 15
    out<i*8+:8> = GF2_inv<UInt>(op<i*8+:8>) *8+:8>;
return out;

```

## Library pseudocode for shared/functions/crypto/AESMixColumns

```

// AESMixColumns()
// =====
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
{
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op< 0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op< 8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = (FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>))
                           in2<c*8+:8> EOR in3<c*8+:8>;
        out1<c*8+:8> = (FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>))
                           in3<c*8+:8> EOR in0<c*8+:8>;
        out2<c*8+:8> = (FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>))
                           in0<c*8+:8> EOR in1<c*8+:8>;
        out3<c*8+:8> = (FFmul02(in3<c*8+:8>) EOR FFmul03(in0<c*8+:8>))
                           in1<c*8+:8> EOR in2<c*8+:8>;

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
}

```

## Library pseudocode for shared/functions/crypto/AESShiftRows

```

// AESShiftRows()
// =====
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)

```

```

        return (
            op< 95: 88> : op< 55: 48> : op< 15: 8> : op<103: 96> :
            op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
            op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
            op<127:120> : op< 87: 80> : op< 47: 40> : op< 7: 0>
        );
    }
}

```

## Library pseudocode for shared/functions/crypto/AESSubBytes

```

// AESSubBytes()
// =====
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbbaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafaf2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
}

```

## Library pseudocode for shared/functions/crypto/FFmul02

```

// FFmul02()
// =====

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
    )
}

```

```

        /*7*/  0xEFCAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
/*6*/  0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
/*5*/  0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
/*4*/  0x9E9C9A98969492908E8C8A8886848280<127:0> :
/*3*/  0x7E7C7A78767472706E6C6A6866646260<127:0> :
/*2*/  0x5E5C5A58565452504E4C4A4846444240<127:0> :
/*1*/  0x3E3C3A38363432302E2C2A2826242220<127:0> :
/*0*/  0x1E1C1A18161412100E0C0A0806040200<127:0>
);
return FFmul_02<UInt>(b) *8+:8>;

```

### Library pseudocode for shared/functions/crypto/FFmul03

```

// FFmul03()
// =====

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/  0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/  0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/  0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/  0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/  0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/  0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/  0xBAB9BCFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/  0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/  0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/  0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/  0xE1E2E7E4EDEEE8F9FAFFF5F6F3F0<127:0> :
        /*4*/  0xD1D2D7D4DDDEDDB8C9CACFCC5C6C3C0<127:0> :
        /*3*/  0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/  0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/  0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/  0x111217141D1E1B18090A0F0C05060300<127:0>
);
return FFmul_03<UInt>(b) *8+:8>;

```

### Library pseudocode for shared/functions/crypto/FFmul09

```

// FFmul09()
// =====

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/  0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/  0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/  0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/  0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/  0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/  0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/  0xB0219102F263D34434A5158676E757C<127:0> :
        /*8*/  0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/  0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/  0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/  0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :

```

```

        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt>(b)*8+:8>;

```

### Library pseudocode for shared/functions/crypto/FFmul0B

```

// FFmul0B()
// =====

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*
            F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt>(b)*8+:8>;

```

### Library pseudocode for shared/functions/crypto/FFmul0D

```

// FFmul0D()
// =====

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*
            F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C7166B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :

```

```

        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt>(b) *8+:8>;

```

### Library pseudocode for shared/functions/crypto/FFmul0E

```

// FFmul0E()
// =====

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBFB5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt>(b) *8+:8>;

```

### Library pseudocode for shared/functions/crypto/ROL

```

// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);

```

### Library pseudocode for shared/functions/crypto/SHA256hash

```

// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, bool
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);

```

```

maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
x<127:96> = t + x<127:96>;
y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
<y, x> = ROL(y : x, 32);
return (if part1 then x else y);

```

### **Library pseudocode for shared/functions/crypto/SHashchoose**

```

// SHashchoose()
// =====

bits(32) SHashchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);

```

### **Library pseudocode for shared/functions/crypto/SHashhashSIGMA0**

```

// SHashhashSIGMA0()
// =====

bits(32) SHashhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);

```

### **Library pseudocode for shared/functions/crypto/SHashhashSIGMA1**

```

// SHashhashSIGMA1()
// =====

bits(32) SHashhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);

```

### **Library pseudocode for shared/functions/crypto/SHamajority**

```

// SHAmajority()
// =====

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));

```

### **Library pseudocode for shared/functions/crypto/SHAparity**

```

// SHAparity()
// =====

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);

```

### **Library pseudocode for shared/functions/crypto/Sbox**

```

// Sbox()
// =====
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = (
        /*          F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05<127:0> :
        /*E*/ 0x2b679a762abe04c3aa44132649860699<127:0> :
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62<127:0> :
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6<127:0> :
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8<127:0> :
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35<127:0> :
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887<127:0> :
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e<127:0> :
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1<127:0> :
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3<127:0> :
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f<127:0> :
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51<127:0> :
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8<127:0> :
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0<127:0> :
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684<127:0> :
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948<127:0>
    );
    constant integer sboxindex = 255 - UInt(sboxin);
    sboxout = Elem[sboxstring, sboxindex, 8];
    return sboxout;

```

### **Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress**

```

// ClearExclusiveByAddress()
// =====
// Clear the global Exclusives monitors for all PEs EXCEPT processorid
// record any part of the physical address region of size bytes starting
// at paddress
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor
// is also cleared if it records any part of the address region.

ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer bytes);

```

### **Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal**

```

// ClearExclusiveLocal()
// =====
// Clear the local Exclusives monitor for the specified processorid.

ClearExclusiveLocal(integer processorid);

```

### **Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus**

```

// ExclusiveMonitorsStatus()
// =====
// Returns '0' to indicate success if the last memory write by this PE
// to the same physical address region endorsed by ExclusiveMonitorsPass()
// Returns '1' to indicate failure if address translation resulted in a
// page fault or if the physical address region was not valid.

```

```
// physical address.  
  
bit ExclusiveMonitorsStatus();
```

### Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// IsExclusiveGlobal()  
// =====  
// Return TRUE if the global Exclusives monitor for processorid includes  
// the physical address region of size bytes starting at paddress.  
  
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, int
```

### Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// IsExclusiveLocal()  
// =====  
// Return TRUE if the local Exclusives monitor for processorid includes  
// the physical address region of size bytes starting at paddress.  
  
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, int
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// MarkExclusiveGlobal()  
// =====  
// Record the physical address region of size bytes starting at paddress  
// the global Exclusives monitor for processorid.  
  
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// MarkExclusiveLocal()  
// =====  
// Record the physical address region of size bytes starting at paddress  
// the local Exclusives monitor for processorid.  
  
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer
```

### Library pseudocode for shared/functions/exclusive/ProcessorID

```
// ProcessorID()  
// =====  
// Return the ID of the currently executing PE.  
  
integer ProcessorID();
```

### Library pseudocode for shared/functions/extension/HaveBF16Ext

```

// HaveBF16Ext()
// =====
// Returns TRUE if AArch64 BFLOAT16 instruction support is implemented,
boolean HaveBF16Ext()
    return IsFeatureImplemented(FEAT_BF16);

```

### Library pseudocode for shared/functions/extension/HaveFeatABLE

```

// HaveFeatABLE()
// =====
// Returns TRUE if support for linking watchpoints to address matching
// breakpoints is implemented, and FALSE otherwise.
boolean HaveFeatABLE()
    return IsFeatureImplemented(FEAT_ABLE);

```

### Library pseudocode for shared/functions/extension/HaveInt8MatMulExt

```

// HaveInt8MatMulExt()
// =====
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction su
// implemented, and FALSE otherwise
boolean HaveInt8MatMulExt()
    return IsFeatureImplemented(FEAT_I8MM);

```

### Library pseudocode for shared/functions/extension/HaveSoftwareLock

```

// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        return FALSE;
    if IsFeatureImplemented(FEAT_DoPD) && component != Component\_CTI th
        return FALSE;
    case component of
        when Component\_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software L
        when Component\_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Loc
        when Component\_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Loc
        otherwise
            Unreachable\(\);

```

### Library pseudocode for shared/functions/extension/HaveTME

```

// HaveTME()
// =====

```

```
boolean HaveTME()
    return IsFeatureImplemented(FEAT_TME);
```

### Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Archit
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture funct
```

### Library pseudocode for shared/functions/extension/ InsertIESBBeforeException

```
// InsertIESBBeforeException()
// =====
// Returns an implementation defined choice whether to insert an implicit
// barrier before exception.
// If SCLTR_ELx.IESB is 1 when an exception is generated to ELx, any per
// SError interrupt must be taken before executing any instructions in
// However, this can be before the branch to the exception handler is m

boolean InsertIESBBeforeException(bits(2) el)
    return (IsFeatureImplemented(FEAT_IESTB) && boolean IMPLEMENTATION_D
        "Has Implicit Error Synchronization Barrier before Exception
```

### Library pseudocode for shared/functions/extension/ IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// =====
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

boolean IsG1ActivityMonitorImplemented(integer i);
```

### Library pseudocode for shared/functions/extension/ IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// =====
// Returns TRUE if a G1 activity monitor offset is implemented for the co
// and FALSE otherwise.

boolean IsG1ActivityMonitorOffsetImplemented(integer i);
```

### Library pseudocode for shared/functions/externalaborts/ AArch32.PEErroState

```

// AArch32.PEErrorHandler()
// =====
// Returns the error state by PE on taking an SError Interrupt
// to AArch32 level.

ErrorState AArch32.PEErrorHandler(FaultRecord fault)
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
        ReportErrorAsUC()) then
            return ErrorState_UC;
    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;
    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;
    return ErrorState_UEO;

```

### **Library pseudocode for shared/functions/externalaborts/ AArch64.PEErrorHandler**

```

// AArch64.PEErrorHandler()
// =====
// Returns the error state by PE on taking a Synchronous
// or Asynchronous exception.

ErrorState AArch64.PEErrorHandler(FaultRecord fault)
    if !IsExternalSyncAbort(fault) && ExtAbortToA64(fault) then
        if ReportErrorAsUncategorized() then
            return ErrorState_Uncategorized;
        if ReportErrorAsIMPDEF() then
            return ErrorState_IMPDEF;
    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
        ReportErrorAsUC()) then
        return ErrorState_UC;
    if !StateIsRecoverable() || ReportErrorAsUEU() then
        if IsExternalSyncAbort(fault) then // Implies taken to AArch64
            return ErrorState_UC;
        else
            return ErrorState_UEU;
    if (ActionRequired() || ReportErrorAsUER()) then
        return ErrorState_UER;
    return ErrorState_UEO;

```

### **Library pseudocode for shared/functions/externalaborts/ActionRequired**

```

// ActionRequired()
// =====
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

```

```
boolean ActionRequired();
```

#### **Library pseudocode for shared/functions/externalaborts/ ClearPendingPhysicalSError**

```
// ClearPendingPhysicalSError()  
// =====  
// Clear a pending physical SError interrupt.  
  
ClearPendingPhysicalSError();
```

#### **Library pseudocode for shared/functions/externalaborts/ ClearPendingVirtualSError**

```
// ClearPendingVirtualSError()  
// =====  
// Clear a pending virtual SError interrupt.  
  
ClearPendingVirtualSError()  
    if ELUsingAArch32(EL2) then  
        HCR.VA = '0';  
    else  
        HCR_EL2.VSE = '0';
```

#### **Library pseudocode for shared/functions/externalaborts/ErrorisContained**

```
// ErrorisContained()  
// =====  
// Return an implementation specific value:  
// TRUE if Error is contained by the PE, FALSE otherwise.  
  
boolean ErrorisContained();
```

#### **Library pseudocode for shared/functions/externalaborts/ErrorIsSynchronized**

```
// ErrorIsSynchronized()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is synchronized by any synchronization event  
// FALSE otherwise.  
  
boolean ErrorIsSynchronized();
```

#### **Library pseudocode for shared/functions/externalaborts/ExtAbortToA64**

```
// ExtAbortToA64()  
// =====  
// Returns TRUE if synchronous exception is being taken to A64 exception  
// level.
```

```

boolean ExtAbortToA64(FaultRecord fault)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault)
                            (IsFeatureImplemented(FEAT_RAS) && HCR_EL2.
                                IsExternalAbort(fault)) || IsDebugException(fault) && MDCR_EL2.TDE == '1');

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_curr[].EA == '1' && IsExternalAbort(fault) || IsExternalSyncAbort(fault.statuscode);

    return route_to_aarch64 && IsExternalSyncAbort(fault.statuscode);

```

### **Library pseudocode for shared/functions/externalaborts/FaultIsCorrected**

```

// FaultIsCorrected()
// =====
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

boolean FaultIsCorrected();

```

### **Library pseudocode for shared/functions/externalaborts/GetPendingPhysicalSError**

```

// GetPendingPhysicalSError()
// =====
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

FaultRecord GetPendingPhysicalSError();

```

### **Library pseudocode for shared/functions/externalaborts/HandleExternalAbort**

```

// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                    AddressDescriptor memaddrdesc, integer size,
                    AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal}
            (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode != 0));

    fault = NoFault(accdesc);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) && !IsExternalAbortTakenSynchronously(memretstatus, iswrite, m))

```

```

size, accdesc)) then
    if fault.statuscode == Fault_SyncParity then
        fault.statuscode = Fault_AsyncParity;
    else
        fault.statuscode = Fault_AsyncExternal;

    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memretstatus.merrorstate;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
        else
            AArch64.Abort(memaddrdesc.vaddress, fault);

    else
        PendSErrorInterrupt(fault);

```

### **Library pseudocode for shared/functions/externalaborts/ HandleExternalReadAbort**

```

// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor
                      integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc)

```

### **Library pseudocode for shared/functions/externalaborts/ HandleExternalTTWAbort**

```

// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean
                                  AddressDescriptor memaddrdesc,
                                  AccessDescriptor accdesc, integer size,
                                  FaultRecord input_fault)

    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite,
                                         size, accdesc)) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update
    // the fault type
    if IsExternalSyncAbort(output_fault) then

```

```

        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if IsFeatureImplemented(FEAT_RAS) then
        output_fault.merrorstate = memretstatus.merrorstate;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    return output_fault;

```

### **Library pseudocode for shared/functions/externalaborts/ HandleExternalWriteAbort**

```

// HandleExternalWriteAbort ()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor
                        integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc)

```

### **Library pseudocode for shared/functions/externalaborts/ IsExternalAbortTakenSynchronously**

```

// IsExternalAbortTakenSynchronously()
// =====
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus,
                                         boolean iswrite,
                                         AddressDescriptor desc,
                                         integer size,
                                         AccessDescriptor accdesc);

```

### **Library pseudocode for shared/functions/externalaborts/ IsPhysicalSErrorPending**

```

// IsPhysicalSErrorPending()
// =====
// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();

```

### **Library pseudocode for shared/functions/externalaborts/ IsSErrorEdgeTriggered**

```

// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered()
    if IsFeatureImplemented(FEAT_DoubleFault) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";

```

### **Library pseudocode for shared/functions/externalaborts/ IsSynchronizablePhysicalSErrorPending**

```

// IsSynchronizablePhysicalSErrorPending()
// =====
// Returns TRUE if a synchronizable physical SError interrupt is pending.

boolean IsSynchronizablePhysicalSErrorPending();

```

### **Library pseudocode for shared/functions/externalaborts/ IsVirtualSErrorPending**

```

// IsVirtualSErrorPending()
// =====
// Return TRUE if a virtual SError interrupt is pending.

boolean IsVirtualSErrorPending()
    if ELUsingAArch32\(EL2\) then
        return HCR.VA == '1';
    else
        return HCR_EL2.VSE == '1';

```

### **Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt**

```

// PendSErrorInterrupt()
// =====
// Pend the SError Interrupt.

PendSErrorInterrupt(FaultRecord fault);

```

### **Library pseudocode for shared/functions/externalaborts/ ReportErrorAsIMPDEF**

```
// ReportErrorAsIMPDEF ()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is IMPDEF, FALSE otherwise.  
  
boolean ReportErrorAsIMPDEF();
```

### **Library pseudocode for shared/functions/externalaborts/ReportErrorAsUC**

```
// ReportErrorAsUC ()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is Uncontainable, FALSE otherwise.  
  
boolean ReportErrorAsUC();
```

### **Library pseudocode for shared/functions/externalaborts/ReportErrorAsUER**

```
// ReportErrorAsUER ()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is Recoverable, FALSE otherwise.  
  
boolean ReportErrorAsUER();
```

### **Library pseudocode for shared/functions/externalaborts/ReportErrorAsUEU**

```
// ReportErrorAsUEU ()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is Unrecoverable, FALSE otherwise.  
  
boolean ReportErrorAsUEU();
```

### **Library pseudocode for shared/functions/externalaborts/ ReportErrorAsUncategorized**

```
// ReportErrorAsUncategorized ()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is uncategorized, FALSE otherwise.  
  
boolean ReportErrorAsUncategorized();
```

### **Library pseudocode for shared/functions/externalaborts/StatelsRecoverable**

```

// StateIsRecoverable()
// =====
// Return an implementation specific value:
// returns TRUE if PE State is unrecoverable else FALSE.

boolean StateIsRecoverable();

```

## Library pseudocode for shared/functions/float/bfloat/BFAdd

```

// BFAdd()
// =====
// Non-widening BFLOAT16 addition used by SVE2 instructions.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;
    return BFAdd(op1, op2, fpcr, fpexc);

// BFAdd()
// =====
// Non-widening BFLOAT16 addition following computational behaviors
// corresponding to instructions that read and write BFLOAT16 values.
// Calculates op1 + op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);

        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 2*N);
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 2*N);
        elseif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result
                result_sign = if rounding == FPRounding_NEGINF then '1'
                result = FPZero(result_sign, 2*N);

```

```

        else
            result = FPRoundBF(result_value, fpcr, rounding, fpexc,
            if fpexc then FPPProcessDenorms(type1, type2, 2*N, fpcr);
        return result<2*N-1:N>;

```

### Library pseudocode for shared/functions/float/bfloat/BFAdd\_ZA

```

// BFAdd_ZA()
// =====
// Non-widening BFLOAT16 addition used by SME2 ZA-targeting instruction

bits(N) BFAdd_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    boolean fpexc = FALSE;
    FPCRType fpcr = fpcr_in;
    fpcr.DN = '1';           // Generate default NaN values
    return BFAdd(op1, op2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/bfloat/BFDotAdd

```

// BFDotAdd()
// =====
// BFLOAT16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(N) BFDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                  bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr_in)
    assert N == 32;
    FPCRType fpcr = fpcr_in;

    bits(N) prod;

    bits(N) result;
    if !IsFeatureImplemented(FEAT_EBF16) || fpcr.EBF == '0' then      // SIMD
        prod = FPAdd\_BF16(BFMulH(op1_a, op2_a), BFMulH(op1_b, op2_b));
        result = FPAdd\_BF16(addend, prod);
    else
        boolean isbf16 = TRUE;
        boolean fpexc = FALSE; // Do not generate floating-point exceptions
        fpcr.DN = '1';           // Generate default NaN values
        prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbf16, fpexc);
        result = FPAdd(addend, prod, fpcr, fpexc);

    return result;

```

### Library pseudocode for shared/functions/float/bfloat/BFIInfinity

```

// BFInfinity()
// =====

bits(N) BFInfinity(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Ones(E) : Zeros(F);

```

### Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```

// BFMatMulAdd()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;

    bits(N) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPC
Elem[result, 2*i + j, 32] = sum;

    return result;

```

### Library pseudocode for shared/functions/float/bfloat/BFMax

```

// BFMax()
// =====
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCRTyPe fpcr)
    boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &
        return BFMax(op1, op2, fpcr, altfp);

// BFMax()
// =====
// BFloat16 maximum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' deter-
// if the function should use alternative floating-point behavior.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCRTyPe fpcr_in, boolean altfp

    assert N == 16;

```

```

FPCRTypc fpcr = fpcr_in;
boolean fpexc = TRUE;
FPRounding rounding = FPRoundingMode(fpcr);
boolean done;
bits(2*N) result;

bits(2*N) op1_s = op1 : Zeros(N);
bits(2*N) op2_s = op2 : Zeros(N);
(type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

if altp && type1 == FPTypc_Zero && type2 == FPTypc_Zero && sign1 !
    // Alternate handling of zeros with differing sign
    return BFZero(sign2, N);
elseif altp && (type1 IN {FPTypc_SNan, FPTypc_QNaN} || type2 IN {FP
    // Alternate handling of NaN inputs
    FPProcessException(FPEExc_InvalidOp, fpcr);
    return (if type2 == FPTypc_Zero then BFZero(sign2, N) else op2);

(done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr);
if !done then
    FPTypc fptype;
    bit sign;
    real value;
    if value1 > value2 then
        (fptype,sign,value) = (type1,sign1,value1);
    else
        (fptype,sign,value) = (type2,sign2,value2);
    if fptype == FPTypc_Infinity then
        result = FPInfinity(sign, 2*N);
    elseif fptype == FPTypc_Zero then
        sign = sign1 AND sign2;                      // Use most positive s
        result = FPZero(sign, 2*N);
    else
        if altp then      // Denormal output is not flushed to zero
            fpcr.FZ = '0';
        result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);
    if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

return result<2*N-1:N>;

```

## Library pseudocode for shared/functions/float/bfloat/BFMaxNum

```

// BFMaxNum()
// =====
// BFloat16 maximum number following computational behaviors correspond
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after round
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRTypc fpcr)

assert N == 16;
boolean fpexc = TRUE;
boolean isbfloat16 = TRUE;
bits(N) op1 = op1_in;
bits(N) op2 = op2_in;

```

```

boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
bits(N) result;

(type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbf16);
(type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbf16);

boolean type1_nan = type1 IN {FPTYPE_QNaN, FPTYPE_SNan};
boolean type2_nan = type2 IN {FPTYPE_QNaN, FPTYPE_SNan};

if !(altfp && type1_nan && type2_nan) then
    // Treat a single quiet-NaN as -Infinity.
    if type1 == FPTYPE_QNaN && type2 != FPTYPE_QNaN then
        op1 = BFInfinity('1', N);
    elseif type1 != FPTYPE_QNaN && type2 == FPTYPE_QNaN then
        op2 = BFInfinity('1', N);

boolean altfmaxfmin = FALSE;      // Do not use alternate NaN handling.
result = BFMax(op1, op2, fpcr, altfmaxfmin);

return result;

```

## Library pseudocode for shared/functions/float/bfloat/BFMin

```

// BFMin()
// ======
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCRTyp e fpcr)
    boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
    return BFMin(op1, op2, fpcr, altfp);

// BFMin()
// ======
// BFloat16 minimum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCRTyp e fpcr_in, boolean altfp)

    assert N == 16;
    FPCRTyp e fpcr = fpcr_in;
    boolean fpexc = TRUE;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPTYPE_Zero && type2 == FPTYPE_Zero && sign1 != sign2
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elseif altfp && (type1 IN {FPTYPE_SNan, FPTYPE_QNaN} || type2 IN {FPTYPE_SNan, FPTYPE_QNaN})
        // Alternate handling of NaN inputs

```

```

FPProcessException(FPExc_InvalidOp, fpcr);
return (if type2 == FPType_Zero then BFZero(sign2, N) else op2);

(done, result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr);
if !done then
    FPTYPE fptype;
    bit sign;
    real value;
    if value1 < value2 then
        (fptype, sign, value) = (type1, sign1, value1);
    else
        (fptype, sign, value) = (type2, sign2, value2);
    if fptype == FPType_Infinity then
        result = FPInfinity(sign, 2*N);
    elsif fptype == FPType_Zero then
        sign = sign1 OR sign2;                                // Use most negative sign
        result = FPZero(sign, 2*N);
    else
        if altpfp then      // Denormal output is not flushed to zero
            fpcr.FZ = '0';
        result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

    if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;

```

## Library pseudocode for shared/functions/float/bfloat/BFMinNum

```

// BFMinNum()
// =====
// BFloat16 minimum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMinNum(bits(N) op1_in, bits(N) op2_in, FPCRTYPE fpcr)

    assert N == 16;
    boolean fpexc = TRUE;
    boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    boolean altpfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altpfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity('0', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity('0', N);

```

```

boolean altfmaxfmin = FALSE;      // Do not use alternate NaN handling
result = BFMin(op1, op2, fpcr, altfmaxfmin);

return result;

```

## Library pseudocode for shared/functions/float/bfloat/BFMul

```

// BFMul()
// ======
// Non-widening BFLOAT16 multiply used by SVE2 instructions.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);

// BFMul()
// ======
// Non-widening BFLOAT16 multiply following computational behaviors
// corresponding to instructions that read and write BFLOAT16 values.
// Calculates op1 * op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPEExc InvalidOp, fpcr);
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elseif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = FPRoundBF(value1*value2, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;

```

## Library pseudocode for shared/functions/float/bfloat/BFMulAdd

```

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr,
    boolean fpexc = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add following computational behavior
// corresponding to instructions that read and write BFloat16 values.
// Calculates addend + op1*op2 with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr,
    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) addend_s = addend : Zeros(N);
    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (typeA,signA,valueA) = FPUnpack(addend_s, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    inf1 = (type1 == FPType\_Infinity);
    inf2 = (type2 == FPType\_Infinity);
    zero1 = (type1 == FPType\_Zero);
    zero2 = (type2 == FPType\_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend_s, op1_s, fpcr);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAAArch32() && fpcr.AH == 0)
        if typeA == FPType\_QNaN && ((inf1 && zero2) || (zero1 && inf2))
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPEExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType\_Infinity);
        zeroA = (typeA == FPType\_Zero);

        // Determine sign and type product will have if it does not cause
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of
        // infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPEExc\_InvalidOp, fpcr);

    // Other cases involving infinities produce an infinity of the same sign

```

```

        elseif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 2*N);
        elseif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 2*N);

        // Cases where the result is exactly zero and its sign is not
        // rounding mode are additions of same-signed zeros.
        elseif zeroA && zeroP && signA == signP then
            result = FPZero(signA, 2*N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result
                result_sign = if rounding == FPRounding_NEGINF then '1'
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc,
                if !invalidop && fpexc then
                    FPProcessDenorms3(typeA, type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAddH

```

// BFMulAddH()
// =====
// Used by BFMLALB, BFMLALT, BFMLSIB and BFMLSLT instructions.

bits(N) BFMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    assert N == 32;
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    FPCRTtype fpcr = fpcr_in;
    boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    boolean fpexc = !altfp;

    if altfp then fpcr.<FIZ,FZ> = '11';

    if altfp then fpcr.RMode = '00';
    return FPMulAdd(addend, value1, value2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAddH\_ZA

```

// BFMulAddH_ZA()
// =====
// Used by SME2 ZA-targeting BFMLAL and BFMLSL instructions.

bits(N) BFMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    assert N == 32;
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    return FPMulAdd_ZA(addend, value1, value2, fpcr);

```

## Library pseudocode for shared/functions/float/bfloat/BFMulAdd\_ZA

```
// BFMulAdd_ZA()
// =====
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting i

bits(N) BFMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType
    boolean fpexc = FALSE;
    FPCRType fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/bfloat/BFMulH

```
// BFMulH()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(2*N) BFMulH(bits(N) op1, bits(N) op2)

    assert N == 16;
    bits(2*N) result;

    FPCRType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType\_QNaN || type2 == FPType\_QNaN then
        result = FPDefaultNaN(fpcr, 2*N);
    else
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elseif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = BFRound(value1*value2, 2*N);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =====

bits(N) BFNeg(bits(N) op)
    assert N == 16;
    boolean honor_altp = TRUE; // Honor alternate handling
    return BFNeg(op, honor_altp);

// BFNeg()
```

```

// =====

bits(N) BFNeg(bits(N) op, boolean honor_altp)
{
    assert N == 16;
    if honor_altp && !UsingAArch32() && IsFeatureImplemented(FEAT_AFP)
        FPCRTyp fpcr = FPCR[];
        if fpcr.AH == '1' then
            boolean fpexc = FALSE;
            boolean isbf16 = TRUE;
            (fptype, -, -) = FPUnpackBase(op, fpcr, fpexc, isbf16);
            if fptype IN {FPTyp_SNaN, FPTyp_QNaN} then
                return op; // When fpcr.AH=1, sign of NaN has no effect
            else
                return NOT(op<N-1>) : op<N-2:0>;
}

```

## Library pseudocode for shared/functions/float/bfloat/BFRound

```

// BFRound()
// =====
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFfloat16 computation behavior.

bits(N) BFRound(real op, integer N)
{
    assert N == 32;
    assert op != 0.0;
    bits(N) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction
    constant integer minimum_exp = -126; constant integer E = 8; constant integer F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign, N);

    // Start creating the exponent value for the result. Start by biasing
    // so that the minimum exponent becomes 1, lower values 0 (indicating
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last
    int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0
    error = mantissa * 2.0^F - Real(int_mant);
}
```

```

// Round to Odd
if error != 0.0 then
    int_mant<0> = '1';

// Deal with overflow and generate result.
if biased_exp >= 2^E - 1 then
    result = FPInfinity(sign, N);           // Overflows generate appropriate
else
    result = sign : biased_exp<(N-2)-F:0> : int_mant<F-1:0>;

return result;

```

## Library pseudocode for shared/functions/float/bfloat/BFSUB

```

// BFSUB()
// ======
// Non-widening BFLOAT16 subtraction used by SVE2 instructions.

bits(N) BFSUB(bits(N) op1, bits(N) op2, FPCRTYPE fpcr)
    boolean fpexc = TRUE;
    return BFSUB(op1, op2, fpcr, fpexc);

// BFSUB()
// ======
// Non-widening BFLOAT16 subtraction following computational behaviors
// corresponding to instructions that read and write BFLOAT16 values.
// Calculates op1 - op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFSUB(bits(N) op1, bits(N) op2, FPCRTYPE fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPTYPE Infinity);
        inf2 = (type2 == FPTYPE Infinity);
        zero1 = (type1 == FPTYPE Zero);
        zero2 = (type2 == FPTYPE Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPPProcessException(FPExc InvalidOp, fpcr);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', 2*N);
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', 2*N);
        elseif zero1 && zero2 && sign1 == NOT(sign2) then

```

```

        result = FPZero(sign1, 2*N);
    else
        result_value = value1 - value2;
        if result_value == 0.0 then // Sign of exact zero result
            result_sign = if rounding == FPRounding_NEGINF then '1'
            result = FPZero(result_sign, 2*N);
        else
            result = FPRoundBF(result_value, fpcr, rounding, fpexc,
                if fpexc then FPPProcessDenorms(type1, type2, 2*N, fpcr));
    return result<2*N-1:N>;

```

### Library pseudocode for shared/functions/float/bfloat/BFSub\_ZA

```

// BFSub_ZA()
// =====
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions
bits(N) BFSub_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    boolean fpexc = FALSE;
    FPCRType fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/bfloat/BFUnpack

```

// BFUnpack()
// =====
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities.
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPTyp, bit, real) BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign = fpval<15>;
        exp = fpval<14:7>;
        frac = fpval<6:0> : Zeros(16);
    else // N == 32
        sign = fpval<31>;
        exp = fpval<30:23>;
        frac = fpval<22:0>;

    FPTyp fptype;
    real value;
    if IsZero(exp) then
        fptype = FPTyp_Zero; value = 0.0; // Fixed Flush to Zero
    elseif IsOnes(exp) then

```

```

        if IsZero(frac) then
            fptype = FPType_Infinity; value = 2.0^1000000;
        else // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
        if sign == '1' then value = -value;
    return (fptype, sign, value);

```

## Library pseudocode for shared/functions/float/bfloat/BFZero

```

// BFZero()
// =====

bits(N) BFZero(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);

```

## Library pseudocode for shared/functions/float/bfloat/FPAdd\_BF16

```

// FPAdd_BF16()
// =====
// Single-precision add following BFLOAT16 computation behaviors.

bits(N) FPAdd_BF16(bits(N) op1, bits(N) op2)

    assert N == 32;
    bits(N) result;

    FPCRTyp fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, N);
    else
        inf1 = (type1 == FPType_Inf);
        inf2 = (type2 == FPType_Inf);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInf('0', N);
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInf('1', N);
        elseif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0', N); // Positive sign when Round
            else

```

```

        result = BFRound(result_value, N);

    return result;

```

## Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```

// FPConvertBF()
// =====
// Converts a single-precision OP to BFloat16 value using the
// Round to Nearest Even rounding mode when executed from AArch64 state
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(N DIV 2) FPConvertBF(bits(N) op, FPCRType fpcr_in, FPRounding roun

    assert N == 32;
    constant integer halfsize = N DIV 2;
    FPCRType fpcr = fpcr_in;
    FPRounding rounding = rounding_in;
    bits(N) result;                                // BF16 value in top
    boolean altpf = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &
    boolean fpexc = !altpf;                         // Generate no float
    if altpf then fpcr.<FIZ,FZ> = '11';           // Flush denormal i
    if altpf then rounding = FPRounding\_TIEEVEN;      // Use RNE rounding

    // Unpack floating-point operand, with always flush-to-zero if fpcr.A
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPType\_SNan || fptype == FPType\_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, N);
        else
            result = FPConvertNaN(op, N);
        if fptype == FPType\_SNan then
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPType\_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType\_Zero then
        result = FPZero(sign, N);
    else
        result = FPRoundBF(value, fpcr, rounding, fpexc, N);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<(2*halfsize)-1:halfsize>;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(N DIV 2) FPConvertBF(bits(N) op, FPCRType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));

```

## Library pseudocode for shared/functions/float/bfloat/FPRoundBF

```

// FPRoundBF()
// =====
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of

```

```

// floating-point exceptions.

bits(N) FPRoundBF(real op, FPCRType fpcr, FPRounding rounding, boolean
    assert N == 32;
    boolean isbf16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbf16, fpexc, N);

```

### Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```

// FixedToFP()
// =====

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType
                    FPRounding rounding, integer N)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;

```

### Library pseudocode for shared/functions/float/fpabs/FPAbs

```

// FPAbs()
// =====

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        FPCRTyp fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;           // When fpcr.AH=1, sign of NaN has no
            end;
        end;
    end;
    return '0' : op<N-2:0>;

```

### Library pseudocode for shared/functions/float/fpadd/FPAdd

```

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;           // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPType Infinity);   inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);       zero2 = (type2 == FPType Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result can be either
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);
        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;

```

### Library pseudocode for shared/functions/float/fpadd/FPAdd\_ZA

```

// FPAdd_ZA()
// =====
// Calculates op1+op2 for SME2 ZA-targeting instructions.

bits(N) FPAdd_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;           // Do not generate floating-point exceptions
    fpcr.DN = '1';                 // Generate default NaN values
    return FPAdd(op1, op2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/fpcompare/FPCompare

```

// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRTy

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPType\_SNaN, FPType\_QNaN} || type2 IN {FPType\_SNaN, FPType\_QNaN}
        result = '0011';
        if type1 == FPType\_SNaN || type2 == FPType\_SNaN || signal_nans
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

### **Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ**

```

// FPCompareEQ()
// =====

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRTy fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType\_SNaN, FPType\_QNaN} || type2 IN {FPType\_SNaN, FPType\_QNaN}
        result = FALSE;
        if type1 == FPType\_SNaN || type2 == FPType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

### **Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE**

```

// FPCompareGE()
// =====

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRTy fpcr)

```

```

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

boolean result;
if type1 IN {FPType\_SNaN, FPType\_QNaN} || type2 IN {FPType\_SNaN, FPType\_QNaN}
    result = FALSE;
FPProcessException(FPEExc\_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by
    result = (value1 >= value2);
FPProcessDenorms(type1, type2, N, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```

// FPCompareGT()
// =====

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

boolean result;
if type1 IN {FPType\_SNaN, FPType\_QNaN} || type2 IN {FPType\_SNaN, FPType\_QNaN}
    result = FALSE;
FPProcessException(FPEExc\_InvalidOp, fpcr);
else
    // All non-NaN cases can be evaluated on the values produced by
    result = (value1 > value2);
FPProcessDenorms(type1, type2, N, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpconvert/FPConvert

```

// FPConvert()
// =====

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding, intec
assert M IN {16,32,64};
assert N IN {16,32,64};
bits(M) result;

// Unpack floating-point operand optionally with flush-to-zero.
(fptype,sign,value) = FPUnpackCV(op, fpcr);

alt_hp = (M == 16) && (fpcr.AHP == '1');

```

```

if fptype == FPTYPE_SNAN || fptype == FPTYPE_QNAN then
    if alt_hp then
        result = FPZero(sign, M);
    elseif fpcr.DN == '1' then
        result = FPDefaultNaN(fpcr, M);
    else
        result = FPCConvertNaN(op, M);
    if fptype == FPTYPE_SNAN || alt_hp then
        FPPProcessException(FPEXC_INVALIDOP, fpcr);
    elseif fptype == FPTYPE_INFINITY then
        if alt_hp then
            result = sign:Ones(M-1);
            FPPProcessException(FPEXC_INVALIDOP, fpcr);
        else
            result = FPInfinity(sign, M);
    elseif fptype == FPTYPE_ZERO then
        result = FPZero(sign, M);
    else
        result = FPRoundCV(value, fpcr, rounding, M);
        FPPProcessDenorm(fptype, N, fpcr);

    return result;

// FPCConvert()
// =====

bits(M) FPCConvert(bits(N) op, FPCRType fpcr, integer M)
    return FPCConvert(op, fpcr, FPRoundingMode(fpcr), M);

```

## Library pseudocode for shared/functions/float/fpconvertnan/FPCConvertNaN

```

// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;

```

### **Library pseudocode for shared/functions/float/fpcrtype/FPCRTyPe**

```
type FPCRTyPe;
```

### **Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM**

```

// FPDecodeRM()
// =====

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

    FPRounding result;
    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF; // P
        when '11' result = FPRounding_NEGINF; // M

    return result;

```

### **Library pseudocode for shared/functions/float/fpdecoderounding/ FPDecodeRounding**

```

// FPDecodeRounding()
// =====

```

```

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF; // P
        when '10' return FPRounding_NEGINF; // M
        when '11' return FPRounding_ZERO; // Z

```

## Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```

// FPDefaultNaN()
// =====

bits(N) FPDefaultNaN(integer N)
    FPCRTyp fpcr = FPCR[];
    return FPDefaultNaN(fpcr, N);

bits(N) FPDefaultNaN(FPCRTyp fpcr, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elseif N == 32 then 8 else 1);
    constant integer F = N - (E + 1);
    bit sign = if IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() then
        0;
    else
        1;

    bits(E) exp = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;

```

## Library pseudocode for shared/functions/float/fpdiv/FPDiv

```

// FPDIV()
// =====

bits(N) FPDIV(bits(N) op1, bits(N) op2, FPCRTyp fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        inf1 = type1 == FPTyp Infinity;
        inf2 = type2 == FPTyp Infinity;
        zero1 = type1 == FPTyp Zero;
        zero2 = type2 == FPTyp Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elseif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elseif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2, N);
    else
        result = FPDiv(type1, sign1, value1, type2, sign2, value2, fpcr);

```

```

        else
            result = FPRound(value1/value2, fpcr, N);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

## Library pseudocode for shared/functions/float/fpdot/FPDot

```

// FPDot()
// =====
// Calculates single-precision result of 2-way 16-bit floating-point dot
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloating16'
// determines whether input operands are BFloating16 or half-precision type
// and 'fpexc' controls the generation of floating-point exceptions.

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op1_c,
               bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr, boolean isbfloating16, in
               boolean fpexc = TRUE;           // Generate floating-point exceptions
               return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloating16, fpexc, N);

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op1_c,
               bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr_in, boolean isbfloating16,
               FPCRType fpcr = fpcr_in;

assert N == 32;
bits(N) result;
boolean done;
fpcr.AHP = '0';           // Ignore alternative half-precision option
rounding = FPRoundingMode(fpcr);

(type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloating16);
(type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloating16);
(type2_a,sign2_a,value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloating16);
(type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloating16);

inf1_a = (type1_a == FPType Infinity); zero1_a = (type1_a == FPType Zero);
inf1_b = (type1_b == FPType Infinity); zero1_b = (type1_b == FPType Zero);
inf2_a = (type2_a == FPType Infinity); zero2_a = (type2_a == FPType Zero);
inf2_b = (type2_b == FPType Infinity); zero2_b = (type2_b == FPType Zero);

(done,result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                 op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

if (((inf1_a && zero2_a) || (zero1_a && inf2_a)) &&
    ((inf1_b && zero2_b) || (zero1_b && inf2_b))) then
    result = FPDefaultNaN(fpcr, N);
    if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

if !done then
    // Determine sign and type products will have if it does not cause
    // Operation.
    signPa = sign1_a EOR sign2_a;
    signPb = sign1_b EOR sign2_b;
    infPa = inf1_a || inf2_a;
    infPb = inf1_b || inf2_b;

```

```

zeroPa = zero1_a || zero2_a;
zeroPb = zero1_b || zero2_b;

// Non SNaN-generated Invalid Operation cases are multiplies of
// by infinity and additions of opposite-signed infinities.
invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
              (inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && inf2_a) ||
              (infPa && zero2_b) || (zero1_a && inf2_b) || (inf1_a && inf2_b) ||
              (inf1_b && inf2_a) || (inf1_b && zero2_a) || (zero1_b && inf2_a) ||
              (inf1_a && zero2_a) || (zero1_b && inf2_b)) && !fpexc;

if invalidop then
    result = FPDefaultNaN(fpcr, N);
    if fpexc then FPPProcessException(FPExc_InvalidOp, fpcr);

// Other cases involving infinities produce an infinity of the same sign.
elseif (infPa && signPa == '0') || (infPb && signPb == '0') then
    result = FPInfinity('0', N);
elseif (infPa && signPa == '1') || (infPb && signPb == '1') then
    result = FPInfinity('1', N);

// Cases where the result is exactly zero and its sign is not determined by
// rounding mode are additions of same-signed zeros.
elseif zeroPa && zeroPb && signPa == signPb then
    result = FPZero(signPa, N);

// Otherwise calculate fused sum of products and round it.
else
    result_value = (value1_a * value2_a) + (value1_b * value2_b);
    if result_value == 0.0 then // Sign of exact zero result can't be determined
        result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign, N);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc, N);

return result;

```

## Library pseudocode for shared/functions/float/fpdot/FPDotAdd

```

// FPDotAdd()
// =====
// Half-precision 2-way dot-product and add to single-precision.

bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                  bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRTType fpcr,
                  assert N == 32;

bits(N) prod;
boolean isbf16 = FALSE;
boolean fpexc = TRUE; // Generate floating-point exceptions
prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbf16, fpexc, N);
result = FPAAdd(addend, prod, fpcr, fpexc);

return result;

```

## Library pseudocode for shared/functions/float/fpdot/FPDotAdd\_ZA

```

// FPDotAdd_ZA()
// =====
// Half-precision 2-way dot-product and add to single-precision

```

```

// for SME ZA-targeting instructions.

bits(N) FPDotAdd_ZA(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2)
                     bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType
FPCRType fpcr = fpcr_in;
assert N == 32;

bits(N) prod;
boolean isbf16 = FALSE;
boolean fpexc = FALSE; // Do not generate floating-point exception
fpcr.DN = '1'; // Generate default NaN values
prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbf16, fpexc, N);
result = FPAdd(addend, prod, fpcr, fpexc);

return result;

```

### Library pseudocode for shared/functions/float/fpexc/FPExc

```

// FPExc
// =====

enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                    FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm}

```

### Library pseudocode for shared/functions/float/fpinfinity/FPIinfinity

```

// FPIinfinity()
// =====

bits(N) FPIinfinity(bit sign, integer N)

assert N IN {16, 32, 64};
constant integer E = (if N == 16 then 5 elseif N == 32 then 8 else 16);
constant integer F = N - (E + 1);
bits(E) exp = Ones(E);
bits(F) frac = Zeros(F);

return sign : exp : frac;

```

### Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```

// FPMatMulAdd()
// =====
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize)

assert N == esize * 2 * 2;
bits(N) result;
bits(esize) prod0, prod1, sum;

for i = 0 to 1
    for j = 0 to 1

```

```

        sum   = Elem[addend, 2*i + j, esize];
        prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                           Elem[op2, 2*j + 0, esize], fpcr);
        prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                           Elem[op2, 2*j + 1, esize], fpcr);
        sum   = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
Elem[result, 2*i + j, esize] = sum;

    return result;
}

```

## Library pseudocode for shared/functions/float/fpmax/FPMax

```

// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altpf = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &
    return FPMax(op1, op2, fpcr, altpf);

// FPMax()
// =====
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altpf' determines
// if the function should use alternative floating-point behavior.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altpf)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if altpf && type1 == FPTyp Zero && type2 == FPTyp Zero && sign1 !=
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elseif altpf && (type1 IN {FPTyp SNaN, FPTyp QNaN} || type2 IN {FPTyp
        // Alternate handling of NaN inputs
        FPProcessException(FPExc InvalidOp, fpcr);
        return (if type2 == FPTyp Zero then FPZero(sign2, N) else op2)

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTyp Infinity then
            result = FPInfinity(sign, N);
        elseif fptype == FPTyp Zero then
            sign = sign1 AND sign2;           // Use most positive sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a tr

```

```

// for a denormalized number even though the result is exact
rounding = FPRoundingMode(fpcr);
if altfp then      // Denormal output is not flushed to zero
    fpcr.FZ = '0';
    fpcr.FZ16 = '0';

result = FPRound(value, fpcr, rounding, TRUE, N);

FPProcessDenorms(type1, type2, N, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```

// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Ones(E-1) : '0';
frac = Ones(F);

return sign : exp : frac;

```

### Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```

// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

assert N IN {16,32,64};
bits(N) op1 = op1_in;
bits(N) op2 = op2_in;
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&

if !(altfp && type1_nan && type2_nan) then
    // Treat a single quiet-NaN as -Infinity.
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1', N);
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1', N);

altfmaxfmin = FALSE;      // Restrict use of FMAX/FMIN NaN propagatio
result = FPMax(op1, op2, fpcr, altfmaxfmin);

return result;

```

## Library pseudocode for shared/functions/float/fpmerge/IsMerging

```
// IsMerging()
// =====
// Returns TRUE if the output elements other than the lowest are taken
// the destination register.

boolean IsMerging(FPCRTYPE fpcr)
    bit nep = (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
               !IsFullA64Enabled\(\) then '0' else fpcr.NEP);
    return IsFeatureImplemented(FEAT_AFP) && !UsingAArch32\(\) && nep ==
```

## Library pseudocode for shared/functions/float/fpmin/FPMIn

```

        result = FPZero(sign, N);
    else
        // The use of FPRound() covers the case where there is a tr
        // for a denormalized number even though the result is exact
        rounding = FPRoundingMode(fpcr);
        if altpf then      // Denormal output is not flushed to zero
            fpcr.FZ = '0';
            fpcr.FZ16 = '0';

        result = FPRound(value, fpcr, rounding, TRUE, N);

FPPprocessDenorms(type1, type2, N, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpmminum/FPMinNum

```

// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

assert N IN {16,32,64};
bits(N) op1 = op1_in;
bits(N) op2 = op2_in;
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNan};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNan};
boolean altpf = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &

if !(altpf && type1_nan && type2_nan) then
    // Treat a single quiet-NaN as +Infinity.
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinit('0', N);
    elseif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinit('0', N);

altfmaxfmin = FALSE;      // Restrict use of FMAX/FMIN NaN propagatio
result = FPMin(op1, op2, fpcr, altfmaxfmin);

return result;

```

### Library pseudocode for shared/functions/float/fpmul/FPMul

```

// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)

assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPPprocessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity);

```

```

inf2 = (type2 == FPType_Infinity);
zero1 = (type1 == FPType_Zero);
zero2 = (type2 == FPType_Zero);

if (inf1 && zero2) || (zero1 && inf2) then
    result = FPDefaultNaN(fpcr, N);
    FPProcessException(FPExc_InvalidOp, fpcr);
elsif inf1 || inf2 then
    result = FPInfinity(sign1 EOR sign2, N);
elsif zero1 || zero2 then
    result = FPZero(sign1 EOR sign2, N);
else
    result = FPRound(value1*value2, fpcr, N);

FPProcessDenorms(type1, type2, N, fpcr);

return result;

```

## Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```

// FPMulAdd()
// =====

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fp
    boolean fpexc = TRUE;           // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =====
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argum
// supplies the FPCR control bits, and 'fpexc' controls the generation
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
                  FPCRType fpcr, boolean fpexc)

assert N IN {16,32,64};

(typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
rounding = FPRoundingMode(fpcr);
inf1 = (type1 == FPType Infinity); zero1 = (type1 == FPType Zero);
inf2 = (type2 == FPType Infinity); zero2 = (type2 == FPType Zero);

(done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2);

if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH =
    if typeA == FPType QNaN && ((inf1 && zero2) || (zero1 && inf2))
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

if !done then
    infA = (typeA == FPType Infinity); zeroA = (typeA == FPType Zero);
    // Determine sign and type product will have if it does not cau
    // Invalid Operation.

```

```

signP = sign1 EOR sign2;
infP = inf1 || inf2;
zeroP = zero1 || zero2;

// Non SNaN-generated Invalid Operation cases are multiplies of
// by infinity and additions of opposite-signed infinities.
invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infB);

if invalidop then
    result = FPDefaultNaN(fpcr, N);
    if fpexc then FPPProcessException(FPExc_InvalidOp, fpcr);
// Other cases involving infinities produce an infinity of the same sign.
elseif (infA && signA == '0') || (infP && signP == '0') then
    result = FPInfinity('0', N);
elseif (infA && signA == '1') || (infP && signP == '1') then
    result = FPInfinity('1', N);

// Cases where the result is exactly zero and its sign is not determined by
// rounding mode are additions of same-signed zeros.
elseif zeroA && zeroP && signA == signP then
    result = FPZero(signA, N);

// Otherwise calculate numerical result and round it.
else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result can't be determined.
        result_sign = if rounding == FPRounding_NEGINF then '1'
        result = FPZero(result_sign, N);
    else
        result = FPRound(result_value, fpcr, rounding, fpexc, N);

if !invalidop && fpexc then
    FPPProcessDenorms3(typeA, type1, type2, N, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd\_ZA

```

// FPMulAdd_ZA()
// =====
// Calculates addend + op1*op2 with a single rounding for SME ZA-target
// instructions.

bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;           // Do not generate floating-point exceptions.
    fpcr.DN = '1';                  // Generate default NaN values
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    boolean fpexc = TRUE;           // Generate floating-point exceptions

```

```

        return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
                   FPCRType fpcr, boolean fpexc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPType Infinity); zero1 = (type1 == FPType Zero);
    inf2 = (type2 == FPType Infinity); zero2 = (type2 == FPType Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, c

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAAArch32() && fpcr.AH =
        if typeA == FPType QNaN && ((inf1 && zero2) || (zero1 && inf2))
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType Infinity); zeroA = (typeA == FPType Zero);

        // Determine sign and type product will have if it does not cause
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same
        elseif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elseif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined
        // by rounding mode are additions of same-signed zeros.
        elseif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result determined
                result_sign = if rounding == FPRounding_NEGINF then '1'
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

```

```

    if !invalidop && fpexc then
        FPProcessDenorm(typeA, N, fpcr);

    return result;

```

### Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH\_ZA

```

// FPMulAddH_ZA()
// =====
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

bits(N) FPMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
                      FPCRType fpcr = fpcr_in;
                      boolean fpexc = FALSE;           // Do not generate floating-point exceptions
                      fpcr.DN = '1';                  // Generate default NaN values
                      return FPMulAddH(addend, op1, op2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```

// FPProcessNaNs3H()
// =====

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
                                      bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                      FPCRType fpcr, boolean fpexc)

assert N IN {32,64};

bits(N) result;
FPType type_nan;
// When TRUE, use alternative NaN propagation rules.
boolean altpf = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
boolean op1_nan = type1 IN {FPType SNaN, FPType QNaN};
boolean op2_nan = type2 IN {FPType SNaN, FPType QNaN};
boolean op3_nan = type3 IN {FPType SNaN, FPType QNaN};
if altpf then
    if (type1 == FPType SNaN || type2 == FPType SNaN || type3 == FPType SNaN)
        type_nan = FPType SNaN;
    else
        type_nan = FPType QNaN;

boolean done;
if altpf && op1_nan && op2_nan && op3_nan then                                // <n> regi
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2,
elsif altpf && op2_nan && (op1_nan || op3_nan) then                                // <n> regi
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2,
elsif altpf && op3_nan && op1_nan then                                         // <m> regi
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op3,
elsif type1 == FPType SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elsif type2 == FPType SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr,
elsif type3 == FPType SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr,
elsif type1 == FPType QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);

```

```

        elseif type2 == FPType_QNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
        elseif type3 == FPType_QNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
        else
            done = FALSE; result = Zeros(N); // 'Don't care' result
    return (done, result);

```

### **Library pseudocode for shared/functions/float/fpmulx/FPMulX**

```

// FPMulX()
// =====

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRTtype fpcr)

    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elseif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elseif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);
    FPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

### **Library pseudocode for shared/functions/float/fpneg/FPNeg**

```

// FPNeg()
// =====

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        FPCRTyp fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTYPE_SNaN, FPTYPE_QNaN} then

                return op;           // When fpcr.AH=1, sign of NaN has no effect

            return NOT(op<N-1>) : op<N-2:0>;

```

### **Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive**

```

// FPOnePointFive()
// =====

bits(N) FPOnePointFive(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elseif N == 32 then 8 else 16);
    constant integer F = N - (E + 1);
    exp = '0':_Ones_(E-1);
    frac = '1':_Zeros_(F-1);
    result = sign : exp : frac;

    return result;

```

### **Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm**

```

// FPProcessDenorm()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPTYPE fptype, integer N, FPCRTYP fpcr)
    boolean altpfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && N != 16;
    if altpfp && fptype == FPTYPE_Denormal then
        FPProcessException(FPEExc_InputDenorm, fpcr);

```

### **Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms**

```

// FPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPTYPE type1, FPTYPE type2, integer N, FPCRTYP fpcr)
    boolean altpfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && N != 16;

```

```

    if altp && N != 16 && (type1 == FPTYPE_Denormal || type2 == FPTYPE_Denormal)
        FPPProcessException(FPEExc_InputDenorm, fpcr);

```

### **Library pseudocode for shared/functions/float/fpprocessdenorms/ FPPProcessDenorms3**

```

// FPPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorms3(FPTYPE type1, FPTYPE type2, FPTYPE type3, integer N,
    boolean altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &
    if altp && N != 16 && (type1 == FPTYPE_Denormal || type2 == FPTYPE_Denormal)
        if type3 == FPTYPE_Denormal) then
            FPPProcessException(FPEExc_InputDenorm, fpcr);

```

### **Library pseudocode for shared/functions/float/fpprocessdenorms/ FPPProcessDenorms4**

```

// FPPProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPPProcessDenorms4(FPTYPE type1, FPTYPE type2, FPTYPE type3, FPTYPE type4,
    boolean altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &
    if altp && N != 16 && (type1 == FPTYPE_Denormal || type2 == FPTYPE_Denormal
        || type3 == FPTYPE_Denormal || type4 == FPTYPE_Denormal) then
            FPPProcessException(FPEExc_InputDenorm, fpcr);

```

### **Library pseudocode for shared/functions/float/fpprocessexception/ FPPProcessException**

```

// FPPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPPProcessException(FPEExc except, FPCRTYPE fpcr)

    integer cumul;
    // Determine the cumulative exception bit number
    case except of
        when FPEExc_InvalidOp      cumul = 0;
        when FPEExc_DivideByZero   cumul = 1;
        when FPEExc_Overflow      cumul = 2;
        when FPEExc_Underflow     cumul = 3;
        when FPEExc_Inexact       cumul = 4;
        when FPEExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if (fpcr<enable> == '1' && (!IsFeatureImplemented(FEAT_SME) || PSTA))
        IsFullA64Enabled()) then
            // Trapping of the exception enabled.

```

```

// It is IMPLEMENTATION_DEFINED whether the enable bit may be set
// and if so then how exceptions and in what order that they may be
// accumulated before calling FPTrappedException().
bits(8) accumulated_exceptions = GetAccumulatedFPEExceptions();
accumulated_exceptions<cumul> = '1';
if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point
    if UsingAArch32() then
        AArch32.FPTrappedException(accumulated_exceptions);
    else
        is_ase = IsAISEInstruction();
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    else
        // The exceptions generated by this instruction are accumulated
        // FPTrappedException is called later during its execution,
        // instruction is executed. This field is cleared at the start
        SetAccumulatedFPEExceptions(accumulated_exceptions);
elseif UsingAArch32() then
    // Set the cumulative exception bit
    FPSCR<cumul> = '1';
else
    // Set the cumulative exception bit
    FPSR<cumul> = '1';

return;

```

## Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```

// FPProcessNaN()
// =====

bits(N) FPProcessNaN(FPTYPE fptype, bits(N) op, FPCRTYPE fpcr)
    boolean fpexc = TRUE;      // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// =====
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control
// The 'fpexc' argument controls the generation of exceptions, regardless
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPTYPE fptype, bits(N) op, FPCRTYPE fpcr, boolean fpexc)
    assert N IN {16,32,64};
    assert fptype IN {FPTYPE_QNaN, FPTYPE_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPTYPE_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);

```

```
    return result;
```

## Library pseudocode for shared/functions/float/fpprocessnans/FPPProcessNaNs

```
// FPPProcessNaNs ()
// =====

(boolean, bits(N)) FPPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                    bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exception
    return FPPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPPProcessNaNs ()
// =====
//
// The boolean part of the return value says whether a NaN has been found
// processed. The bits(N) part is only relevant if it has and supplies
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behavior for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                    FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE; result = FPPprocessNaN(type_nan, op1, fpcr, fpexc);
    elseif type1 == FPType_SNaN then
        done = TRUE; result = FPPprocessNaN(type1, op1, fpcr, fpexc);
    elseif type2 == FPType_SNaN then
        done = TRUE; result = FPPprocessNaN(type2, op2, fpcr, fpexc);
    elseif type1 == FPType_QNaN then
        done = TRUE; result = FPPprocessNaN(type1, op1, fpcr, fpexc);
    elseif type2 == FPType_QNaN then
        done = TRUE; result = FPPprocessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result

    return (done, result);
```

## Library pseudocode for shared/functions/float/fpprocessnans3/FPPProcessNaNs3

```

// FPProcessNaNs3()
// =====

(boolean, bits(N)) FPProcessNaNs3(FPTYPE type1, FPTYPE type2, FPTYPE ty
                                bits(N) op1, bits(N) op2, bits(N) op3
                                FPCRTYPE fpcr)
    boolean fpexc = TRUE;      // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found
// processed. The bits(N) part is only relevant if it has and supplies
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPTYPE type1, FPTYPE type2, FPTYPE ty
                                bits(N) op1, bits(N) op2, bits(N) op3
                                FPCRTYPE fpcr, boolean fpexc)

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN {FPTYPE_SNaN, FPTYPE_QNaN};
    boolean op2_nan = type2 IN {FPTYPE_SNaN, FPTYPE_QNaN};
    boolean op3_nan = type3 IN {FPTYPE_SNaN, FPTYPE_QNaN};

    boolean altpfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
FPTYPE type_nan;
    if altpfp then
        if type1 == FPTYPE_SNaN || type2 == FPTYPE_SNaN || type3 == FPTYPE_SNaN
            type_nan = FPTYPE_SNaN;
        else
            type_nan = FPTYPE_QNaN;

    boolean done;
    if altpfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elseif altpfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elseif altpfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
    elseif type1 == FPTYPE_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elseif type2 == FPTYPE_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elseif type3 == FPTYPE_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    elseif type1 == FPTYPE_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elseif type2 == FPTYPE_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elseif type3 == FPTYPE_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    else

```

```

        done = FALSE;    result = Zeros(N); // 'Don't care' result
    return (done, result);
}

```

## Library pseudocode for shared/functions/float/fpprocessnans4/ FPProcessNaNs4

```

// FPProcessNaNs4()
// =====
// The boolean part of the return value says whether a NaN has been found
// processed. The bits(N) part is only relevant if it has and supplies
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(N)) FPProcessNaNs4(FPTYPE type1, FPTYPE type2, FPTYPE type3,
                                  bits(N DIV 2) op1, bits(N DIV 2) op2,
                                  bits(N DIV 2) op4, FPCRTYPE fpcr, boolean fpexc);

assert N == 32;

bits(N) result;
boolean done;
// The FPCR.AH control does not affect these checks
if type1 == FPTYPE_SNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type1, op1, fpexc));
elseif type2 == FPTYPE_SNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type2, op2, fpexc));
elseif type3 == FPTYPE_SNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type3, op3, fpexc));
elseif type4 == FPTYPE_SNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type4, op4, fpexc));
elseif type1 == FPTYPE_QNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type1, op1, fpexc));
elseif type2 == FPTYPE_QNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type2, op2, fpexc));
elseif type3 == FPTYPE_QNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type3, op3, fpexc));
elseif type4 == FPTYPE_QNaN then
    done = TRUE;    result = FPCConvertNaN(FPProcessNaN(type4, op4, fpexc));
else
    done = FALSE;   result = Zeros(N); // 'Don't care' result
return (done, result);
}

```

## Library pseudocode for shared/functions/float/fprecipestimate/ FPRecipEstimate

```

// FPRecipEstimate()
// =====
bits(N) FPRecipEstimate(bits(N) operand, FPCRTYPE fpcr_in)

assert N IN {16,32,64};
FPCRTYPE fpcr = fpcr_in;
}

```

```

bits(N) result;
boolean overflow_to_inf;
// When using alternative floating-point behavior, do not generate
// floating-point exceptions, flush denormal input and output to zero
// and use RNE rounding mode.
boolean altpf = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &&
boolean fpexc = !altpf;
if altpf then fpcr.<FIZ,FZ> = '11';
if altpf then fpcr.RMode = '00';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

FPRounding rounding = FPRoundingMode(fpcr);
if fptype == FPType_SNan || fptype == FPType_QNaN then
    result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elseif fptype == FPType_Infinity then
    result = FPZero(sign, N);
elseif fptype == FPType_Zero then
    result = FPInfinity(sign, N);
    if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
elseif (
    (N == 16 && Abs(value) < 2.0^-16) ||
    (N == 32 && Abs(value) < 2.0^-128) ||
    (N == 64 && Abs(value) < 2.0^-1024)
) then
    case rounding of
        when FPRounding_TIEEVEN
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO
            overflow_to_inf = FALSE;
    result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNc
if fpexc then
    FPProcessException(FPExc_Overflow, fpcr);
    FPProcessException(FPExc_Inexact, fpcr);
elseif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
    && (
        (N == 16 && Abs(value) >= 2.0^14) ||
        (N == 32 && Abs(value) >= 2.0^126) ||
        (N == 64 && Abs(value) >= 2.0^1022)
) then
    // Result flushed to zero of correct sign
result = FPZero(sign, N);

// Flush-to-zero never generates a trapped exception.
if UsingAArch32() then
    FPSCR.UFC = '1';
else
    if fpexc then FPSR.UFC = '1';
else
    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in s
    // calculate result exponent. Scaled value has copied sign bit,
    // exponent = 1022 = double-precision biased version of -1,
    // fraction = original fraction
    bits(52) fraction;
    integer exp;
    case N of

```

```

when 16
    fraction = operand<9:0> : Zeros(42);
    exp = UInt(operand<14:10>);
when 32
    fraction = operand<22:0> : Zeros(29);
    exp = UInt(operand<30:23>);
when 64
    fraction = operand<51:0>;
    exp = UInt(operand<62:52>);

if exp == 0 then
    if fraction<51> == '0' then
        exp = -1;
        fraction = fraction<49:0>:'00';
    else
        fraction = fraction<50:0>:'0';

integer scaled;
boolean increasedprecision = N==32 && IsFeatureImplemented(FEAT);

if !increasedprecision then
    scaled = UInt('1':fraction<51:44>);
else
    scaled = UInt('1':fraction<51:41>);

integer result_exp;
case N of
    when 16 result_exp = 29 - exp; // In range 29-30 = -1 to 0
    when 32 result_exp = 253 - exp; // In range 253-254 = -1 to 0
    when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 0

// Scaled is in range 256 .. 511 or 2048 .. 4095 range represented
// fixed-point number in range [0.5 .. 1.0].
estimate = RecipEstimate(scaled, increasedprecision);

// Estimate is in the range 256 .. 511 or 4096 .. 8191 represented
// fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit,
// high-order bits from estimate, and exponent calculated above.
if !increasedprecision then
    fraction = estimate<7:0> : Zeros(44);
else
    fraction = estimate<11:0> : Zeros(40);

if result_exp == 0 then
    fraction = '1' : fraction<51:1>;
elseif result_exp == -1 then
    fraction = '01' : fraction<51:2>;
    result_exp = 0;

case N of
    when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
    when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
    when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;

```

## Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```

// RecipEstimate()
// =====
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;                                // Round to nearest
        a = a*2+1;
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2;                                         // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;                            // Round to nearest
        a = a*2+1;
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5;   // Error cannot be exactly 0.5
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;

    return r;

```

## Library pseudocode for shared/functions/float/fprecpx/FPRecpX

```

// FPRecpX()
// =====

bits(N) FPRecpX(bits(N) op, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    boolean isbf16 = FALSE;
    constant (E, F, -) = FPBits(N, isbf16);
    bits(N)           result;
    bits(E)           exp;
    bits(E)           max_exp;
    constant bits(F) frac = Zeros(F);

    boolean altp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    boolean fpexc = !altp;                               // Generate no floating-point
    if altp then fpcr.<FIZ,FZ> = '11';      // Flush denormal input and
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
    exp = op<F+:E>;
    max_exp = Ones(E) - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else

```

```

        if IsZero(exp) then                                // Zero and denormals
            result = ZeroExtend(sign:max_exp:frac, N);
        else                                              // Infinities and normals
            result = ZeroExtend(sign:NOT(exp):frac, N);

    return result;

```

### Library pseudocode for shared/functions/float/fpround/FPBits

```

// FPBits()
// ======
// Returns the minimum exponent, numbers of exponent and fraction bits.

(integer, integer, integer) FPBits(integer N, boolean isbf16)
    integer E;
    integer F;
    integer minimum_exp;
    if N == 16 then
        minimum_exp = -14;   E = 5;   F = 10;
    elseif N == 32 && isbf16 then
        minimum_exp = -126;  E = 8;   F = 7;
    elseif N == 32 then
        minimum_exp = -126;  E = 8;   F = 23;
    else // N == 64
        minimum_exp = -1022; E = 11;  F = 52;

    return (E, F, minimum_exp);

```

### Library pseudocode for shared/functions/float/fpround/FPRound

```

// FPRound()
// ======
// Generic conversion from precise, unbounded real data type to IEEE float

bits(N) FPRound(real op, FPCRType fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// ======
// For directed FP conversion, includes an explicit 'rounding' argument

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
    boolean fpexc = TRUE;      // Generate floating-point exceptions
    return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// ======
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, boolean fpexc)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbf16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbf16, fpexc, N);

```

## Library pseudocode for shared/functions/float/fround/FPRoundBase

```
// FPRoundBase()
// =====
// For BFloat16, includes an explicit 'isbf16' argument.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbf16, fpexc, N);

// FPRoundBase()
// =====
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding,
    boolean isbf16, boolean fpexc, integer N)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent
    constant (E, F, minimum_exp) = FPBits(N, isbf16);

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0; exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0; exponent = exponent + 1;

    // When TRUE, detection of underflow occurs after rounding and the
    // denormalized number for single and double precision values occur
    altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AFP == TRUE;

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' &&
        exponent < minimum_exp)) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign, N);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
```

```

error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

// Start creating the exponent value for the result. Start by biasing
// so that the minimum exponent becomes 1, lower values 0 (indicating
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and results
// in the Underflow exception being trapped. This applies before rounding
boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFPTrapEnabled());
if !altpf && biased_exp == 0 && (error != 0.0 || trapped_UF) then
    if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altpf then

    case rounding of
        when FPRounding_TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained < 0));
            round_up = (error > 0.5 || (error == 0.5 && int_mant < 0));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then // Rounded up
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained = int_mant_unconstrained DIV 2;

// Deal with flush-to-zero and underflow after rounding if FPCR.AE=1
if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0
    // the result of unconstrained rounding is less than the minimum
    if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)
        if fpexc then
            FPSR.UFC = '1';
            FPProcessException(FPExc_Inexact, fpcr);
            return FPZero(sign, N);
    elseif error != 0.0 || trapped_UF then
        if fpexc then FPProcessException(FPExc_Underflow, fpcr);
else // altpf == FALSE
    case rounding of

```

```

        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0>));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then          // Rounded up from denormalized to
            biased_exp = 1;
        if int_mant == 2^(F+1) then     // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;

    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then // Single, double or IEEE half
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign, N) else FPMIN;
            if fpexc then FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0; // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros;
    else
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0; // Ensure that an Inexact exception does not
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros;

    // Deal with Inexact exception.
    if error != 0.0 then
        if fpexc then FPProcessException(FPExc_Inexact, fpcr);

    return result;

```

## Library pseudocode for shared/functions/float/fround/FPRoundCV

```

// FPRoundCV()
// =====
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr_in, FPRounding rounding, integ
FPCRType fpcr = fpcr_in;
fpcr.FZ16 = '0';
boolean fpexc = TRUE; // Generate floating-point exceptions

```

```

boolean isbfloat16 = FALSE;
return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

```

### Library pseudocode for shared/functions/float/fprounding/FPRounding

```

// FPRounding
// =====
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

enumeration FPRounding {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF, FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};

```

### Library pseudocode for shared/functions/float/fproundingmode/ FPRoundingMode

```

// FPRoundingMode()
// =====
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);

```

### Library pseudocode for shared/functions/float/froundint/FPRoundInt

```

// FPRoundInt()
// =====

// Round op to nearest integral floating point value using rounding mode
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to
// bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, bool
// assert rounding != FPRounding\_ODD;
// assert N IN {16,32,64};

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altpf = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AFP;
fpexc = !altpf;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

bits(N) result;
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    result = FPProcessNaN(fptype, op, fpcr);
elseif fptype == FPType_Infinity then
    result = FPIinfinity(sign, N);
elseif fptype == FPType_Zero then
    result = FPZero(sign, N);
else
    // Extract integer component.
    int_result = RoundDown(value);

```

```

        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result < 0));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result < 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;

```

## Library pseudocode for shared/functions/float/fprroundintn/FPRoundIntN

```

// FPRoundIntN()
// =====

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, int
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.A
    fpexc = !altp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPType_SNan, FPType_QNaN, FPType_Infinity} then
        if N == 32 then

```

```

        exp = 126 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
    else
        exp = 1022+intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
        FPPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0);
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result > 0);

        if round_up then int_result = int_result + 1;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1)+1;

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
                FPPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact
            if real_result == 0.0 then
                result = FPZero(sign, N);
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

            // Generate inexact exceptions.
            if error != 0.0 then
                FPPProcessException(FPExc_Inexact, fpcr);

        return result;

```

## Library pseudocode for shared/functions/float/fprsqrtestimate/ FPRSqrtEstimate

```

// FPRSqrtEstimate()
// =====

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() &
    boolean fpexc = !altp;
    if altp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType SNaN || fptype == FPType QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType Infinity then
        result = FPZero('0', N);
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in
        // evenness or oddness of the exponent unchanged, and calculate
        // Scaled value has copied sign bit, exponent = 1022 or 1021 =
        // biased version of -1 or -2, fraction = original fraction ext

        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

            integer scaled;
            boolean increasedprecision = N==32 && IsFeatureImplemented(FEAT_AFP);

            if !increasedprecision then
                if exp<0> == '0' then
                    scaled = UInt('1':fraction<51:44>);
                else
                    scaled = UInt('01':fraction<51:45>);
            end;
        end;
    end;
end;

```

```

        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

        integer result_exp;
        case N of
            when 16 result_exp = ( 44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 represented
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Z
            when 32
                if !increasedprecision then
                    result = '0' : result_exp<N-25:0> : estimate<7:0>:Z
                else
                    result = '0' : result_exp<N-25:0> : estimate<11:0>:Z
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Z

        return result;
    
```

## Library pseudocode for shared/functions/float/fprsqrtestimate/ **RecipSqrtEstimate**

```

// RecipSqrtEstimate()
// =====
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then
            a = a*2+1;                                // 0.25 .. 0.5
                                                // a in units of 1/512 round
                                                // 0.5 .. 1.0
        else
            a = (a >> 1) << 1;                      // Discard bottom bit
            a = (a+1)*2;                            // a in units of 1/256 round
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a)
    r = (b+1) DIV 2;                           // Round to nearest

```

```

        assert 256 <= r && r < 512;
else
    assert 1024 <= a && a < 4096;
    real real_val;
    real error;
    integer int_val;

    if a < 2048 then                                // 0.25... 0.5
        a = a*2 + 1;                               // Take 10 bits of fraction
        real_val = Real(a)/2.0;
    else
        a = (a >> 1) << 1;                      // 0.5..1.0
        a = a+1;                                 // Discard bottom bit
        real_val = Real(a);                      // Take 10 bits of fraction

    real_val = Sqrt(real_val);                     // This number will lie in [0.5, 1]
                                                // Round to nearest even fraction
    real_val = real_val * Real(2^47);             // The integer is the size of the
    int_val = RoundDown(real_val);                // Calculate rounding value
    error = real_val - Real(int_val);
    round_up = error > 0.5;                      // Error cannot be exactly zero
    if round_up then int_val = int_val+1;

    real_val = Real(2^65)/Real(int_val);          // Lies in the range 4096
    int_val = RoundDown(real_val);                // Round that (to nearest even)
    error = real_val - Real(int_val);
    round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'))
    if round_up then int_val = int_val+1;

    r = int_val;
    assert 4096 <= r && r < 8192;

return r;

```

## Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```

// FPSqrt()
// =====

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elseif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elseif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elseif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;

```

## Library pseudocode for shared/functions/float/fpsub/FPSub

```

// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;           // Generate floating-point exceptions
    return FPSub(op1, op2, fpcr, fpexc);

// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
    
```

```

        elseif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result can be either
                result_sign = if rounding == FPRounding_NEGINF then '1'
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);
        if fpexc then FPPProcessDenorms(type1, type2, N, fpcr);

    return result;

```

### Library pseudocode for shared/functions/float/fpsub/FPSub\_ZA

```

// FPSub_ZA()
// =====
// Calculates op1-op2 for SME2 ZA-targeting instructions.

bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;           // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPSub(op1, op2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/fpthree/FPTThree

```

// FPTThree()
// =====

bits(N) FPTThree(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elseif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;

```

### Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```

// FPToFixed()
// =====

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned,
                  FPRounding rounding, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};

```

```

assert fbits >= 0;
assert rounding != FPRounding_ODD;

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.A
fpexc = !altp;

// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

// If NaN, set cumulative flag or take exception.
if fptype == FPType_SNAN || fptype == FPType_QNaN then
    FPProcessException(FPEExc_InvalidOp, fpcr);

// Scale by fractional bits and produce integer rounded towards min
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.
boolean round_up;
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_result<0));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 && int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

if round_up then int_result = int_result + 1;

// Generate saturated result and exceptions.
(result, overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPEExc_InvalidOp, fpcr);
elsif error != 0.0 then
    FPProcessException(FPEExc_Inexact, fpcr);

return result;

```

## Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```

// FPToFixedJS()
// =====

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64, int

assert M == 64 && N == 32;

// If FALSE, never generate Input Denormal floating-point exceptions

```

```

fpexc_idenorm = !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32())

// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

z = '1';
// If NaN, set cumulative flag or take exception.
if fptype == FPTYPE_SNAN || fptype == FPTYPE_QNAN then
    FPProcessException(FPEExc_InvalidOp, fpcr);
    z = '0';

int_result = RoundDown(value);
error = value - Real(int_result);

// Determine whether supplied rounding mode requires an increment.

round_it_up = (error != 0.0 && int_result < 0);
if round_it_up then int_result = int_result + 1;

integer result;
if int_result < 0 then
    result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
else
    result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

// Generate exceptions.
if int_result < -(2^31) || int_result > (2^31)-1 then
    FPProcessException(FPEExc_InvalidOp, fpcr);
    z = '0';
elsif error != 0.0 then
    FPProcessException(FPEExc_Inexact, fpcr);
    z = '0';
elsif sign == '1' && value == 0.0 then
    z = '0';
elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
    z = '0';

if fptype == FPTYPE_INFINITY then result = 0;

return (result<N-1:0>, z);

```

## Library pseudocode for shared/functions/float/fptwo/FPTwo

```

// FPTwo()
// =====

bits(N) FPTwo(bit sign, integer N)

assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 1);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = Zeros(F);
result = sign : exp : frac;

return result;

```

## Library pseudocode for shared/functions/float/fptype/FPTType

```
// FPType
// =====

enumeration FPType {FPType_Zero,
                     FPType_Denormal,
                     FPType_Nonzero,
                     FPType_Infinity,
                     FPType_QNaN,
                     FPType_SNaN};
```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// =====

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUnpack()
// =====

//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in, boolean f
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// =====

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr, boolean f
    boolean isbf16 = FALSE;
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc, isbf16);
    return (fp_type, sign, value);

// FPUnpackBase()
// =====

//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for
// and infinities, is very large in magnitude for infinities, and is 0.0
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls
```

```

// generation of floating-point exceptions and 'isbfloating16' determines whether
// N=16 signifies BFloating16 or half-precision type. Status information is
// directly in the FPSR where appropriate.

(FPTYPE, bit, real) FPUnpackBase(bits(N) fpval, FPCRTYPE fpcr_in, boolean
                                  boolean isbfloating16)

    assert N IN {16,32,64};

    FPCRTYPE fpcr = fpcr_in;

    boolean altp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
    boolean fiz = altp && fpcr.FIZ == '1';
    boolean fz = fpcr.FZ == '1' && !(altp && fpcr.AH == '1');
    real value;
    bit sign;
    FPTYPE fptype;

    if N == 16 && !isbfloating16 then
        sign = fpval<15>;
        exp16 = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPTYPE_Zero; value = 0.0;
            else
                fptype = FPTYPE_Denormal; value = 2.0^-14 * (Real(UINT)
        elseif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN
            if IsZero(frac16) then
                fptype = FPTYPE_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPTYPE_QNaN else FPTYPE_NaN;
                value = 0.0;
            else
                fptype = FPTYPE_Nonzero;
                value = 2.0^(UINT(exp16)-15) * (1.0 + Real(UINT(frac16)) *
        elseif N == 32 || isbfloating16 then
            bits(8) exp32;
            bits(23) frac32;
            if isbfloating16 then
                sign = fpval<15>;
                exp32 = fpval<14:7>;
                frac32 = fpval<6:0> : Zeros(16);
            else
                sign = fpval<31>;
                exp32 = fpval<30:23>;
                frac32 = fpval<22:0>;
            if IsZero(exp32) then
                if IsZero(frac32) then
                    // Produce zero if value is zero.
                    fptype = FPTYPE_Zero; value = 0.0;
                elseif fz || fiz then // Flush-to-zero if FIZ==1 or AH==1
                    fptype = FPTYPE_Zero; value = 0.0;
                    // Check whether to raise Input Denormal floating-point
                    // fpcr.FIZ==1 does not raise Input Denormal exception.
                    if fz then
                        // Denormalized input flushed to zero
                        if fpexc then FPProcessException(FPExc_InputDenorm,

```

```

        else
            fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UIr
        elseif Is Ones(exp32) then
            if IsZero(frac32) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac32<22> == '1' then FPType_QNaN else FPT
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) *

        else // N == 64
            sign = fpval<63>;
            exp64 = fpval<62:52>;
            frac64 = fpval<51:0>;

            if IsZero(exp64) then
                if IsZero(frac64) then
                    // Produce zero if value is zero.
                    fptype = FPType_Zero;  value = 0.0;
                elseif fz || fiz then          // Flush-to-zero if FIZ==1 or AHP
                    fptype = FPType_Zero;  value = 0.0;
                    // Check whether to raise Input Denormal floating-point
                    // fpcr.FIZ==1 does not raise Input Denormal exception.
                    if fz then
                        // Denormalized input flushed to zero
                        if fpexc then FPProcessException(FPExc_InputDenorm,
                    else
                        fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UI
                elseif Is Ones(exp64) then
                    if IsZero(frac64) then
                        fptype = FPType_Infinity;  value = 2.0^1000000;
                    else
                        fptype = if frac64<51> == '1' then FPType_QNaN else FPT
                        value = 0.0;
                else
                    fptype = FPType_Nonzero;
                    value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64))

            if sign == '1' then value = -value;

            return (fptype, sign, value);
    
```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```

// FPUnpackCV()
// =====
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr_in)
FPCRType fpcr = fpcr_in;
fpcr.FZ16 = '0';
boolean fpexc = TRUE;      // Generate floating-point exceptions
(fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);
    
```

## Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// =====

bits(N) FPZero(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 16);
    constant integer F = N - (E + 1);
    exp = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// =====

bits(N) VFPExpandImm(bits(8) imm8, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 16);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>;
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

## Library pseudocode for shared/functions/interrupts/InterruptID

```
// InterruptID
// =====
```

```

enumeration InterruptID {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
    InterruptID_PMBIRQ,
}

```

## Library pseudocode for shared/functions/interrupts/SetInterruptRequestLevel

```

// SetInterruptRequestLevel()
// =====
// Set a level-sensitive interrupt to the specified level.

SetInterruptRequestLevel(InterruptID id, Signal level);

```

## Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```

// AArch64.BranchAddr()
// =====
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    constant integer msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);

```

## Library pseudocode for shared/functions/memory/AccessDescriptor

```

// AccessDescriptor
// =====
// Memory access or translation invocation details that steer architecture

type AccessDescriptor is (
    AccessType acctype,                                // Acting EL for the access
    bits(2) el,                                         // Acting Security State for the access
    SecurityState ss,                                 // Acquire with Sequential Consistency
    boolean acqsc,                                     // FEAT_LRCPC: Acquire with Process Consistency
    boolean acqpc,                                     // Release with Sequential Consistency
    boolean relsc,                                     // Release with Sequential Consistency
)

```

```

        boolean limitedordered,           // FEAT_LOR: Acquire/Release with 1
        boolean exclusive,              // Access has Exclusive semantics
        boolean atomicop,               // FEAT_LSE: Atomic read-modify-write
MemAtomicOp modop,                // FEAT_LSE: The modification operation
        boolean nontemporal,             // Hints the access is non-temporal
        boolean read,                   // Read from memory or only requires
        boolean write,                  // Write to memory or only requires
CacheOp cacheop,                 // DC/IC: Cache operation
CacheOpScope opscope,            // DC/IC: Scope of cache operation
CacheType cachetype,              // DC/IC: Type of target cache
        boolean pan,                    // FEAT_PAN: The access is subject to
        boolean transactional,           // FEAT_TME: Access is part of a transaction
        boolean nonfault,                // SVE: Non-faulting load
        boolean firstfault,              // SVE: First-fault load
        boolean first,                   // SVE: First-fault load for the first
        boolean contiguous,              // SVE: Contiguous load/store not guaranteed
        boolean streamingsve,            // SME: Access made by PE while in stream
        boolean ls64,                   // FEAT_LS64: Accesses by accelerators
        boolean mops,                   // FEAT_MOPS: Memory operation (CPY/MOV)
        boolean rcw,                    // FEAT_THE: Read-Check-Write access
        boolean rcws,                   // FEAT_THE: Read-Check-Write Software
        boolean toplevel,                // FEAT_THE: Translation table walk
VARange varange,                  // FEAT_THE: The corresponding TTBR
        boolean a32lsmrd,                // A32 Load/Store Multiple Data access
        boolean tagchecked,              // FEAT_MTE2: Access is tag checked
        boolean tagaccess,                // FEAT_MTE: Access targets the tag
        boolean ispair,                  // Access represents a Load/Store pair
        boolean highestaddressfirst,      // FEAT_LRCPC3: Highest address is a
MPAMinfo mpam                   // FEAT_MPAM: MPAM information
)

```

## Library pseudocode for shared/functions/memory/AccessType

```

// AccessType
// =====

enumeration AccessType {
    AccessType_IFETCH,      // Instruction FETCH
    AccessType_GPR,          // Software load/store to a General Purpose Register
    AccessType_ASIMD,         // Software ASIMD extension load/store instructions
    AccessType_SVE,           // Software SVE load/store instructions
    AccessType_SME,           // Software SME load/store instructions
    AccessType_IC,             // Sysop IC
    AccessType_DC,             // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,          // Sysop DC {Z,G,GZ}VA
    AccessType_AT,             // Sysop AT
    AccessType_NV2,             // NV2 memory redirected access
    AccessType_SPE,             // Statistical Profiling buffer access
    AccessType_GCS,             // Guarded Control Stack access
    AccessType_TRBE,             // Trace Buffer access
    AccessType_GPTW,             // Granule Protection Table Walk
    AccessType_TTW              // Translation Table Walk
};

```

## Library pseudocode for shared/functions/memory/AddrTop

```

// AddrTop()
// =====
// Return the MSB number of a virtual address in the stage 1 translation
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zeroed
// by the stage 1 translation.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;

```

### Library pseudocode for shared/functions/memory/AlignmentEnforced

```

// AlignmentEnforced()
// =====
// For the active translation regime, determine if alignment is required.

boolean AlignmentEnforced()
    Regime regime = TranslationRegime(PSTATE.EL);

    bit A;
    case regime of
        when Regime_EL3      A = SCTRLR_EL3.A;
        when Regime_EL30     A = SCTRLR.A;
        when Regime_EL2      A = if ELUsingAArch32(EL2) then HSCTRLR.A else 0;
        when Regime_EL20     A = SCTRLR_EL2.A;
        when Regime_EL10     A = if ELUsingAArch32(EL1) then SCTRLR.A else 0;
        otherwise           A = Unreachable();

    return A == '1';

```

### Library pseudocode for shared/functions/memory/Allocation

```

constant bits(2) MemHint_No = '00';          // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';          // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';          // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';         // Read-Allocate, Write-Allocate

```

### Library pseudocode for shared/functions/memory/BigEndian

```

// BigEndian()
// =====

boolean BigEndian(AccessType acctype)
    boolean bigend;
    if IsFeatureImplemented(FEAT_NV2) && acctype == AccessType_NV2 then
        return SCTRLR_EL2.EE == '1';

    if UsingAArch32() then

```

```

        bigend = (PSTATE.E != '0');
    elseif PSTATE.EL == EL0 then
        bigend = (SCTRLR_ELx[]).EOE != '0');
    else
        bigend = (SCTRLR_ELx[]).EE != '0');
return bigend;

```

### **Library pseudocode for shared/functions/memory/BigEndianReverse**

```

// BigEndianReverse()
// =====
bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    if width == 8 then return value;
    constant integer half = width DIV 2;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<w

```

### **Library pseudocode for shared/functions/memory/Cacheability**

```

constant bits(2) MemAttr_NC = '00';           // Non-cacheable
constant bits(2) MemAttr_WT = '10';           // Write-through
constant bits(2) MemAttr_WB = '11';           // Write-back

```

### **Library pseudocode for shared/functions/memory/CreateAccDescA32LSMD**

```

// CreateAccDescA32LSMD()
// =====
// Access descriptor for A32 loads/store multiple general purpose regis
AccessDescriptor CreateAccDescA32LSMD (MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR) ;

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.a32lsmd       = TRUE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescASIMD**

```

// CreateAccDescASIMD()
// =====
// Access descriptor for ASIMD&FP loads/stores

AccessDescriptor CreateAccDescASIMD (MemOp memop, boolean nontemporal, b
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD) ;

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;

```

```

accdesc.pan          = TRUE;
accdesc.streamingsve = InStreamingMode();
if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
    "No tag checking of SIMD&FP loads and stores in Streaming SVE"
    accdesc.tagchecked = FALSE;
else
    accdesc.tagchecked = tagchecked;
accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

return accdesc;

```

### Library pseudocode for shared/functions/memory/ CreateAccDescASIMDAcqRel

```

// CreateAccDescASIMDAcqRel()
// =====
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

AccessDescriptor CreateAccDescASIMDAcqRel(MemOp memop, boolean tagcheck
                                         AccessDescriptor accdesc = NewAccDesc(AccessType ASIMD);

    accdesc.acqpc      = memop == MemOp LOAD;
    accdesc.relsc      = memop == MemOp STORE;
    accdesc.read       = memop == MemOp LOAD;
    accdesc.write      = memop == MemOp STORE;
    accdesc.pan        = TRUE;
    accdesc.streamingsve = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE"
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescAT

```

// CreateAccDescAT()
// =====
// Access descriptor for address translation operations

AccessDescriptor CreateAccDescAT(SecurityState ss, bits(2) el, ATAccess
                                         AccessDescriptor accdesc = NewAccDesc(AccessType AT);

    accdesc.el          = el;
    accdesc.ss          = ss;
    case ataccess of
        when ATAccess Read
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE,
        when ATAccess ReadPAN
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE,
        when ATAccess Write
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE,
        when ATAccess WritePAN
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE,
        when ATAccess Any

```

```

        (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, FALSE,
return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescAcqRel

```

// CreateAccDescAcqRel()
// =====
// Access descriptor for general purpose register loads/stores with ordering

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = memop == MemOp_LOAD;
    accdesc.relsc          = memop == MemOp_STORE;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescAtomicOp

```

// CreateAccDescAtomicOp()
// =====
// Access descriptor for atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire,
                                      boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = acquire;
    accdesc.relsc          = release;
    accdesc.atomicop       = TRUE;
    accdesc.modop          = modop;
    accdesc.read           = TRUE;
    accdesc.write          = TRUE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescDC

```

// CreateAccDescDC()
// =====
// Access descriptor for data cache operations

AccessDescriptor CreateAccDescDC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DC);

    accdesc.cacheop         = cache.cacheop;

```

```

accdesc.cachetype      = cache.cachetype;
accdesc.opscope        = cache.opscope;

return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescDCZero**

```

// CreateAccDescDCZero()
// =====
// Access descriptor for data cache zero operations

AccessDescriptor CreateAccDescDCZero(boolean tagaccess, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DCZero);

    accdesc.write          = TRUE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked    = tagchecked;
    accdesc.tagaccess     = tagaccess;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescExLDST**

```

// CreateAccDescExLDST()
// =====
// Access descriptor for general purpose register loads/stores with exceptions

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean acqsc)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = acqrel && memop == MemOp_LOAD;
    accdesc.relsc          = acqrel && memop == MemOp_STORE;
    accdesc.exclusive      = TRUE;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked    = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescGCS**

```

// CreateAccDescGCS()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack

AccessDescriptor CreateAccDescGCS(bits(2) el, MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el          = el;
    accdesc.read        = memop == MemOp_LOAD;
    accdesc.write       = memop == MemOp_STORE;

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescGCSSS1

```

// CreateAccDescGCSSS1()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack to SSS1

AccessDescriptor CreateAccDescGCSSS1(bits(2) el)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el          = el;
    accdesc.atomicop    = TRUE;
    accdesc.modop       = MemAtomicOp_GCSSS1;
    accdesc.read         = TRUE;
    accdesc.write        = TRUE;

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescGPR

```

// CreateAccDescGPR()
// =====
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el          = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal = nontemporal;
    accdesc.read        = memop == MemOp_LOAD;
    accdesc.write       = memop == MemOp_STORE;
    accdesc.pan          = TRUE;
    accdesc.tagchecked   = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.TME;

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescGPTW

```

// CreateAccDescGPTW()
// =====

```

```

// Access descriptor for Granule Protection Table walks

AccessDescriptor CreateAccDescGPTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescIC

```

// CreateAccDescIC()
// =====
// Access descriptor for instruction cache operations

AccessDescriptor CreateAccDescIC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_IC);

    accdesc.cacheop      = cache.cacheop;
    accdesc.cachetype    = cache.cachetype;
    accdesc.opscope      = cache.opscope;

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescIFetch

```

// CreateAccDescIFetch()
// =====
// Access descriptor for instruction fetches

AccessDescriptor CreateAccDescIFetch()
    AccessDescriptor accdesc = NewAccDesc(AccessType_IFETCH);

    return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescLDAcqPC

```

// CreateAccDescLDAcqPC()
// =====
// Access descriptor for general purpose register loads with local ordering

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqpc        = TRUE;
    accdesc.read         = TRUE;
    accdesc.pan          = TRUE;
    accdesc.tagchecked   = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

## Library pseudocode for shared/functions/memory/CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====
// Access descriptor for tag memory loads/stores

AccessDescriptor CreateAccDescLDGSTG (MemOp memop)
    AccessDescriptor accdesc = NewAccDesc (AccessType_GPR);

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.tagaccess     = TRUE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLOR

```
// CreateAccDescLOR()
// =====
// Access descriptor for general purpose register loads/stores with limited ordering

AccessDescriptor CreateAccDescLOR (MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc (AccessType_GPR);

    accdesc.acqsc          = memop == MemOp_LOAD;
    accdesc.relsc          = memop == MemOp_STORE;
    accdesc.limitedordered = TRUE;
    accdesc.read           = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLS64

```
// CreateAccDescLS64()
// =====
// Access descriptor for accelerator-supporting memory accesses

AccessDescriptor CreateAccDescLS64 (MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc (AccessType_GPR);

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.ls64          = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescMOPS

```
// CreateAccDescMOPS()
// =====
// Access descriptor for data memory copy and set instructions

AccessDescriptor CreateAccDescMOPS(MemOp memop, boolean privileged, boolean
AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el           = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal = nontemporal;
    accdesc.read         = memop == MemOp_LOAD;
    accdesc.write        = memop == MemOp_STORE;
    accdesc.pan          = TRUE;
    accdesc.mops          = TRUE;
    accdesc.tagchecked   = TRUE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.
```

return accdesc;

## Library pseudocode for shared/functions/memory/CreateAccDescNV2

```
// CreateAccDescNV2()
// =====
// Access descriptor nested virtualization memory indirection loads/stores

AccessDescriptor CreateAccDescNV2(MemOp memop)
AccessDescriptor accdesc = NewAccDesc(AccessType_NV2);

    accdesc.el           = EL2;
    accdesc.ss            = SecurityStateAtEL(EL2);
    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write          = memop == MemOp_STORE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.
```

return accdesc;

## Library pseudocode for shared/functions/memory/CreateAccDescRCW

```
// CreateAccDescRCW()
// =====
// Access descriptor for atomic read-check-write memory accesses

AccessDescriptor CreateAccDescRCW(MemAtomicOp modop, boolean soft, boolean
                                release, boolean tagchecked)
AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = acquire;
    accdesc.relsc          = release;
    accdesc.rcw             = TRUE;
    accdesc.rcws            = soft;
    accdesc.atomicop        = TRUE;
    accdesc.modop          = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
```

```

accdesc.tagchecked      = tagchecked;
accdesc.transactional   = IsFeatureImplemented(FEAT_TME) && TSTATE.

return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescS1TTW

```

// CreateAccDescS1TTW()
// =====
// Access descriptor for stage 1 translation table walks

AccessDescriptor CreateAccDescS1TTW(boolean toplevel, VARange varange,
AccessDescriptor accdesc = NewAccDesc\(AccessType\_TTW\);

accdesc.el              = accdesc_in.el;
accdesc.ss              = accdesc_in.ss;
accdesc.read             = TRUE;
accdesc.toplevel         = toplevel;
accdesc.varange          = varange;
accdesc.mpam             = accdesc_in.mpam;

return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescS2TTW

```

// CreateAccDescS2TTW()
// =====
// Access descriptor for stage 2 translation table walks

AccessDescriptor CreateAccDescS2TTW(AccessDescriptor accdesc_in)
AccessDescriptor accdesc = NewAccDesc\(AccessType\_TTW\);

accdesc.el              = accdesc_in.el;
accdesc.ss              = accdesc_in.ss;
accdesc.read             = TRUE;
accdesc.mpam             = accdesc_in.mpam;

return accdesc;

```

### Library pseudocode for shared/functions/memory/CreateAccDescSME

```

// CreateAccDescSME()
// =====
// Access descriptor for SME loads/stores

AccessDescriptor CreateAccDescSME(MemOp memop, boolean nontemporal, boolean
                                tagchecked)
AccessDescriptor accdesc = NewAccDesc\(AccessType\_SME\);

accdesc.nontemporal       = nontemporal;
accdesc.read               = memop == MemOp\_LOAD;
accdesc.write              = memop == MemOp\_STORE;
accdesc.pan                = TRUE;
accdesc.contiguous          = contiguous;
accdesc.streamingsve        = TRUE;

```

```

        if boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR
            accdesc.tagchecked = FALSE;
        else
            accdesc.tagchecked = tagchecked;
        accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

        return accdesc;
    
```

### **Library pseudocode for shared/functions/memory/CreateAccDescSPE**

```

// CreateAccDescSPE()
// =====
// Access descriptor for memory accesses by Statistical Profiling unit

AccessDescriptor CreateAccDescSPE(SecurityState owning_ss, bits(2) owning_el,
                                 AccessDescriptor accdesc = NewAccDesc(AccessType_SPE));

    accdesc.el           = owning_el;
    accdesc.ss           = owning_ss;
    accdesc.write        = TRUE;
    accdesc.mpam         = GenMPAMatEL(AccessType_SPE, owning_el);

    return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescSTGMOPS**

```

// CreateAccDescSTGMOPS()
// =====
// Access descriptor for tag memory set instructions

AccessDescriptor CreateAccDescSTGMOPS(boolean privileged, boolean nontemporal,
                                       AccessDescriptor accdesc = NewAccDesc(AccessType_GPR));

    accdesc.el           = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal = nontemporal;
    accdesc.write        = TRUE;
    accdesc.pan          = TRUE;
    accdesc.mops          = TRUE;
    accdesc.tagaccess    = TRUE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescSVE**

```

// CreateAccDescSVE()
// =====
// Access descriptor for general SVE loads/stores

AccessDescriptor CreateAccDescSVE(MemOp memop, boolean nontemporal, boolean tagchecked)
                                 AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.nontemporal = nontemporal;
    accdesc.read         = memop == MemOp_LOAD;

```

```

accdesc.write          = memop == MemOp_STORE;
accdesc.pan            = TRUE;
accdesc.contiguous     = contiguous;
accdesc.streamingsve   = InStreamingMode();
if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
    "No tag checking of SIMD&FP loads and stores in Streaming SVE"
    accdesc.tagchecked = FALSE;
else
    accdesc.tagchecked = tagchecked;
accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescSVEFF**

```

// CreateAccDescSVEFF()
// =====
// Access descriptor for first-fault SVE loads

AccessDescriptor CreateAccDescSVEFF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read          = TRUE;
    accdesc.pan            = TRUE;
    accdesc.firstfault     = TRUE;
    accdesc.first           = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve   = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE"
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

    return accdesc;

```

### **Library pseudocode for shared/functions/memory/CreateAccDescSVENF**

```

// CreateAccDescSVENF()
// =====
// Access descriptor for non-fault SVE loads

AccessDescriptor CreateAccDescSVENF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read          = TRUE;
    accdesc.pan            = TRUE;
    accdesc.nonfault       = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve   = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE"
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.

```

```
    return accdesc;
```

### Library pseudocode for shared/functions/memory/CreateAccDescTRBE

```
// CreateAccDescTRBE()
// =====
// Access descriptor for memory accesses by Trace Buffer Unit

AccessDescriptor CreateAccDescTRBE(SecurityState owning_ss, bits(2) own
    AccessDescriptor accdesc = NewAccDesc\(AccessType\_TRBE\);

    accdesc.el          = owning_el;
    accdesc.ss          = owning_ss;
    accdesc.write       = TRUE;

    return accdesc;
```

### Library pseudocode for shared/functions/memory/CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// =====
// Access descriptor for translation table entry HW update

AccessDescriptor CreateAccDescTTEUpdate(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc\(AccessType\_TTW\);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.atomicop   = TRUE;
    accdesc.modop      = MemAtomicOp\_CAS;
    accdesc.read        = TRUE;
    accdesc.write       = TRUE;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

### Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
// DataMemoryBarrier()
// =====

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

### Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// =====

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean
```

## Library pseudocode for shared/functions/memory/DeviceType

```
// DeviceType
// =====
// Extended memory types for Device memory.

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGr}
```

## Library pseudocode for shared/functions/memory/EffectiveMTX

```
// EffectiveMTX()
// =====
// Returns the effective MTX in the AArch64 stage 1 translation regime for address.

bit EffectiveMTX(bits(64) address, boolean is_instr, bits(2) el)
    bit mtx;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if !IsFeatureImplemented(FEAT_MTE4) || is_instr then
        mtx = '0';
    else
        case regime of
            when EL1
                mtx = if address<55> == '1' then TCR_EL1.MTX1 else TCR_EL1.MTX0;
            when EL2
                if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                    mtx = if address<55> == '1' then TCR_EL2.MTX1 else TCR_EL2.MTX0;
                else
                    mtx = TCR_EL2.MTX;
            when EL3
                mtx = TCR_EL3.MTX;

    return mtx;
```

## Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for address.

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBII1 else TCR_EL1.TBII0;
            if IsFeatureImplemented(FEAT_PAAuth) then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
```

```

        tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
        if IsFeatureImplemented(FEAT_PAuth) then
            tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
        else
            tbi = TCR_EL2.TBI;
            if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL2.TBID;
    when EL3
        tbi = TCR_EL3.TBI;
        if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL3.TBID;

    return (if (tbi == '1' && (!IsFeatureImplemented(FEAT_PAuth) || !IsInstr)) then '1' else '0');

```

## Library pseudocode for shared/functions/memory/EffectiveTCMA

```

// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation
// path.

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;

```

## Library pseudocode for shared/functions/memory/ErrorState

```

// ErrorState
// =====
// The allowed error states that can be returned by memory and used by
// the system.

enumeration ErrorState {ErrorState_UC,                                // Uncontainable
                        ErrorState_UEU,                                // Unrecoverable state
                        ErrorState_UEO,                                // Restartable state
                        ErrorState_UER,                                // Recoverable state
                        ErrorState_CE,                                 // Corrected
                        ErrorState_Uncategorized,
                        ErrorState_IMPDEF};

```

## Library pseudocode for shared/functions/memory/Fault

```
// Fault
// =====
// Fault types.

enumeration Fault {Fault_None,
                     Fault_AccessFlag,
                     Fault_Alignment,
                     Fault_Background,
                     Fault_Domain,
                     Fault_Permission,
                     Fault_Translation,
                     Fault_AddressSize,
                     Fault_SyncExternal,
                     Fault_SyncExternalOnWalk,
                     Fault_SyncParity,
                     Fault_SyncParityOnWalk,
                     Fault_GPCFOnWalk,
                     Fault_GPCFOnOutput,
                     Fault_AsyncParity,
                     Fault_AsyncExternal,
                     Fault_TagCheck,
                     Fault_Debug,
                     Fault_TLBConflict,
                     Fault_BranchTarget,
                     Fault_HWUpdateAccessFlag,
                     Fault_Lockdown,
                     Fault_Exclusive,
                     Fault_ICacheMaint};
```

### **Library pseudocode for shared/functions/memory/FaultRecord**

```

// FaultRecord
// =====
// Fields that relate only to Faults.

type FaultRecord is (
    Fault          statuscode,           // Fault Status
    AccessDescriptor accessdesc,         // Details of the faulting access
    FullAddress    ipaddress,          // Intermediate physical address
    GPCFRecord    gpcf,                // Granule Protection Check Fault
    FullAddress    paddress,            // Physical address
    boolean           gpcfs2walk,        // GPC for a stage 2 translation
    boolean           s2fs1walk,          // Is on a Stage 1 translation
    boolean           write,               // TRUE for a write, FALSE for read
    boolean           sltagnotdata,      // TRUE for a fault due to tag mismatch
    boolean           tagaccess,          // TRUE for a fault due to NoTag
    integer           level,               // For translation, access flag
    bit               extflag,             // IMPLEMENTATION DEFINED syndrome
    boolean           secondstage,       // Is a Stage 2 abort
    boolean           assuredonly,       // Stage 2 Permission fault due to access
    boolean           toplevel,            // Stage 2 Permission fault due to permission
    boolean           overlay,             // Fault due to overlay permission
    boolean           dirtybit,            // Fault due to dirty state
    bits(4)           domain,              // Domain number, AArch32 only
    ErrorState     merrorstate,        // Incoming error state from memory
    boolean           maybe_false_match, // Watchpoint matches rounded result
    integer           watchpt_num,        // Matching watchpoint number
    bits(4)           debugmoe,            // Debug method of entry, from memory
)

```

## Library pseudocode for shared/functions/memory/FullAddress

```

// FullAddress
// =====
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate
// the full address type has 56 bits to allow interprocessing with AArch64
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED
// but never exceeds 56 bits.

type FullAddress is (
    PASpace  paspace,
    bits(56)   address
)

```

## Library pseudocode for shared/functions/memory/GPCF

```

// GPCF
// ====
// Possible Granule Protection Check Fault reasons

enumeration GPCF {
    GPCF_None,           // No fault
    GPCF_AddressSize,   // GPT address size fault
    GPCF_Walk,           // GPT walk fault
    GPCF_EABT,           // Synchronous External abort on GPT fetch
    GPCF_Fail            // Granule protection fault
};

```

## Library pseudocode for shared/functions/memory/GPCFRecord

```
// GPCFRecord
// =====
// Full details of a Granule Protection Check Fault

type GPCFRecord is (
    GPCF      gpf,
    integer level
)
```

## Library pseudocode for shared/functions/memory/Hint\_Prefetch

```
// Hint_Prefetch()
// =====
// Signals the memory system that memory accesses of type HINT to or from
// likely in the near future. The memory system may take some action to
// accesses when they do occur, such as pre-loading the specified address
// caches as indicated by the innermost cache level target (0=L1, 1=L2,
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch
// synchronous abort due to Alignment or Translation faults and the like
// software-visible state should be on caches and TLBs associated with
// accessible by reads, writes or execution, as defined in the translation
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be
// instruction fetch, therefore if all associated MMUs are disabled, the
// memory location that cannot be accessed by instruction fetches.
```

```
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, bool
```

## Library pseudocode for shared/functions/memory/Hint\_RangePrefetch

```
// Hint_RangePrefetch()
// =====
// Signals the memory system that data memory accesses from a specified
// range of addresses are likely to occur in the near future. The memory system
// respond by taking actions that are expected to speed up the memory access
// when they do occur, such as preloading the locations within the specified
// address ranges into one or more caches.

Hint_RangePrefetch(bits(64) address, integer length, integer stride,
                   integer count, integer reuse, bits(6) operation);
```

## Library pseudocode for shared/functions/memory/IsDataAccess

```
// IsDataAccess()
// =====
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccessType acctype)
    return !(acctype IN {AccessType IFETCH,
                         AccessType TTW,
                         AccessType DC,
```

```
AccessType_IC,  
AccessType_AT
```

### Library pseudocode for shared/functions/memory/IsSMEAccess

```
// IsSMEAccess()  
// =====  
// Return TRUE if access is of SME load/stores.  
  
boolean IsSMEAccess(AccessDescriptor accdesc)  
    return IsFeatureImplemented(FEAT_SME) && accdesc.acctype == AccessT
```

### Library pseudocode for shared/functions/memory/IsSVEAccess

```
// IsSVEAccess()  
// =====  
// Return TRUE if memory access is load/stores in an SVE mode.  
  
boolean IsSVEAccess(AccessDescriptor accdesc)  
    return IsFeatureImplemented(FEAT_SVE) && accdesc.acctype == AccessT
```

### Library pseudocode for shared/functions/memory/MBReqDomain

```
// MBReqDomain  
// =====  
// Memory barrier domain.  
  
enumeration MBReqDomain { MBReqDomain_Nonshareable, MBReqDomain_Inner  
                           MBReqDomain_OuterShareable, MBReqDomain_Full
```

### Library pseudocode for shared/functions/memory/MBReqTypes

```
// MBReqTypes  
// =====  
// Memory barrier read/write.  
  
enumeration MBReqTypes { MBReqTypes_Reads, MBReqTypes_Writes, MBReqT
```

### Library pseudocode for shared/functions/memory/MPAM

```
// MPAM Types  
// =====  
  
type PARTIDtype = bits(16);  
  
type PMGtype = bits(8);  
  
enumeration PARTIDspaceType {  
    PIDSpace_Secure,  
    PIDSpace_Root,  
    PIDSpace_Realm,
```

```

    PIDSpace_NonSecure
};

type MPAMinfo is (
    PARTIDspaceType mpam_sp,
    PARTIDtype partid,
    PMGtype pmg
)

```

## Library pseudocode for shared/functions/memory/MemAtomicOp

```
// MemAtomicOp
// =====
// Atomic data processing instruction types.

enumeration MemAtomicOp {
    MemAtomicOp_GCSSS1,
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS
};

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};

enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoE,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_PoPA,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};

enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};

enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_Any, // Applicable only for DC *SW / IC IALLU* i
               // match entries from any PA Space
};
```

```

        CPAS_RealmNonSecure,      // Applicable only for DC *SW / IC IALLU* i
                                // match entries from Realm or Non-Secure P
        CPAS_Realm,
        CPAS_Root,
        CPAS_SecureNonSecure,    // Applicable only for DC *SW / IC IALLU* i
                                // match entries from Secure or Non-Secure
        CPAS_Secure
    };
```

### **Library pseudocode for shared/functions/memory/MemAttrHints**

```

// MemAttrHints
// =====
// Attributes and hints for Normal memory.

type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

### **Library pseudocode for shared/functions/memory/MemOp**

```

// MemOp
// =====
// Memory access instruction types.

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

### **Library pseudocode for shared/functions/memory/MemType**

```

// MemType
// =====
// Basic memory types.

enumeration MemType {MemType_Normal, MemType_Device};
```

### **Library pseudocode for shared/functions/memory/Memory**

```

// Memory Tag type
// =====

enumeration MemTagType {
    MemTag_Untagged,
    MemTag_AllocationTagged,
    MemTag_CanonicallyTagged
};
```

### **Library pseudocode for shared/functions/memory/MemoryAttributes**

```

// MemoryAttributes
// =====
// Memory attributes descriptor

type MemoryAttributes is (
    MemType          memtype,
    DeviceType       device,           // For Device memory types
    MemAttrHints    inner,            // Inner hints and attributes
    MemAttrHints    outer,            // Outer hints and attributes
    Shareability   shareability,     // Shareability attribute
    MemTagType      tags,             // MTE tag type for this memory.
    boolean            notagaccess,     // Allocation Tag access permission
    bit                xs                // XS attribute
)

```

### Library pseudocode for shared/functions/memory/NewAccDesc

```

// NewAccDesc()
// =====
// Create a new AccessDescriptor with initialised fields

AccessDescriptor NewAccDesc(AccessType acctype)
    AccessDescriptor accdesc;

    accdesc.acctype          = acctype;
    accdesc.el               = PSTATE.EL;
    accdesc.ss               = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc            = FALSE;
    accdesc.acqpc            = FALSE;
    accdesc.relsc            = FALSE;
    accdesc.limitedordered   = FALSE;
    accdesc.exclusive        = FALSE;
    accdesc.rcw               = FALSE;
    accdesc.rcws              = FALSE;
    accdesc.atomicop          = FALSE;
    accdesc.nontemporal      = FALSE;
    accdesc.read              = FALSE;
    accdesc.write             = FALSE;
    accdesc.pan               = FALSE;
    accdesc.nonfault          = FALSE;
    accdesc.firstfault        = FALSE;
    accdesc.first              = FALSE;
    accdesc.contiguous         = FALSE;
    accdesc.streamingsve     = FALSE;
    accdesc.ls64               = FALSE;
    accdesc.mops              = FALSE;
    accdesc.a32lsmd            = FALSE;
    accdesc.tagchecked        = FALSE;
    accdesc.tagaccess          = FALSE;
    accdesc.transactional     = FALSE;
    accdesc.mpam               = GenMPAMcurEL(acctype);
    accdesc.ispair             = FALSE;
    accdesc.highestaddressfirst = FALSE;

    return accdesc;

```

### Library pseudocode for shared/functions/memory/PASpace

```

// PASpace
// ======
// Physical address spaces

enumeration PASpace {
    PAS_NonSecure,
    PAS_Secure,
    PAS_Root,
    PAS_Realm
};

```

### Library pseudocode for shared/functions/memory/Permissions

```

// Permissions
// =====
// Access Control bits in translation table descriptors

type Permissions is (
    bits(2) ap_table,      // Stage 1 hierarchical access permissions
    bit    xn_table,       // Stage 1 hierarchical execute-never for single
    bit    pxn_table,      // Stage 1 hierarchical privileged execute-never
    bit    uxn_table,      // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,            // Stage 1 access permissions
    bit    xn,              // Stage 1 execute-never for single EL regimes
    bit    uxn,             // Stage 1 unprivileged execute-never
    bit    pxn,             // Stage 1 privileged execute-never
    bits(4) ppi,           // Stage 1 privileged indirect permissions
    bits(4) upi,           // Stage 1 unprivileged indirect permissions
    bit    ndirty,          // Stage 1 dirty state for indirect permissions
    bits(4) s2pi,           // Stage 2 indirect permissions
    bit    s2dirty,          // Stage 2 dirty state
    bits(4) po_index,       // Stage 1 overlay permissions index
    bits(4) s2po_index,     // Stage 2 overlay permissions index
    bits(2) s2ap,           // Stage 2 access permissions
    bit    s2tag_na,         // Stage 2 tag access
    bit    s2xnx,            // Stage 2 extended execute-never
    bit    s2xn,             // Stage 2 execute-never
)

```

### Library pseudocode for shared/functions/memory/PhysMemRead

```

// PhysMemRead()
// =====
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, AccessDescriptor accdesc);

```

### Library pseudocode for shared/functions/memory/PhysMemRetStatus

```

// PhysMemRetStatus
// =====

```

```

// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus is (
    Fault      statuscode,      // Fault Status
    bit          extflag,        // IMPLEMENTATION DEFINED syndrome for E
    ErrorState merrorstate,    // Optional error state returned on a p
    bits(64)     store64bstatus // Status of 64B store
)

```

### Library pseudocode for shared/functions/memory/PhysMemWrite

```

// PhysMemWrite()
// =====
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus includes
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, Acc
                           bits(8*size) value);

```

### Library pseudocode for shared/functions/memory/PrefetchHint

```

// PrefetchHint
// =====
// Prefetch hint types.

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC}

```

### Library pseudocode for shared/functions/memory/S1AccessControls

```

// S1AccessControls
// =====
// Effective access controls defined by stage 1 translation

type S1AccessControls is (
    bit r,                      // Stage 1 base read permission
    bit w,                      // Stage 1 base write permission
    bit x,                      // Stage 1 base execute permission
    bit gcs,                    // Stage 1 GCS permission
    boolean overlay,            // Stage 1 overlay feature enabled
    bit or,                     // Stage 1 overlay read permission
    bit ow,                     // Stage 1 overlay write permission
    bit ox,                     // Stage 1 overlay execute permission
    bit wxn                     // Stage 1 write permission implies execute-
)

```

### Library pseudocode for shared/functions/memory/S2AccessControls

```

// S2AccessControls
// =====
// Effective access controls defined by stage 2 translation

type S2AccessControls is (

```

```

    bit r,                      // Stage 2 read permission.
    bit w,                      // Stage 2 write permission.
    bit x,                      // Stage 2 execute permission.
    bit r_rcw,                  // Stage 2 Read perms for RCW instruction.
    bit w_rcw,                  // Stage 2 Write perms for RCW instruction.
    bit r_mmu,                  // Stage 2 Read perms for TTW data.
    bit w_mmu,                  // Stage 2 Write perms for TTW data.
    bit toplevel0,              // IPA as top level table for TTBR0_EL1.
    bit toplevel1,              // IPA as top level table for TTBR1_EL1.
    boolean overlay,             // Overlay enable
    bit or,                     // Stage 2 overlay read permission.
    bit ow,                     // Stage 2 overlay write permission.
    bit ox,                     // Stage 2 overlay execute permission.
    bit or_rcw,                 // Stage 2 overlay Read perms for RCW instru
    bit ow_rcw,                 // Stage 2 overlay Write perms for RCW instru
    bit or_mmu,                 // Stage 2 overlay Read perms for TTW data.
    bit ow_mmu,                 // Stage 2 overlay Write perms for TTW data.
)

```

### **Library pseudocode for shared/functions/memory/Shareability**

```

// Shareability
// =====

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};

```

### **Library pseudocode for shared/functions/memory/ SpeculativeStoreBypassBarrierToPA**

```

// SpeculativeStoreBypassBarrierToPA()
// =====

SpeculativeStoreBypassBarrierToPA();

```

### **Library pseudocode for shared/functions/memory/ SpeculativeStoreBypassBarrierToVA**

```

// SpeculativeStoreBypassBarrierToVA()
// =====

SpeculativeStoreBypassBarrierToVA();

```

### **Library pseudocode for shared/functions/memory/Tag**

```

constant integer LOG2_TAG_GRANULE = 4;
constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;

```

## Library pseudocode for shared/functions/memory/VARange

```
// VARange
// =====
// Virtual address ranges

enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

## Library pseudocode for shared/functions/mpam/AltPARTIDspace

```
// AltPARTIDspace()
// =====
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

PARTIDspaceType AltPARTIDspace(bits(2) el, SecurityState security,
                                PARTIDspaceType primaryPIdSpace)
    case security of
        when SS_NonSecure
            assert el != EL3;
            return primaryPIdSpace;
        when SS_Secure
            assert el != EL3;
            if primaryPIdSpace == PIdSpace_NonSecure then
                return primaryPIdSpace;
            return AltPIdSecure(el, primaryPIdSpace);
        when SS_Root
            assert el == EL3;
            if MPAM3_EL3.ALTSP_EL3 == '1' then
                if MPAM3_EL3.RT_ALTSP_NS == '1' then
                    return PIdSpace_NonSecure;
                else
                    return PIdSpace_Secure;
            else
                return primaryPIdSpace;
        when SS_Realm
            assert el != EL3;
            return AltPIdRealm(el, primaryPIdSpace);
    otherwise
        Unreachable();
```

## Library pseudocode for shared/functions/mpam/AltPIdRealm

```
// AltPIdRealm()
// =====
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPIdRealm(bits(2) el, PARTIDspaceType primaryPIdSpace,
                            PARTIDspaceType PIdSpace = primaryPIdSpace);
```

```

case el of
    when EL0
        if ELIsInHost(EL0) then
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
            elseif !UsePrimarySpaceEL10() then
                PIdSpace = PIdSpace_NonSecure;
        when EL1
            if !UsePrimarySpaceEL10() then
                PIdSpace = PIdSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
        otherwise
            Unreachable();
return PIdSpace;

```

### Library pseudocode for shared/functions/mpam/AltPIdSecure

```

// AltPIdSecure()
// =====
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPIdSecure(bits(2) el, PARTIDspaceType primaryPIdSpace
    PARTIDspaceType PIdSpace = primaryPIdSpace;
    boolean el2en = EL2Enabled();
    case el of
        when EL0
            if el2en then
                if ELIsInHost(EL0) then
                    if !UsePrimarySpaceEL2() then
                        PIdSpace = PIdSpace_NonSecure;
                    elseif !UsePrimarySpaceEL10() then
                        PIdSpace = PIdSpace_NonSecure;
                elseif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC ==
                    PIdSpace = PIdSpace_NonSecure;
        when EL1
            if el2en then
                if !UsePrimarySpaceEL10() then
                    PIdSpace = PIdSpace_NonSecure;
                elseif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC ==
                    PIdSpace = PIdSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
        otherwise
            Unreachable();
return PIdSpace;

```

### Library pseudocode for shared/functions/mpam/DefaultMPAMinfo

```

// DefaultMPAMinfo()
// =====
// Returns default MPAM info. The partidspace argument sets
// the PARTID space of the default MPAM information returned.

```

```

MPAMinfo DefaultMPAMinfo(PARTIDspaceType partidspace)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_sp = partidspace;
    DefaultInfo.partid = DefaultPARTID;
    DefaultInfo.pmg = DefaultPMG;
    return DefaultInfo;

```

### Library pseudocode for shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

### Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype DefaultPMG = 0<7:0>;
```

### Library pseudocode for shared/functions/mpam/GenMPAMatEL

```

// GenMPAMatEL()
// =====
// Returns MPAMinfo for the specified EL.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMatEL(AccessType acctype, bits(2) el)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = SecurityStateAtEL(el);
    boolean InD = FALSE;
    boolean InSM = FALSE;
    PARTIDspaceType pspace = PARTIDspaceFromSS(security);
    if pspace == PIdSpace_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccessType_NV2 then EL2 else el;
        validEL = TRUE;
    case acctype of
        when AccessType_IFETCH, AccessType_IC
            InD = TRUE;
        when AccessType_SME
            InSM = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                     boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1" label pr
        when AccessType_ASIMD
            InSM = (IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' ||
                     (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                      boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1" label pr
        when AccessType_SVE
            InSM = (IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' ||
                     (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                      boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1" label pr

```

```

        otherwise
            // Other access types are DATA accesses
            InD = FALSE;
        if !validEL then
            return DefaultMPAMinfo(pspace);
        elseif IsFeatureImplemented(FEAT_RME) && MPAMIDR_EL1.HAS_ALTSP == '1'
            // Substitute alternative PARTID space if selected
            pspace = AltPARTIDspace(mpamEL, security, pspace);
        if IsFeatureImplemented(FEAT_MPAMv0p1) && MPAMIDR_EL1.HAS_FORCE_NS
            if MPAM3_EL3.FORCE_NS == '1' && security == SS_Secure then
                pspace = PIdSpace_NonSecure;
            if ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p1)) && MPAMIDR_EL1.HAS_SDEFLT == '1') then
                if MPAM3_EL3.SDEFLT == '1' && security == SS_Secure then
                    return DefaultMPAMinfo(pspace);
            if !MPAMisEnabled() then
                return DefaultMPAMinfo(pspace);
        else
            return genMPAM(mpamEL, InD, InSM, pspace);
    
```

## Library pseudocode for shared/functions/mpam/GenMPAMcurEL

```

// GenMPAMcurEL()
// =====
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccessType acctype)
    return GenMPAMatEL(acctype, PSTATE.EL);
    
```

## Library pseudocode for shared/functions/mpam/MAP\_vPARTID

```

// MAP_vPARTID()
// =====
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt      = UInt(vpartid);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);
    
```

```

// Check for valid mapping entry.
if MPAMVPMV_EL2<virt> == '1' then
    // vpartid has a valid mapping so access the map.
    ret = mapvpmw(virt);
    err = FALSE;

    // Is the default virtual PARTID valid?
    elseif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
else
    ret = DefaultPARTID;
    err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
return (ret, err);

```

### **Library pseudocode for shared/functions/mpam/MPAMisEnabled**

```

// MPAMisEnabled()
// =====
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
el = HighestEL();
case el of
    when EL3 return MPAM3_EL3.MPAMEN == '1';
    when EL2 return MPAM2_EL2.MPAMEN == '1';
    when EL1 return MPAM1_EL1.MPAMEN == '1';

```

### **Library pseudocode for shared/functions/mpam/MPAMisVirtual**

```

// MPAMisVirtual()
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
        ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
          (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
         (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1'))));

```

### **Library pseudocode for shared/functions/mpam/PARTIDspaceFromSS**

```

// PARTIDspaceFromSS()
// =====
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
    case security of
        when SS\_NonSecure
            return PIdSpace\_NonSecure;
        when SS\_Root
            return PIdSpace\_Root;
        when SS\_Realm
            return PIdSpace\_Realm;
        when SS\_Secure
            return PIdSpace\_Secure;
        otherwise
            Unreachable();

```

## Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL10

```

// UsePrimarySpaceEL10()
// =====
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

boolean UsePrimarySpaceEL10()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || !EL2Enabled() || MPAM2_EL2.ALTSP_HFC ==

```

## Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL2

```

// UsePrimarySpaceEL2()
// =====
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

boolean UsePrimarySpaceEL2()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';

```

## Library pseudocode for shared/functions/mpam/genMPAM

```

// genMPAM()
// =====
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I field
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// If InSM is TRUE returns MPAM information using PARTID_D and PMG_D field
// of MPAMSM_EL1 register.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, boolean InSM, PARTIDspaceType

```

```

MPAMinfo returninfo;
PARTIDtype partidel;
boolean perr;
// gtplk is guest OS application locked by the EL2 hypervisor to
// only use EL1 the virtual machine's PARTIDs.
boolean gtplk = (el == EL0 && EL2Enabled() &&
                  MPAMHCR_EL2.GSTAPP_PLK == '1' &&
                  HCR_EL2.TGE == '0');
bits(2) eff_el = if gtplk then EL1 else el;
(partidel, perr) = genPARTID(eff_el, InD, InSM);
PMGtype groupel = genPMG(eff_el, InD, InSM, perr);
returninfo.mpam_sp = pspace;
returninfo.partid = partidel;
returninfo.pmg = groupel;
return returninfo;

```

### Library pseudocode for shared/functions/mpam/genPARTID

```

// genPARTID()
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMEl_ELx.PARTID_I and
// otherwise from MPAMEl_ELx.PARTID_D.
// If InSM is TRUE then PARTID is from MPAMSM_EL1.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD, boolean InSM)
    PARTIDtype partidel = getMPAM_PARTID(el, InD, InSM);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);

```

### Library pseudocode for shared/functions/mpam/genPMG

```

// genPMG()
// =====
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should
// be called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean InSM, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD, InSM);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;

```

### Library pseudocode for shared/functions/mpam/getMPAM\_PARTID

```

// getMPAM_PARTID()
// =====
// Returns a PARTID from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD, boolean InSM)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InSM then
        partid = MPAMSM_EL1.PARTID_D;
        return partid;

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    else
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_D;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else
            when '01' partid = MPAM1_EL1.PARTID_D;
            when '00' partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;

```

## Library pseudocode for shared/functions/mpam/getMPAM\_PMG

```

// getMPAM_PMG()
// =====
// Returns a PMG from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD, boolean InSM)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InSM then
        pmg = MPAMSM_EL1.PMG_D;
        return pmg;

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros;
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    else

```

```

        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_D;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros;
            when '01' pmg = MPAM1_EL1.PMG_D;
            when '00' pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
        return pmg;
    
```

## Library pseudocode for shared/functions/mpam/mapvpmw

```

// mapvpmw()
// ======
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;

```

## Library pseudocode for shared/functions/predictionrestrict/ASID

```

// ASID []
// =====
// Effective ASID.

bits(16) ASID[]
    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) && HCR_EL2.<E2H, TGE> == '1'
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;

    elseif !ELUsingAArch32\(EL1\) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;

    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else

```

```

        if TTBCR.A1 == '1' then
            return ZeroExtend(TTBR1.ASID, 16);
        else
            return ZeroExtend(TTBR0.ASID, 16);
    
```

### **Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt**

```

// ExecutionCntxt
// =====
// Context information for prediction restriction operation.

type ExecutionCntxt is (
    boolean           is_vmid_valid, // is vmid valid for current context
    boolean           all_vmid,      // should the operation be applied
    bits(16)          vmid,         // if all_vmid = FALSE, vmid to which
    boolean           is_asid_valid, // is asid valid for current context
    boolean           all_asid,     // should the operation be applied
    bits(16)          asid,          // if all_asid = FALSE, ASID to which
    bits(2)           target_el,    // target EL at which operation is
    SecurityState   security,      // security level
    RestrictType    restriction   // type of restriction operation
)

```

### **Library pseudocode for shared/functions/predictionrestrict/ RESTRICT\_PREDICTIONS**

```

// RESTRICT_PREDICTIONS()
// =====
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;

```

### **Library pseudocode for shared/functions/predictionrestrict/RestrictType**

```

// RestrictType
// =====
// Type of restriction on speculation.

enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch,
    RestrictType_Other           // Any other trained speculation method
};

```

### **Library pseudocode for shared/functions/predictionrestrict/ TargetSecurityState**

```

// TargetSecurityState()
// =====
// Decode the target security state for the prediction context.

```

```

SecurityState TargetSecurityState(bit NS, bit NSE)
    curr_ss = SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elseif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;
    elseif IsFeatureImplemented(FEAT_RME) then
        if curr_ss == SS_Root then
            case NSE:NS of
                when '00' return SS_Secure;
                when '01' return SS_NonSecure;
                when '11' return SS_Realm;
                when '10' return SS_Root;
        elseif curr_ss == SS_Realm then
            return SS_Realm;
    Unreachable();

```

## Library pseudocode for shared/functions/registers/BranchTo

```

// BranchTo()
// =====
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has
// In AArch64 state the address might include a tag in the top eight bits

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE);
        if (IsFeatureImplemented(FEAT_BRBE) &&
            branch_type IN {BranchType_DIR, BranchType_INDIR,
                            BranchType_DIRCALL, BranchType_INDCALL,
                            BranchType_RET}) then
            BRBEBRANCH(branch_type, branch_conditional, target_vaddress);
        boolean branch_taken = TRUE;

        if IsFeatureImplemented(FEAT_SPE) then
            SPEBranch(target, branch_type, branch_conditional, branch_type);

        _PC = target_vaddress;
    return;

```

## Library pseudocode for shared/functions/registers/BranchToAddr

```

// BranchToAddr()
// =====
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);

```

```

        if N == 32 then
            assert UsingAArch32\(\);
            _PC = ZeroExtend(target, 64);
        else
            assert N == 64 && !UsingAArch32\(\);
            _PC = target<63:0>;
        return;
    
```

### **Library pseudocode for shared/functions/registers/BranchType**

```

// BranchType
// =====
// Information associated with a change in control flow.

enumeration BranchType {
    BranchType_DIRCALL,           // Direct Branch with link
    BranchType_INDCALL,          // Indirect Branch with link
    BranchType_ERET,             // Exception return (indirect)
    BranchType_DBGEXIT,          // Exit from Debug state
    BranchType_RET,               // Indirect branch with function return hint
    BranchType_DIR,               // Direct branch
    BranchType_INDIR,             // Indirect branch
    BranchType_EXCEPTION,         // Exception entry
    BranchType_TMFFAIL,           // Transaction failure
    BranchType_RESET,              // Reset
    BranchType_UNKNOWN};          // Other

```

### **Library pseudocode for shared/functions/registers/Hint\_Branch**

```

// Hint_Branch()
// =====
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration
// the next instruction.

Hint_Branch(BranchType hint);

```

### **Library pseudocode for shared/functions/registers/NextInstrAddr**

```

// NextInstrAddr()
// =====
// Return address of the sequentially next instruction.

bits(N) NextInstrAddr(integer N);

```

### **Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters**

```

// ResetExternalDebugRegisters()
// =====
// Reset the External Debug registers in the Core power domain.

ResetExternalDebugRegisters(boolean cold_reset);

```

## Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

## Library pseudocode for shared/functions/registers/\_PC

```
bits(64) _PC;
```

## Library pseudocode for shared/functions/registers/\_R

```
// _R[] - the general-purpose register file
// =====

array bits(64) _R[0..30];
```

## Library pseudocode for shared/functions/sysregisters/SPSR\_ELx

```
// SPSR_ELx[] - non-assignment form
// =====

bits(64) SPSR_ELx[]
    bits(64) result;
    case PSTATE.EL of
        when EL1          result = SPSR_EL1<63:0>;
        when EL2          result = SPSR_EL2<63:0>;
        when EL3          result = SPSR_EL3<63:0>;
        otherwise        Unreachable();
    return result;

// SPSR_ELx[] - assignment form
// =====

SPSR_ELx[] = bits(64) value
    case PSTATE.EL of
        when EL1          SPSR_EL1<63:0> = value<63:0>;
        when EL2          SPSR_EL2<63:0> = value<63:0>;
        when EL3          SPSR_EL3<63:0> = value<63:0>;
        otherwise        Unreachable();
    return;
```

## Library pseudocode for shared/functions/sysregisters/SPSR\_curr

```
// SPSR_curr[] - non-assignment form
// =====

bits(32) SPSR_curr[]
```

```

bits(32) result;
case PSTATE.M of
    when M32_FIQ      result = SPSR_fiq<31:0>;
    when M32 IRQ     result = SPSR_irq<31:0>;
    when M32_Svc     result = SPSR_svc<31:0>;
    when M32_Monitor result = SPSR_mon<31:0>;
    when M32_Abort   result = SPSR_abt<31:0>;
    when M32_Hyp    result = SPSR_hyp<31:0>;
    when M32_Undef   result = SPSR_und<31:0>;
    otherwise        Unreachable\(\);
return result;

// SPSR_curr[] - assignment form
// =====

SPSR_curr[] = bits(32) value
case PSTATE.M of
    when M32_FIQ      SPSR_fiq<31:0> = value<31:0>;
    when M32 IRQ     SPSR_irq<31:0> = value<31:0>;
    when M32_Svc     SPSR_svc<31:0> = value<31:0>;
    when M32_Monitor SPSR_mon<31:0> = value<31:0>;
    when M32_Abort   SPSR_abt<31:0> = value<31:0>;
    when M32_Hyp    SPSR_hyp<31:0> = value<31:0>;
    when M32_Undef   SPSR_und<31:0> = value<31:0>;
    otherwise        Unreachable\(\);
return;

```

### Library pseudocode for shared/functions/system/AArch64.ChkFeat

```

// AArch64.ChkFeat()
// =====
// Indicates the status of some features

bits(64) AArch64.ChkFeat(bits(64) feat_select)
    bits(64) feat_en = Zeros(64);
    feat_en[0] = if IsFeatureImplemented(FEAT_GCS) && GCSEnabled(PSTATE)
    return feat_select AND NOT(feat_en);

```

### Library pseudocode for shared/functions/system/ AddressNotInNaturallyAlignedBlock

```

// AddressNotInNaturallyAlignedBlock()
// =====
// The 'address' is not in a naturally aligned block if it doesn't meet
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set
// implemented or EDHCSR.FnP is being set to 0b0 (as appropriate).
// * Is no larger than the smallest implemented translation granule if
// (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or
// accesses that triggered the watchpoint.

boolean AddressNotInNaturallyAlignedBlock(bits(64) address);

```

### Library pseudocode for shared/functions/system/BranchTargetCheck

```

// BranchTargetCheck()
// =====
// This function is executed checks if the current instruction is a valid
// taken into, or inside, a guarded page. It is executed on every cycle.
// instruction has been decoded and the values of InGuardedPage and BTy
// determined for the current instruction.

BranchTargetCheck()
    assert IsFeatureImplemented(FEAT_BT) && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTyCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTY != '00' && !BTyCompatible && !Has
        bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTY defaults to 00 for instructions that do not explicitly
    if !(branch_instr || bti_instr) then
        BTyNext = '00';

```

## Library pseudocode for shared/functions/system/ClearEventRegister

```

// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;

```

## Library pseudocode for shared/functions/system/ConditionHolds

```

// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');
        when '001' result = (PSTATE.C == '1');
        when '010' result = (PSTATE.N == '1');
        when '011' result = (PSTATE.V == '1');
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');
        when '101' result = (PSTATE.N == PSTATE.V);
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');
        when '111' result = TRUE;

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

```

```
        return result;
```

### Library pseudocode for shared/functions/system/ ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()  
// =====  
  
ConsumptionOfSpeculativeDataBarrier();
```

### Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()  
// =====  
  
InstrSet CurrentInstrSet()  
    InstrSet result;  
    if UsingAArch32() then  
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32  
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state m  
    else  
        result = InstrSet_A64;  
    return result;
```

### Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()  
// =====  
  
PrivilegeLevel CurrentPL()  
    return PLofEL(PSTATE.EL);
```

### Library pseudocode for shared/functions/system/CurrentSecurityState

```
// CurrentSecurityState()  
// =====  
// Returns the effective security state at the exception level based off  
  
SecurityState CurrentSecurityState()  
    return SecurityStateAtEL(PSTATE.EL);
```

### Library pseudocode for shared/functions/system/DSBAlias

```
// DSBAlias  
// =====  
// Aliases of DSB.  
  
enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

## Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

## Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// =====
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_curr[].NS == '1' || IsSe...
```

## Library pseudocode for shared/functions/system/EL3SDDUndef

```
// EL3SDDUndef()
// =====
// Returns TRUE if in Debug state and EDSCR.SDD is set.

boolean EL3SDDUndef()
    return Halted() && EDSCR.SDD == '1';
```

## Library pseudocode for shared/functions/system/EL3SDDUndefPriority

```
// EL3SDDUndefPriority()
// =====
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by
// EL3 control register has priority over other traps.
// The IMPLEMENTATION_DEFINED priority may be different for each case.

boolean EL3SDDUndefPriority()
    return (Halted() && EDSCR.SDD == '1' &&
        boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD
```

## Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //     'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid
    //     and the current value of SCR.NS/SCR_EL3.NS.
    //     'EL'      is the Exception level decoded from 'mode'.
```

```

bits(2) el;
boolean valid = !BadMode(mode); // Check for modes that are not valid
bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();

case mode of
    when M32_Monitor
        el = EL3;
    when M32_Hyp
        el = EL2;
    when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
        // If EL3 is implemented and using AArch32, then these modes
        // state, and EL1 modes in Non-secure state. If EL3 is not
        // AArch64, then these modes are EL1 modes.
        el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then
    when M32_User
        el = EL0;
    otherwise
        valid = FALSE; // Passed an illegal mode value

if valid && el == EL2 && HaveEL(EL3) && SCR_curr[].NS == '0' then
    valid = FALSE; // EL2 only valid in Non-secure state

elsif valid && IsFeatureImplemented(FEAT_RME) && effective_nse_ns ==
    valid = FALSE; // Illegal Exception Return from EL2
                    // selects a reserved encoding

if !valid then el = bits(2) UNKNOWN;
return (valid, el);

```

## Library pseudocode for shared/functions/system/ELFromSPSR

```

// ELFromSPSR()
// =====

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current
//   'EL'      is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    bits(2) effective_nse_ns;
    if spsr<4> == '0' then // AArch64 state
        el = spsr<3:2>;
        effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
        if !HaveAArch64() then
            valid = FALSE; // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE; // Exception level not implemented
        elsif spsr<1> == '1' then
            valid = FALSE; // M[1] must be 0
        elsif el == EL0 && spsr<0> == '1' then
            valid = FALSE; // for EL0, M[0] must be 0
        elsif IsFeatureImplemented(FEAT_RME) && el != EL3 && effective_nse_ns ==
            valid = FALSE; // Only EL3 valid in Root state
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR.NS ==
            valid = FALSE; // Unless Secure EL2 is enabled, EL2 va

```

```

        else
            valid = TRUE;
        elseif HaveAArch32\(\) then // AArch32 state
            (valid, el) = ELFromM32(spsr<4:0>);
        else
            valid = FALSE;

        if !valid then el = bits(2) UNKNOWN;
        return (valid,el);
    
```

### Library pseudocode for shared/functions/system/**ELIsInHost**

```

// ELIsInHost()
// =====

boolean ELIsInHost(bits(2) el)
    if !IsFeatureImplemented(FEAT_VHE) || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return EL2Enabled() && HCR_EL2.E2H == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
        otherwise
            Unreachable();
    
```

### Library pseudocode for shared/functions/system/**ELStateUsingAArch32**

```

// ELStateUsingAArch32()
// =====

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;

```

### Library pseudocode for shared/functions/system/**ELStateUsingAArch32K**

```

// ELStateUsingAArch32K()
// =====

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    // 'known' is FALSE for EL0 if the current Exception level is not
    // using AArch64, since it cannot determine the state of the
    // 'aarch32' is TRUE if the specified Exception level is using AArch64
    if !HaveAArch32EL(el) then
        return (TRUE, FALSE); // Exception level is using AArch64
    elseif secure && el == EL2 then

```

```

        return (TRUE, FALSE); // Secure EL2 is using AArch64
    elseif !HaveAArch64() then
        return (TRUE, TRUE); // Highest Exception level, therefore a

// Remainder of function deals with the interprocessing cases when
// Exception level is using AArch64

boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;

aarch32_below_el3 = (HaveEL(EL3) &&
                     (!secure || !IsFeatureImplemented(FEAT_SEL2) ||
                      SCR_EL3.RW == '0'));
aarch32_at_el1 = (aarch32_below_el3 ||
                   (HaveEL(EL2) && (!secure ||
                     (IsFeatureImplemented(FEAT_SEL2) ||
                      !(IsFeatureImplemented(FEAT_VHE) && HCR_EL2.<E2H> ||
                      HCR_EL2.RW == '0'))));
if el == EL0 && !aarch32_at_el1 then // Only know if EL0 using AArch64
    if PSTATE.EL == EL0 then
        aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
    else
        known = FALSE; // EL0 state is UNKNOWN
else
    aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el != EL2);

if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);

```

### **Library pseudocode for shared/functions/system/ELUsingAArch32**

```

// ELUsingAArch32()
// =====

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());

```

### **Library pseudocode for shared/functions/system/ELUsingAArch32K**

```

// ELUsingAArch32K()
// =====

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());

```

### **Library pseudocode for shared/functions/system/EffectiveEA**

```

// EffectiveEA()
// =====
// Returns effective SCR_EL3.EA value

bit EffectiveEA()
    if Halted() && EDSCR.SDD == '0' then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3.EA else SCR.EA;

```

### **Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_NS**

```

// EffectiveSCR_EL3_NS()
// =====
// Return Effective SCR_EL3.NS value.

bit EffectiveSCR_EL3_NS()
    if !HaveSecureState() then
        return '1';
    elseif !HaveEL(EL3) then
        return '0';
    else
        return SCR_EL3.NS;

```

### **Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_NSE**

```

// EffectiveSCR_EL3_NSE()
// =====
// Return Effective SCR_EL3.NSE value.

bit EffectiveSCR_EL3_NSE()
    return if !IsFeatureImplemented(FEAT_RME) then '0' else SCR_EL3.NSE;

```

### **Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_RW**

```

// EffectiveSCR_EL3_RW()
// =====
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64() then
        return '0';
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && SCR_EL3.NS == '1' then
            return '1';
        if IsFeatureImplemented(FEAT_SEL2) && SCR_EL3.EEL2 == '1' && SCR_EL3.NS == '1'
            return '1';
    return SCR_EL3.RW;

```

### **Library pseudocode for shared/functions/system/EffectiveTGE**

```

// EffectiveTGE()
// =====
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0';           // Effective value of TGE is zero

```

### **Library pseudocode for shared/functions/system/EndOfInstruction**

```

// EndOfInstruction()
// =====
// Terminate processing of the current instruction.

EndOfInstruction();

```

### **Library pseudocode for shared/functions/system/EnterLowPowerState**

```

// EnterLowPowerState()
// =====
// PE enters a low-power state.

EnterLowPowerState();

```

### **Library pseudocode for shared/functions/system/EventRegister**

```
bits(1) EventRegister;
```

### **Library pseudocode for shared/functions/system/ ExceptionalOccurrenceTargetState**

```

// ExceptionalOccurrenceTargetState
// =====
// Enumeration to represent the target state of an Exceptional Occurrence
// The Exceptional Occurrence can be either Exception or Debug State etc.

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};

```

### **Library pseudocode for shared/functions/system/FIQPending**

```

// FIQPending()
// =====
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.

(boolean, boolean) FIQPending();

```

### **Library pseudocode for shared/functions/system/ GetAccumulatedFPEceptions**

```

// GetAccumulatedFPEceptions()
// =====
// Returns FP exceptions accumulated by the PE.

bits(8) GetAccumulatedFPEceptions();

```

### **Library pseudocode for shared/functions/system/GetLoadStoreType**

```

// GetLoadStoreType()
// =====
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.

bits(2) GetLoadStoreType();

```

### **Library pseudocode for shared/functions/system/GetPSRFromPSTATE**

```

// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
    if UsingAArch32\(\) && targetELState == AArch32\_NonDebugState then
        assert N == 32;
    else
        assert N == 64;

    bits(N) spsr = Zeros(N);
    spsr<31:28> = PSTATE.<N, Z, C, V>;
    if IsFeatureImplemented(FEAT_PAN) then spsr<22> = PSTATE.PAN;
    spsr<20> = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        if targetELState != AArch32\_NonDebugState then
            spsr<33> = PSTATE.PPEND;
        spsr<27> = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if IsFeatureImplemented(FEAT_SSBS) then spsr<23> = PSTATE.SSBS;
        if IsFeatureImplemented(FEAT_DIT) then
            if targetELState == AArch32\_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else // AArch64_NonD
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64\_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
        spsr<19:16> = PSTATE.GE;

```

```

        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>      = PSTATE.E;
        spsr<8:6>    = PSTATE.<A,I,F>;                                // No PSTATE.D
        spsr<5>      = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;                                // bit [4] is t
        spsr<4:0>    = PSTATE.M;
    else
        // AArch64 state
        if IsFeatureImplemented(FEAT_GCS) then spsr<34> = PSTATE.EXLOCK;
        if IsFeatureImplemented(FEAT_SEBEP) then spsr<33> = PSTATE.PPEN;
        if IsFeatureImplemented(FEAT_EBEP) then spsr<32> = PSTATE.PM;
        if IsFeatureImplemented(FEAT_MTE) then spsr<25> = PSTATE.TCO;
        if IsFeatureImplemented(FEAT_DIT) then spsr<24> = PSTATE.DIT;
        if IsFeatureImplemented(FEAT_UAO) then spsr<23> = PSTATE.UAO;
        spsr<21>      = PSTATE.SS;
        if IsFeatureImplemented(FEAT_NMI) then spsr<13> = PSTATE.ALLINT;
        if IsFeatureImplemented(FEAT_SSBS) then spsr<12> = PSTATE.SSBS;
        if IsFeatureImplemented(FEAT_BTI) then spsr<11:10> = PSTATE.BTY;
        spsr<9:6>    = PSTATE.<D,A,I,F>;
        spsr<4>       = PSTATE.nRW;
        spsr<3:2>    = PSTATE.EL;
        spsr<0>       = PSTATE.SP;
    return spsr;

```

## Library pseudocode for shared/functions/system/HasArchVersion

```

// HasArchVersion()
// =====
// Returns TRUE if the implemented architecture includes the extensions
// architecture version.

boolean HasArchVersion(boolean version)
    return version;

```

## Library pseudocode for shared/functions/system/HaveAArch32

```

// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return IsFeatureImplemented(FEAT_AA32EL0);

```

## Library pseudocode for shared/functions/system/HaveAArch32EL

```

// HaveAArch32EL()
// =====
// Return TRUE if Exception level 'el' supports AArch32 in this impleme

boolean HaveAArch32EL(bits(2) el)
    case el of
        when EL0 return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 return IsFeatureImplemented(FEAT_AA32EL3);

```

## Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return (IsFeatureImplemented(FEAT_AA64EL0) || IsFeatureImplemented(FEAT_AA64EL1) || IsFeatureImplemented(FEAT_AA64EL2))
```

## Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    case el of
        when EL1, EL0
            return TRUE; // EL1 and EL0 must exist
        when EL2
            return IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA64EL3)
        when EL3
            return IsFeatureImplemented(FEAT_AA64EL3) || IsFeatureImplemented(FEAT_AA64EL4)
        otherwise
            Unreachable();
```

## Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2);
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) || (secure == boolean IMPLEMENTATION_DEFINED "Secure-"))
```

## Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =====
// Return TRUE if FP16 extension is supported
```

```
boolean HaveFP16Ext()
    return IsFeatureImplemented(FEAT_FP16);
```

### Library pseudocode for shared/functions/system/HaveSecureState

```
// HaveSecureState()
// =====
// Return TRUE if Secure State is supported.

boolean HaveSecureState()
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    if IsFeatureImplemented(FEAT_RME) && !IsFeatureImplemented(FEAT_SEI)
        return FALSE;
    return TRUE;
```

### Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elseif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

### Library pseudocode for shared/functions/system/Hint\_CLRBHB

```
// Hint_CLRBHB()
// =====
// Provides a hint to clear the branch history for the current context.

Hint_CLRBHB();
```

### Library pseudocode for shared/functions/system/Hint\_DGH

```
// Hint_DGH()
// =====
// Provides a hint to close any gathering occurring within the micro-ar

Hint_DGH();
```

### Library pseudocode for shared/functions/system/Hint\_WFE

```
// Hint_WFE()
// =====
```

```

// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WFxType wfxtYPE)
    if IsEventRegisterSet() then
        ClearEventRegister();
    elseif IsFeatureImplemented(FEAT_WFXT) && LocalTimeoutEvent(locaLTIMEOUT)
        // No further operation if the local timeout has expired.
        EndOfInstruction();
    else
        bits(2) target_el;
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or
            if IsFeatureImplemented(FEAT_TWED) then
                sctlr      = SCTRLR_ELx[];
                trap       = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWFxTrap(EL1, wfxtYPE);
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost
            // Check for traps described by the Hypervisor.
            if IsFeatureImplemented(FEAT_TWED) then
                trap      = HCR_EL2.TWE == '1';
                target_el = EL2;
            else
                AArch64.CheckForWFxTrap(EL2, wfxtYPE);
        if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            if IsFeatureImplemented(FEAT_TWED) then
                trap      = SCR_EL3.TWE == '1';
                target_el = EL3;
            else
                AArch64.CheckForWFxTrap(EL3, wfxtYPE);
        if trap && PSTATE.EL != EL3 then
            // Determine if trap delay is enabled and delay amount
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired so trap WF
                AArch64.WFxTrap(wfxtYPE, target_el);
        else
            WaitForEvent(locaLTIMEOUT);

```

## Library pseudocode for shared/functions/system/Hint\_WFI

```

// Hint_WFI()
// =====
// Provides a hint indicating that the PE can enter a low-power state a
// remain there until a wakeup event occurs or, for WFIT, a local timec
// event is generated when the virtual timer value equals or exceeds th
// supplied threshold value.

Hint_WFI(integer localtimeout, WFxType wfxtYPE)
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then

```

```

FailTransaction(TMFailure_ERR, FALSE);

if (InterruptPending() || (IsFeatureImplemented(FEAT_WFXT) &&
    LocalTimeoutEvent(localttimeout))) then
    // No further operation if an interrupt is pending or the local
    EndOfInstruction();
else
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS.
        AArch64.CheckForWFxTrap(EL1, wfxtype);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, wfxtype);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, wfxtype);
    WaitForInterrupt(localttimeout);

```

### Library pseudocode for shared/functions/system/Hint\_Yield

```

// Hint_Yield()
// =====
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

Hint_Yield();

```

### Library pseudocode for shared/functions/system/IRQPending

```

// IRQPending()
// =====
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.

(boolean, boolean) IRQPending();

```

### Library pseudocode for shared/functions/system/IllegalExceptionReturn

```

// IllegalExceptionReturn()
// =====

boolean IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    //     * To an unimplemented Exception level.
    //     * To EL2 in Secure state, when SecureEL2 is not enabled.
    //     * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //     * To AArch64 state with SPSR.M[1]==1.
    //     * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

```

```

spsr_mode_is_aarch32 = (spsr<4> == '1');

// Check for illegal return:
//   * To EL1, EL2 or EL3 with register width specified in the SPSR
//     Execution state used in the Exception level being returned to
//     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset
//   * To EL0 using AArch64 state when EL1 is using AArch32 state and
//     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
//   * To AArch64 state from AArch32 state (should be caught by abc)
(known, target_el_is_aarch32) = ELUsingAArch32K(target);
assert known || (target == EL0 && !ELUsingAArch32(EL1));
if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return FALSE;

// Check for illegal return from AArch32 to AArch64
if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

// Check for illegal return to EL1 when HCR.TGE is set and when either
// * SecureEL2 is enabled.
// * SecureEL2 is not enabled and EL1 is in Non-secure state.
if EL2Enabled() && target == EL1 && HCR_EL2.TGE == '1' then
    if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;

if (IsFeatureImplemented(FEAT_GCS) && PSTATE.EXLOCK == '0' &&
    PSTATE.EL == target && GetCurrentEXLOCKEN()) then
    return TRUE;

return FALSE;

```

## Library pseudocode for shared/functions/system/InstrSet

```
// InstrSet
// =====

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

## **Library pseudocode for shared/functions/system/ InstructionSynchronizationBarrier**

```
// InstructionSynchronizationBarrier()  
// =====  
InstructionSynchronizationBarrier();
```

## Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending\(\);
    (fiq_pending, -) = FIQPending\(\);
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
IsPhysicalSErrorPending\(\));
```

```

if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
    boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending)
    boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending)
    boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending)
    HCR_EL2.VSE == '1'
    pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending
return pending_physical_interrupt || pending_virtual_interrupt;

```

### **Library pseudocode for shared/functions/system/IsASEInstruction**

```

// IsASEInstruction()
// =====
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction
boolean IsASEInstruction();

```

### **Library pseudocode for shared/functions/system/IsCurrentSecurityState**

```

// IsCurrentSecurityState()
// =====
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.
boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState() == ss;

```

### **Library pseudocode for shared/functions/system/IsEventRegisterSet**

```

// IsEventRegisterSet()
// =====
// Return TRUE if the Event Register of this PE is set, and FALSE if it is not
boolean IsEventRegisterSet()
    return EventRegister == '1';

```

### **Library pseudocode for shared/functions/system/IsHighestEL**

```

// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level
boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;

```

### **Library pseudocode for shared/functions/system/IsInHost**

```

// IsInHost()
// =====

```

```

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);

```

## Library pseudocode for shared/functions/system/IsSecure

```

// IsSecure()
// =====
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL\(EL3\) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elseif HaveEL\(EL3\) && UsingAArch32() && PSTATE.M == M32\_Monitor then
        return TRUE;
    return IsSecureBelowEL3();

```

## Library pseudocode for shared/functions/system/IsSecureBelowEL3

```

// IsSecureBelowEL3()
// =====
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL\(EL3\) then
        return SCR\_curr[].NS == '0';
    elseif HaveEL\(EL2\) && (!IsFeatureImplemented\(FEAT\_SEL2\) || !HaveAArc)
        // If Secure EL2 is not an architecture option then we must be
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementati

```

## Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```

// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL\(EL2\) && IsFeatureImplemented\(FEAT\_SEL2\) then
        if HaveEL\(EL3\) then
            if !ELUsingAArch32\(EL3\) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return SecureOnlyImplementation();
    else
        return FALSE;

```

## **Library pseudocode for shared/functions/system/LocalTimeoutEvent**

```
// LocalTimeoutEvent()
// =====
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localttimeout value.

boolean LocalTimeoutEvent(integer localttimeout);
```

## **Library pseudocode for shared/functions/system/Mode\_Bits**

```
constant bits(5) M32_User      = '10000';
constant bits(5) M32_FIQ       = '10001';
constant bits(5) M32_IRQ       = '10010';
constant bits(5) M32_Svc        = '10011';
constant bits(5) M32_Monitor   = '10110';
constant bits(5) M32_Abort      = '10111';
constant bits(5) M32_Hyp        = '11010';
constant bits(5) M32_Undef      = '11011';
constant bits(5) M32_System     = '11111';
```

## **Library pseudocode for shared/functions/system/ NonSecureOnlyImplementation**

```
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implem

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

## **Library pseudocode for shared/functions/system/PLOfEL**

```
// PLOfEL()
// =====

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if !HaveAArch64() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

## **Library pseudocode for shared/functions/system/PSTATE**

```
ProcState PSTATE;
```

## **Library pseudocode for shared/functions/system/PhysicalCountInt**

```

// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;

```

### Library pseudocode for shared/functions/system/PrivilegeLevel

```

// PrivilegeLevel
// =====
// Privilege Level abstraction.

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};

```

### Library pseudocode for shared/functions/system/ProcState

```

// ProcState
// =====
// Armv8 processor state bits.
// There is no significance to the field order.

type ProcState is (
    bits (1) N,           // Negative condition flag
    bits (1) Z,           // Zero condition flag
    bits (1) C,           // Carry condition flag
    bits (1) V,           // Overflow condition flag
    bits (1) D,           // Debug mask bit
    bits (1) A,           // SError interrupt mask bit
    bits (1) I,           // IRQ mask bit
    bits (1) F,           // FIQ mask bit
    bits (1) EXLOCK,      // Lock exception return state
    bits (1) PAN,          // Privileged Access Never Bit
    bits (1) UAO,          // User Access Override
    bits (1) DIT,          // Data Independent Timing
    bits (1) TCO,          // Tag Check Override
    bits (1) PM,           // PMU exception Mask
    bits (1) PPEND,         // synchronous PMU exception to be observed
    bits (2) BTYPE,         // Branch Type
    bits (1) ZA,           // Accumulation array enabled
    bits (1) SM,           // Streaming SVE mode enabled
    bits (1) ALLINT,        // Interrupt mask bit
    bits (1) SS,            // Software step bit
    bits (1) IL,            // Illegal Execution state bit
    bits (2) EL,            // Exception level
    bits (1) nRW,           // Execution state: 0=AArch64, 1=AArch32
    bits (1) SP,             // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,              // Cumulative saturation flag
    bits (4) GE,             // Greater than or Equal flags
    bits (1) SSBS,           // Speculative Store Bypass Safe
    bits (8) IT,              // If-then bits, RES0 in CPSR
    bits (1) J,               // J bit, RES0
    bits (1) T,               // T32 bit, RES0 in CPSR
    bits (1) E,               // Endianness bit
    bits (5) M                // Mode field
)

```

[AArch64 only] [v8.1] [v8.2] [v8.4] [v8.5, AArch32 only] [SME] [SME]

## Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether t
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable ILZEROIT) then retu
            else return it;

    // The IT bits are forced to zero when they are set to a reserved v
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or w
    // with the ITD bit set to 1, and the IT bits are describing a mult
    itd = if PSTATE.EL == EL2 then HSCTLR.ITYD else SCTLR.ITYD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>
        return '00000000';
    else
        return it;
```

## Library pseudocode for shared/functions/system/SCRTyp

```
type SCRTyp
```

## Library pseudocode for shared/functions/system/SCR\_curr

```
// SCR_curr[]
// =====

SCRTyp SCR_curr[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally m
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SCR, 64);
    else
        r = SCR_EL3;
    return r;
```

## Library pseudocode for shared/functions/system/SecureOnlyImplementation

```
// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this impleme
boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

## Library pseudocode for shared/functions/system/SecurityState

```
// SecurityState
// =====
// The Security state of an execution context

enumeration SecurityState {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```

## Library pseudocode for shared/functions/system/SecurityStateAtEL

```
// SecurityStateAtEL()
// =====
// Returns the effective security state at the exception level based off

SecurityState SecurityStateAtEL(bits(2) EL)
    if IsFeatureImplemented(FEAT_RME) then
        if EL == EL3 then return SS_Root;
        effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
        case effective_nse_ns of
            when '00' if IsFeatureImplemented(FEAT_SEL2) then return SS_Secure;
            when '01' return SS_NonSecure;
            when '11' return SS_Realm;
            otherwise      Unreachable();

    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
    elseif EL == EL3 then
        return SS_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
```

## Library pseudocode for shared/functions/system/SendEvent

```
// SendEvent()
// =====
// Signal an event to all PEs in a multiprocessor system to set their Event
// When a PE executes the SEV instruction, it causes this function to be called

SendEvent();
```

## Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// =====
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to

SendEventLocal()
    EventRegister = '1';
    return;
```

## **Library pseudocode for shared/functions/system/ SetAccumulatedFPEceptions**

```
// SetAccumulatedFPExceptions()
// =====
// Stores FP Exceptions accumulated by the PE.

SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

## **Library pseudocode for shared/functions/system/SetPSTATEFromPSR**

```

PSTATE.nRW = '0';
PSTATE.EL  = spsr<3:2>;
PSTATE.SP  = spsr<0>;
if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = spsr<
if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr<
if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = spsr<23>;
if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = spsr<24>;
if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = spsr<25>;
if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = spsr<

// If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the
// copied from SPSR.
if PSTATE.IL == '1' && PSTATE.nRW == '1' then
    if ConstrainUnpredictableBool\(Unpredictable\_ILZERO\) then spsr<

// State that is reinstated regardless of illegal exception return
PSTATE.<N,Z,C,V> = spsr<31:28>;
if IsFeatureImplemented(FEAT_PAN) then PSTATE.PAN = spsr<22>;
if PSTATE.nRW == '1' then                                // AArch32 state
    PSTATE.Q      = spsr<27>;
    PSTATE.IT     = RestoredITBits\(spsr\);
    ShouldAdvanceIT = FALSE;
    if IsFeatureImplemented(FEAT_DIT) then
        PSTATE.DIT = (if (Restarting\(\) || from_aarch64) then spsr<
    PSTATE.GE     = spsr<19:16>;
    PSTATE.E      = spsr<9>;
    PSTATE.<A,I,F> = spsr<8:6>;                      // No PSTATE.D in AAr
    PSTATE.T      = spsr<5>;                           // PSTATE.J is RES0
else
    PSTATE.PM     = spsr<32>;
    if IsFeatureImplemented(FEAT_NMI) then PSTATE.ALLINT = spsr<13>;
    PSTATE.<D,A,I,F> = spsr<9:6>;                  // No PSTATE.<Q,IT,GE>
return;

```

### **Library pseudocode for shared/functions/system/ShouldAdvanceHS**

```
boolean ShouldAdvanceHS;
```

### **Library pseudocode for shared/functions/system/ShouldAdvanceIT**

```
boolean ShouldAdvanceIT;
```

### **Library pseudocode for shared/functions/system/ShouldAdvanceSS**

```
boolean ShouldAdvanceSS;
```

### **Library pseudocode for shared/functions/system/ShouldSetPPEND**

```
boolean ShouldSetPPEND;
```

### **Library pseudocode for shared/functions/system/SmallestTranslationGranule**

```
// SmallestTranslationGranule()
// =====
// Smallest implemented translation granule.

integer SmallestTranslationGranule()
    if boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule" then
        if boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule" then
            if boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule" then
                Unreachable();
            else;
        end;
    end;
```

### **Library pseudocode for shared/functions/system/SpeculationBarrier**

```
// SpeculationBarrier()
// =====
SpeculationBarrier();
```

### **Library pseudocode for shared/functions/system/SyncCounterOverflowed**

```
boolean SyncCounterOverflowed;
```

### **Library pseudocode for shared/functions/system/SynchronizeContext**

```
// SynchronizeContext()
// =====
SynchronizeContext();
```

### **Library pseudocode for shared/functions/system/SynchronizeErrors**

```
// SynchronizeErrors()
// =====
// Implements the error synchronization event.

SynchronizeErrors();
```

### **Library pseudocode for shared/functions/system/ TakeUnmaskedPhysicalSErrorInterrupts**

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt.

TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

### **Library pseudocode for shared/functions/system/ TakeUnmaskedSErrorInterrupts**

```

// TakeUnmaskedSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt or unmasked virtual
// interrupt.

TakeUnmaskedSErrorInterrupts();

```

### **Library pseudocode for shared/functions/system/ThisInstr**

```

// ThisInstr()
// =====
bits(32) ThisInstr();

bits(32) ThisInstr();

```

### **Library pseudocode for shared/functions/system/ThisInstrLength**

```

// ThisInstrLength()
// =====
integer ThisInstrLength();

integer ThisInstrLength();

```

### **Library pseudocode for shared/functions/system/Unreachable**

```

// Unreachable()
// =====
Unreachable();

Unreachable();

```

### **Library pseudocode for shared/functions/system/UsingAArch32**

```

// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if
// not.

boolean UsingAArch32()
{
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;
}

```

### **Library pseudocode for shared/functions/system/ValidSecurityStateAtEL**

```

// ValidSecurityStateAtEL()
// =====
// Returns TRUE if the current settings and architecture choices for the
// implementation permit a valid Security state at the indicated EL.

boolean ValidSecurityStateAtEL(bits(2) el)
{
    if !HaveEL(el) then
        return FALSE;
}

```

```

        if el == EL3 then
            return TRUE;

        if IsFeatureImplemented(FEAT_RME) then
            bits(2) effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
            if effective_nse_ns == '10' then
                return FALSE;

        if el == EL2 then
            return EL2Enabled();

        return TRUE;
    
```

### **Library pseudocode for shared/functions/system/VirtualFIQPending**

```

// VirtualFIQPending()
// =====
// Returns TRUE if there is any pending virtual FIQ.

boolean VirtualFIQPending();
    
```

### **Library pseudocode for shared/functions/system/VirtualIRQPending**

```

// VirtualIRQPending()
// =====
// Returns TRUE if there is any pending virtual IRQ.

boolean VirtualIRQPending();
    
```

### **Library pseudocode for shared/functions/system/WFxType**

```

// WFxType
// =====
// WFx instruction types.

enumeration WFxType {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WF}
    
```

### **Library pseudocode for shared/functions/system/WaitForEvent**

```

// WaitForEvent()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timer
// It is IMPLEMENTATION_DEFINED whether restarting execution after the
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
    if !(IsEventRegisterSet()) ||
        (IsFeatureImplemented(FEAT_WFXT) && LocalTimeoutEvent(localtime
            EnterLowPowerState());
    return;

```

### **Library pseudocode for shared/functions/system/WaitForInterrupt**

```

// WaitForInterrupt()
// =====
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timer

WaitForInterrupt(integer localtimeout)
    if !(IsFeatureImplemented(FEAT_WFXT) && LocalTimeoutEvent(localtime
        EnterLowPowerState());
    return;

```

### **Library pseudocode for shared/functions/system/ WatchpointRelatedSyndrome**

```

// WatchpointRelatedSyndrome()
// =====
// Update common Watchpoint related fields.

bits(24) WatchpointRelatedSyndrome(FaultRecord fault, bits(64) vaddress)
    bits(24) syndrome = Zeros(24);

    if fault.maybe_false_match then
        syndrome<16> = '1';
    else
        syndrome<16> = bit IMPLEMENTATION_DEFINED "WPF value on TRUE WPF";
    end;

    if IsSVEAccess(fault.accessdesc) || IsSMEAcess(fault.accessdesc) then
        if HaltOnBreakpointOrWatchpoint() then
            if boolean IMPLEMENTATION_DEFINED "EDWAR is not valid on watchpoints"
                syndrome<10> = '1';
            end;
        else
            if boolean IMPLEMENTATION_DEFINED "FAR is not valid on watchpoints"
                syndrome<10> = '1';
            end;
        end;
    else
        if WatchpointFARNotPrecise(fault) then
            syndrome<15> = '1';
        end;
    end;
end;

```

```

// Watchpoint number is valid if FEAT_Debugv8p9 is implemented or
// if Feat_Debugv8p2 is implemented and below set of conditions are
// - Either FnV = 1 or FnP = 1.
// - If the address recorded in FAR is not within a naturally-aligned
// Otherwise , it is IMPLEMENTATION_DEFINED if watchpoint number is
if IsFeatureImplemented(FEAT_Debugv8p9) then
    syndrome<17> = '1';
    syndrome<23:18> = fault.watchpt_num<5:0>;
elseif IsFeatureImplemented(FEAT_Debugv8p2) then
    if syndrome<15> == '1' || syndrome<10> == '1' then
        syndrome<17> = '1';
    elseif AddressNotInNaturallyAlignedBlock(vaddress) then
        syndrome<17> = '1';
    elseif boolean IMPLEMENTATION_DEFINED "WPTV field is valid" then
        syndrome<17> = '1';
    if syndrome<17> == '1' then
        syndrome<23:18> = fault.watchpt_num<5:0>;
    else
        syndrome<23:18> = bits(6) UNKNOWN;

return syndrome;

```

### **Library pseudocode for shared/functions/unpredictable/ ConstrainUnpredictable**

```

// ConstrainUnpredictable()
// =====
// Return the appropriate Constraint result to control the caller's behavior
// The return value is IMPLEMENTATION_DEFINED within a permitted list for
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at
Constraint ConstrainUnpredictable(Unpredictable which);

```

### **Library pseudocode for shared/functions/unpredictable/ ConstrainUnpredictableBits**

```

// ConstrainUnpredictableBits()
// =====
// This is a variant of ConstrainUnpredictable for when the result can
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN
// value is always an allocated value; that is, one for which the behavior
// CONSTRAINED.
(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which);

```

### **Library pseudocode for shared/functions/unpredictable/ ConstrainUnpredictableBool**

```

// ConstrainUnpredictableBool()
// =====
// This is a variant of the ConstrainUnpredictable function where the result
// Constraint_TRUE or Constraint_FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);

```

### **Library pseudocode for shared/functions/unpredictable/ ConstrainUnpredictableInteger**

```

// ConstrainUnpredictableInteger()
// =====
// This is a variant of ConstrainUnpredictable for when the result can
// If the result is Constraint_UNKNOWN then the function also returns a
// value in the range low to high, inclusive.

(Constraint, integer) ConstrainUnpredictableInteger(integer low, integer
Unpredictable which)

```

### **Library pseudocode for shared/functions/unpredictable/ ConstrainUnpredictableProcedure**

```

// ConstrainUnpredictableProcedure()
// =====
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED
// The details of possible outcomes may not be present in the code and
// for each use with respect to the CONSTRAINED UNPREDICTABLE specification
// for the specific area.

ConstrainUnpredictableProcedure(Unpredictable which);

```

### **Library pseudocode for shared/functions/unpredictable/Constraint**

```

// Constraint
// =====
// List of Constrained Unpredictable behaviors.

enumeration Constraint      { // General
                                Constraint_NONE,           // Instruction
                                Constraint_UNDEF,          // no change
                                Constraint_UNDEFEL0,        // to its destination
                                Constraint_NOP,            // has UNKNOWN
                                Constraint_TRUE,           // Destination
                                Constraint_FALSE,          // has UNKNOWN
                                Constraint_DISABLED,        // Instruction
                                Constraint_UNCOND,         // Instruction
}

```

```

        Constraint_COND,           // Instructional
        Constraint_ADDITIONAL_DECODE, // Instructional
                                // with addition
        // Load-store
        Constraint_WBSUPPRESS,
        ConstraintFAULT,
        Constraint_LIMITED_ATOMICITY, // Accesses are
                                // single-copy
                                // above the boundary
        Constraint_NVNV1_00,
        Constraint_NVNV1_01,
        Constraint_NVNV1_11,
        Constraint_EL1TIMESTAMP,      // Constraint type
        Constraint_EL2TIMESTAMP,      // Constraint type
        Constraint_OSH,              // Constraint type
        Constraint_ISH,              // Constraint type
        Constraint_NSH,              // Constraint type

        Constraint_NC,               // Constraint type
        Constraint_WT,               // Constraint type
        Constraint_WB,               // Constraint type

        // IPA too large
        Constraint_FORCE, Constraint_FORCENOSLCHECK,
        // An unallocated System register value maps
        Constraint_MAPTOALLOCATED,
        // PMSCR_PCT reserved values select Virtual
        Constraint_PMSCR_PCT_VIRT
    };
}

```

## Library pseudocode for shared/functions/unpredictable/Unpredictable

```

// Unpredictable
// =====
// List of Constrained Unpredictable situations.

enumeration Unpredictable {
    // VMSR on MVFR
    Unpredictable_VMSR,
    // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LDPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Effect of SCTRLR_ELx.C on Tagged attribute
}

```

```
Unpredictable_S1CTAGGED,
// Reserved Stage 2 MemAttr value
Unpredictable_S2RESMEMATTR,
// Reserved TEX:C:B value
Unpredictable_RESTEXCB,
// Reserved PRRR value
Unpredictable_RESPRR,
// Reserved DACR field
Unpredictable_RESDACR,
// Reserved VTCR.S value
Unpredictable_RESVTCRS,
// Reserved TCR.TnSZ value
Unpredictable_RESTnSZ,
// Reserved SCTRL_ELx.TCF value
Unpredictable_RESTCF,
// Tag stored to Device memory
Unpredictable_DEVICETAGSTORE,
// Out-of-range TCR.TnSZ value
Unpredictable_OORTnSZ,

// IPA size exceeds PA size
Unpredictable_LARGEIPA,
// Syndrome for a known-passing conditional
Unpredictable_ESRCONDPASS,
// Illegal State exception: zero PSTATE.IT
Unpredictable_ILZEROIT,
// Illegal State exception: zero PSTATE.T
Unpredictable_ILZEROT,
// Debug: prioritization of Vector Catch
Unpredictable_BPVECTORCATCHPRI,
// Debug Vector Catch: match on 2nd halfword
Unpredictable_VCMATCHHALF,
// Debug Vector Catch: match on Data Abort
// or Prefetch abort
Unpredictable_VCMATCHDAPA,
// Debug watchpoints: nonzero MASK and non-contiguous BAS
Unpredictable_WPMASKANDBAS,
// Debug watchpoints: non-contiguous BAS
Unpredictable_WPBASECONTIGUOUS,
// Debug watchpoints: reserved MASK
Unpredictable_RESWPMASK,
// Debug watchpoints: nonzero MASKed bits of
Unpredictable_WPMASKEDBITS,
// Debug breakpoints and watchpoints: reserved type
Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
// Debug breakpoints and watchpoints: reserved type
Unpredictable_RESMDSELR,
// Debug breakpoints: not-context-aware break
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword
Unpredictable_BPMISMATCHHALF,
// Debug breakpoints: a breakpoint is linked
// programmed with linking enabled
Unpredictable_BLINKINGDISABLED,
```

```
// Debug breakpoints: reserved MASK  
Unpredictable_RESERVEDMASK,  
// Debug breakpoints: MASK is set for a Cont  
// breakpoint or when DBGBCR_EL1[n].BAS != '  
Unpredictable_BPMASK,  
// Debug breakpoints: nonzero MASKed bits of  
Unpredictable_BPMAKEDBITS,  
// Debug breakpoints: A linked breakpoint is  
// linked to an address matching breakpoint  
Unpredictable_BPLINKEDADDRMATCH,  
// Debug: restart to a misaligned AArch32 PC  
Unpredictable_RESTARTALIGNPC,  
// Debug: restart to a not-zero-extended AAr  
Unpredictable_RESTARTZEROUPPERPC,  
// Zero top 32 bits of X registers in AArch3  
Unpredictable_ZEROUPPER,  
// Zero top 32 bits of PC on illegal return  
// AArch32 state  
Unpredictable_ERETZEROUPPERPC,  
// Force address to be aligned when interwo  
// branch to A32 state  
Unpredictable_A32FORCEALIGNPC,  
// SMC disabled  
Unpredictable_SMD,  
// FF speculation  
Unpredictable_NONFAULT,  
// Zero top bits of Z registers in EL change  
Unpredictable_SVEZEROUPPER,  
// Load mem data in NF loads  
Unpredictable_SVELDNFDATA,  
// Write zeros in NF loads  
Unpredictable_SVELDNFZERO,  
// SP alignment fault when predicate is all  
Unpredictable_CHECKSPNONEACTIVE,  
// Zero top bits of ZA registers in EL change  
Unpredictable_SMEZEROUPPER,  
// Watchpoint match of last rounded up memor  
// 16 byte rounding  
Unpredictable_16BYTEROUNDEDUPACCESS,  
// Watchpoint match of first rounded down mem  
// 16 byte rounding  
Unpredictable_16BYTEROUNDEDOWNACCESS,  
// HCR_EL2.<NV,NV1> == '01'  
Unpredictable_NVNV1,  
// Reserved shareability encoding  
Unpredictable_Shareability,  
// Access Flag Update by HW  
Unpredictable_AFUPDATE,  
// Dirty Bit State Update by HW  
Unpredictable_DBUPDATE,  
// Consider SCTRLR_ELx[].IESB in Debug state  
Unpredictable_IESBInDebug,  
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/P  
Unpredictable_BADPMSFCR,  
// Zero saved BType value in SPSR_ELx/DPSR_E  
Unpredictable_ZEROBTYPE,  
// Timestamp constrained to virtual or physi  
Unpredictable_EL2TIMESTAMP,  
Unpredictable_EL1TIMESTAMP,  
// Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_E
```

```

        Unpredictable_RESERVEDNSxB,
        // WFET or WFIT instruction in Debug state
        Unpredictable_WFxTDEBUG,
        // Address does not support LS64 instruction
        Unpredictable_LS64UNSUPPORTED,
        // Misaligned exclusives, atomics, acquire/release
        // to region that is not Normal Cacheable WE
        Unpredictable_MISALIGNEDATOMIC,
        // 128-bit Atomic or 128-bit RCW{S} transfer
        Unpredictable_LSE128OVERLAP,
        // Clearing DCC/ITR sticky flags when instruction
        Unpredictable_CLEARERRITEZERO,
        // ALUEXCEPTIONRETURN when in user/system mode
        // A32 instructions
        Unpredictable_ALUEXCEPTIONRETURN,
        // Trap to register in debug state are ignored
        Unpredictable_IGNORETRAPINDEBUG,
        // Compare DBGBVR.RESS for BP/WP
        Unpredictable_DBGxVR_RESS,
        // Inaccessible event counter
        Unpredictable_PMUEVENTCOUNTER,
        // Reserved PMSCR.PCT behavior
        Unpredictable_PMSCR_PCT,
        // MDCR_EL2.HPMN or HDCR.HPMN is larger than
        // FEAT_HPMN0 is not implemented and HPMN is
        Unpredictable_CounterReservedForEL2,
        // Generate BRB_FILTRATE event on BRB inject
        Unpredictable_BRBFILTRATE,
        // Generate PMU_SNAPSHOT event in Debug state
        Unpredictable_PMUSNAPSHOTEVENT,
        // Reserved MDCR_EL3.EPMSSAD value
        Unpredictable_SETEPMSSAD,
        // Reserved PMECR_EL1.SSE value
        Unpredictable_RESPMSSE,
        // Enable for PMU exception and PMUIRQ
        Unpredictable_RESPMEE,
        // Operands for CPY*/SET* instructions overlap
        // use 0b1111 as a register specifier
        Unpredictable_MOPSOVERLAP31,
        // Store-only Tag checking on a failed Atomic
        Unpredictable_STOREONLYTAGCHECKEDCAS,
        // Reserved MDCR_EL3.ETBAD value
        Unpredictable_RES_ETBAD,
        // accessing DBGDSCRint via MRC in debug state
        Unpredictable_MRC_APSCR_TARGET,
        // Reserved PMEVTYPE<n>_EL0.TC value
        Unpredictable_RESTC
    };
}

```

### Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```

// AdvSIMDExpandImm()
// =====
bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
{
    bits(64) imm64;
    case cmode<3:1> of
        when '000'

```

```

        imm64 = Replicate(Zeros(24):imm8, 2);
when '001'
        imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
when '010'
        imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
when '011'
        imm64 = Replicate(imm8:Zeros(24), 2);
when '100'
        imm64 = Replicate(Zeros(8):imm8, 4);
when '101'
        imm64 = Replicate(imm8:Zeros(8), 4);
when '110'
        if cmode<0> == '0' then
            imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
        else
            imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
when '111'
        if cmode<0> == '0' && op == '0' then
            imm64 = Replicate(imm8, 8);
        if cmode<0> == '0' && op == '1' then
            imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
            imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
            imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
            imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
            imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
        if cmode<0> == '1' && op == '0' then
            imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5>;
            imm64 = Replicate(imm32, 2);
        if cmode<0> == '1' && op == '1' then
            if UsingAArch32() then ReservedEncoding();
            imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5>;
return imm64;

```

## Library pseudocode for shared/functions/vector/MatMulAdd

```

// MatMulAdd()
// =====
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned,
                  boolean op2_unsigned)
assert N == 128;

bits(N) result;
bits(32) sum;
integer prod;

for i = 0 to 1
    for j = 0 to 1
        sum = Elem[addend, 2*i + j, 32];
        for k = 0 to 7
            prod = (Int(Elem[op1, 8*i + k, 8], op1_unsigned) *
                     Int(Elem[op2, 8*j + k, 8], op2_unsigned));
            sum = sum + prod;
        Elem[result, 2*i + j, 32] = sum;

```

```
    return result;
```

### Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// =====

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

### Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// =====

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ
    return (result, sat);
```

### Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// =====

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1; saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1)); saturated = TRUE;
    else
        result = i; saturated = FALSE;
    return (result<N-1:0>, saturated);
```

### Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then // Operands <= 0x3FFFFFFF produce
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25..1]
```

```
// estimate is in the range 256 .. 511 representing [1.0 .. 2.0
increasedprecision = FALSE;
estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
// result is in the range 0x80000000 .. 0xff800000 representing
result = estimate<8:0> : Zeros(N-9);

return result;
```

## **Library pseudocode for shared/functions/vector/UnsignedRecipEstimate**

```
// UnsignedRecipEstimate()
// =====

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then // Operands <= 0xFFFFFFFF produce 0xFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0]
        // estimate is in the range 256 to 511 representing [1.0 .. 2.0]
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [0.5 .. 1.0]
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

## Library pseudocode for shared/functions/vector/UnsignedSatQ

```

// UnsignedSatQ()
// =====

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

```

## Library pseudocode for shared/trace/Common/DebugMemWrite

```

PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;

// Translate virtual address
AddressDescriptor addrdesc;
integer size = 1;
addrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

if IsFault(addrdesc) then
    return (memstatus, addrdesc);

memstatus = PhysMemWrite(addrdesc, 1, accdesc, data);

return (memstatus, addrdesc);

```

## Library pseudocode for shared/trace/Common/DebugWriteExternalAbort

```

// DebugWriteExternalAbort()
// =====
// Populate the syndrome register for an External abort caused by a call

DebugWriteExternalAbort(PhysMemRetStatus memstatus, AddressDescriptor addrdesc,
                      bits(64) start_vaddr)

boolean iswrite = TRUE;

boolean handle_asSError = FALSE;
boolean async_external_abort = FALSE;
bits(64) syndrome;
case addrdesc.fault.accessdesc.acctype of
    when AccessType SPE
        handle_asSError = boolean IMPLEMENTATION_DEFINED "SPE Sync"
        async_external_abort = boolean IMPLEMENTATION_DEFINED "SPE Sync"
        syndrome = PMBSR_EL1<63:0>;
    otherwise
        Unreachable();
    end

boolean ttw_abort;
ttw_abort = addrdesc.fault.statuscode IN {Fault SyncExternalOnWalk,
                                         Fault SyncParityOnWalk};

Fault statuscode = if ttw_abort then addrdesc.fault.statuscode else
bit extflag = if ttw_abort then addrdesc.fault.extflag else memstatus;
if (statuscode IN {Fault AsyncExternal, Fault AsyncParity} || handle_asSError)
    // ASYNC Fault -> SError or SYNC Fault handled as SError
    FaultRecord fault = NoFault();
    boolean parity = statuscode IN {Fault SyncParity, Fault AsyncParity,
                                    Fault SyncParityOnWalk};
    fault.statuscode = if parity then Fault AsyncParity else Fault SyncParity;
    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memstatus.merrorstate;
    fault.extflag = extflag;
    fault.accessdesc.acctype = addrdesc.fault.accessdesc.acctype;
    PendSErrorInterrupt(fault);
else
    // SYNC Fault, not handled by SError
    // Generate Buffer Management Event
    // EA bit
    syndrome<18> = '1';
end

```

```

// DL bit for SPE
if addrdesc.fault.accessdesc.acctype == AccessType_SPE && (async
    (start_vaddr != addrdesc.vaddress)) then
    syndrome<19> = '1';

// Do not change following values if previous Buffer Management
// has not been handled.
// S bit
if IsZero(syndrome<17>) then
    syndrome<17> = '1';

// EC bits
bits(6) ec;
if (IsFeatureImplemented(FEAT_RME) && addrdesc.fault.gpcf.gpf != GPCF_Fail) then
    ec = '011110';
else
    ec = if addrdesc.fault.secondstage then '100101' else '';
syndrome<31:26> = ec;

// MSS bits
if async_external_abort then
    syndrome<15:0> = Zeros(10) : '010001';
else
    syndrome<15:0> = Zeros(10) : EncodeLDFSC(statuscode, acctype);

case addrdesc.fault.accessdesc.acctype of
when AccessType_SPE
    PMBSR_EL1<63:0> = syndrome;
otherwise
    Unreachable();

```

## Library pseudocode for shared/trace/Common/DebugWriteFault

```

// DebugWriteFault()
// =====
// Populate the syndrome register for a Translation fault caused by a core

DebugWriteFault(bits(64) vaddress, FaultRecord fault)
    bits(64) syndrome;
    case fault.accessdesc.acctype of
        when AccessType_SPE
            syndrome = PMBSR_EL1<63:0>;
        otherwise
            Unreachable();

    // MSS
    syndrome<15:0> = Zeros(10) : EncodeLDFSC(fault.statuscode, fault.level);

    // MSS2
    syndrome<55:32> = Zeros(24);

    // EC bits
    bits(6) ec;
    if (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &
        fault.gpcf.gpf != GPCF_Fail) then
        ec = '011110';

```

```

        else
            ec = if fault.secondstage then '100101' else '100100';
            syndrome<31:26> = ec;

        // S bit
        syndrome<17> = '1';

        if fault.statuscode == Fault_Permission then
            // assuredonly bit
            syndrome<39> = if fault.assuredonly then '1' else '0';
            // overlay bit
            syndrome<38> = if fault.overlay then '1' else '0';
            // dirtybit
            syndrome<37> = if fault.dirtybit then '1' else '0';

        case fault.accessdesc.acctype of
            when AccessType_SPE
                PMBSR_EL1<63:0> = syndrome;
            otherwise
                Unreachable();
        endcase;

        // Buffer Write Pointer already points to the address that generated
        // Writing to memory never started so no data loss. DL is unchanged
    endfunction;

```

## Library pseudocode for shared/trace/Common/GetTimestamp

```

// GetTimestamp()
// =====
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp_Physical
            return PhysicalCountInt();
        when TimeStamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp_OffsetPhysical
            bits(64) phyoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2
                           else 0;
            return PhysicalCountInt() - phyoff;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
    endcase;

```

## Library pseudocode for shared/trace/Common/PhysicalOffsetIsValid

```

// PhysicalOffsetIsValid()
// =====
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
    if !HaveAArch64() then
        return FALSE;

```

```

        elseif !HaveEL(EL2) || !IsFeatureImplemented(FEAT_ECV) then
            return FALSE;
        elseif HaveEL(EL3) && SCR_EL3.NS == '1' && EffectiveSCR_EL3_RW() ==
            return FALSE;
        elseif HaveEL(EL3) && SCR_EL3.ECVen == '0' then
            return FALSE;
        elseif CNTHCTL_EL2.ECV == '0' then
            return FALSE;
        else
            return TRUE;
    
```

### Library pseudocode for shared/trace/TraceBranch/BranchNotTaken

```

// BranchNotTaken()
// =====
// Called when a branch is not taken.

BranchNotTaken(BranchType branchtype, boolean branch_conditional)
    boolean branchtaken = FALSE;
    if IsFeatureImplemented(FEAT_SPE) then
        SPEBranch(bits(64) UNKNOWN, branchtype, branch_conditional, bra
    return;

```

### Library pseudocode for shared/trace/TraceBuffer/ AllowExternalTraceBufferAccess

```

// AllowExternalTraceBufferAccess()
// =====
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess()
    return AllowExternalTraceBufferAccess(AccessState());

// AllowExternalTraceBufferAccess()
// =====
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed for the given Security state, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess(SecurityState access_state)
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    assert IsFeatureImplemented(FEAT_Debugv8p4); // Required when Trac

    bits(2) etbad = if HaveEL(EL3) then MDCR_EL3.ETBAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01', '10'} then
        Constraint c;
        (c, etbad) = ConstrainUnpredictableBits(Unpredictable RES ETBA
    assert c IN {Constraint DISABLED, Constraint UNKNOWN};
    if c == Constraint DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits m
    // not-reserved value

    case etbad of

```

```

when '00'
    SecurityState ss = (if IsFeatureImplemented(FEAT_RME) then
        return access_state == ss;
when '01'
    return access_state IN {SS_Root, SS_Realm};
when '10'
    return access_state IN {SS_Root, SS_Secure};
when '11'
    return TRUE;

```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferEnabled

```

// TraceBufferEnabled()
// =====
boolean TraceBufferEnabled()
    if !IsFeatureImplemented(FEAT_TRBE) || TRBLIMITR_EL1.E == '0' then
        return FALSE;
    if !SelfHostedTraceEnabled() then
        return FALSE;
    (-, el) = TraceBufferOwner();
    return !ELUsingAArch32(el);

```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferOwner

```

// TraceBufferOwner()
// =====
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
    assert IsFeatureImplemented(FEAT_TRBE) && SelfHostedTraceEnabled();

    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if IsFeatureImplemented(FEAT_RME) then
            state_bits = MDCR_EL3.<NSTBE, NSTB>;
            if (state_bits IN {'10x'}) ||
                (!IsFeatureImplemented(FEAT_SEL2) && state_bits IN {'0x000'}) ||
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable);
        else
            state_bits = '0' : MDCR_EL3.NSTB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
        else
            owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;
    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled())
        owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    return (owning_ss, owning_el);

```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferRunning

```
// TraceBufferRunning()
// =====
boolean TraceBufferRunning()
    return TraceBufferEnabled\(\) && TRBSR_EL1.S == '0';
```

## Library pseudocode for shared/trace/TraceInstrumentationAllowed/TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// =====
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

boolean TraceInstrumentationAllowed(SecurityState ss, bits(2) el)
    if !IsFeatureImplemented(FEAT_ITE) then return FALSE;
    if ELUsingAArch32(el) then return FALSE;

    if TraceAllowed(el) then
        bit ite_bit;
        case el of
            when EL3 ite_bit = '0';
            when EL2 ite_bit = TRCITECR_EL2.E2E;
            when EL1 ite_bit = TRCITECR_EL1.E1E;
            when EL0
                if EffectiveTGE() == '1' then
                    ite_bit = TRCITECR_EL2.E0HE;
                else
                    ite_bit = TRCITECR_EL1.E0E;

        if SelfHostedTraceEnabled() then
            return ite_bit == '1';
        else
            bit el_bit;
            bit ss_bit;
            case el of
                when EL0 el_bit = TRCITEEDCR.E0;
                when EL1 el_bit = TRCITEEDCR.E1;
                when EL2 el_bit = TRCITEEDCR.E2;
                when EL3 el_bit = TRCITEEDCR.E3;
            case ss of
                when SS\_Realm ss_bit = TRCITEEDCR.RL;
                when SS\_Secure ss_bit = TRCITEEDCR.S;
                when SS\_NonSecure ss_bit = TRCITEEDCR.NS;
                otherwise ss_bit = '1';

            boolean ed_allowed = ss_bit == '1' && el_bit == '1';

            if TRCCONFIGR.ITO == '1' then
                return ed_allowed;
            else
                return ed_allowed && ite_bit == '1';
        else
            return FALSE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =====
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
    return if ELUsingAArch32\(EL2\) then HTRFCR.E0HTRE else TRFCR_EL2.E0H
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

```
// EffectiveE0TRE()
// =====
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32\(EL1\) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

```
// EffectiveE1TRE()
// =====
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32\(\) then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

```
// EffectiveE2TRE()
// =====
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32\(\) then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

## Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    bit secure_trace_enable = '0';
    if !(HaveTraceExt\(\) && IsFeatureImplemented(FEAT_TRF)) then return
    if EDSCR.TFO == '0' then return TRUE;
    if IsFeatureImplemented(FEAT_RME) then
        secure_trace_enable = if IsFeatureImplemented(FEAT_SEL2) then M
        return ((secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebug\(EL2\)) && (MDCR_EL3.RLTE == '1' && !ExternalRealmNoninvasiveDebug\(EL3\))
    if HaveEL\(EL3\) then
        secure_trace_enable = if ELUsingAArch32\(EL3\) then SDCR.STE else
```

```

    else
        secure_trace_enable = if SecureOnlyImplementation() then '1' else
            if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled(el)
                return TRUE;
            return FALSE;

```

## Library pseudocode for shared/trace/selfhosted/TraceAllowed

```

// TraceAllowed()
// =====
// Returns TRUE if Self-hosted Trace is allowed in the given Exception

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never
        case ss of
            when SS_NonSecure
                trace_allowed = TRUE;
            when SS_Secure
                bit trace_bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';
            when SS_Realm
                trace_allowed = MDCR_EL3.RLTE == '1';
            when SS_Root
                trace_allowed = FALSE;

        // Tracing is prohibited if the trace buffer owning security state
        // current Security state or the owning Exception level is a local
        if IsFeatureImplemented(FEAT_TRBE) && TraceBufferEnabled() then
            (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;

        bit TRE_bit;
        case el of
            when EL3  TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else
            when EL2  TRE_bit = EffectiveE2TRE();
            when EL1  TRE_bit = EffectiveE1TRE();
            when EL0
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveE0HTRE();
                else
                    TRE_bit = EffectiveE0TRE();

            return trace_allowed && TRE_bit == '1';
        else
            return ExternalNoninvasiveDebugAllowed(el);

```

## Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// =====

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL) || !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled()) || TRFCR_EL2.CX == '1';
```

## Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

```
// TraceSynchronizationBarrier()
// =====
// Memory barrier instruction that preserves the relative order of memory
// registers due to trace operations and other memory accesses to the same
// address.
```

TraceSynchronizationBarrier();

## Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// =====

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    // Otherwise ConstrainUnpredictableBits removes this
                    assert IsFeatureImplemented(FEAT_ECV);
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;

            TS_el1 = TRFCR_EL1.TS;
            if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10')
                // Reserved value
                (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable);

            case TS_el1 of
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    assert IsFeatureImplemented(FEAT_ECV);
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;
```

```

        otherwise
            Unreachable();           // ConstrainUnpredictableBits re
    else
        return TimeStamp_CoreSight;

```

### Library pseudocode for shared/trace/system/IsTraceCorePowered

```

// IsTraceCorePowered()
// =====
// Returns TRUE if the Trace Core Power Domain is powered up
boolean IsTraceCorePowered();

```

### Library pseudocode for shared/translation/at

```

enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_Any,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};

```

### Library pseudocode for shared/translation/at/EncodePARAttrs

```

// EncodePARAttrs()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;

    if IsFeatureImplemented(FEAT_MTE) && memattrs.tags == MemTag_Allocated
        if IsFeatureImplemented(FEAT_MTE_PERM) && memattrs.notagaccess
            result<7:0> = '11100000';
        else
            result<7:0> = '11110000';
        return result;

    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        case memattrs.device of
            when DeviceType_nGnRnE result<3:0> = '0000';
            when DeviceType_nGnRE result<3:0> = '0100';
            when DeviceType_nGRE result<3:0> = '1000';
            when DeviceType_GRE result<3:0> = '1100';
            otherwise
                Unreachable();
        result<0> = NOT memattrs.xs;
    else
        if memattrs.xs == '0' then

```

```

        if (memattrs.outer.attrs == MemAttr_WT && memattrs.inner.attrs == MemAttr_NC)
            !memattrs.outer.transient && memattrs.outer.hints == MemAttr_NC)
                return '10100000';
        elseif memattrs.outer.attrs == MemAttr_NC && memattrs.inner.attrs == MemAttr_WT)
            !memattrs.outer.transient && memattrs.outer.hints == MemAttr_WT)
                return '01000000';

        if memattrs.outer.attrs == MemAttr_WT then
            result<7:6> = if memattrs.outer.transient then '00' else '11';
            result<5:4> = memattrs.outer.hints;
        elseif memattrs.outer.attrs == MemAttr_WB then
            result<7:6> = if memattrs.outer.transient then '01' else '10';
            result<5:4> = memattrs.outer.hints;
        else // MemAttr_NC
            result<7:4> = '0100';

        if memattrs.inner.attrs == MemAttr_WT then
            result<3:2> = if memattrs.inner.transient then '00' else '11';
            result<1:0> = memattrs.inner.hints;
        elseif memattrs.inner.attrs == MemAttr_WB then
            result<3:2> = if memattrs.inner.transient then '01' else '10';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
            result<3:0> = '0100';

    return result;

```

## Library pseudocode for shared/translation/at/PAREncodeShareability

```

// PAREncodeShareability()
// =====
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
         memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';

```

## Library pseudocode for shared/translation/at/ReportedPARAttrs

```

// ReportedPARAttrs()
// =====
// The value returned in this field can be the resulting attribute, as determined by
// implementation choices and any applicable configuration bits, instead of
// in the translation table descriptor.

bits(8) ReportedPARAttrs(bits(8) parattrs);

```

## Library pseudocode for shared/translation/at/ReportedPARShareability

```

// ReportedPARShareability()
// =====
// The value returned in SH field can be the resulting attribute, as determined by
// permitted implementation choices and any applicable configuration bits.
// the value that appears in the translation table descriptor.

bits(2) ReportedPARShareability(bits(2) sh);

```

### Library pseudocode for shared/translation/attrs/DecodeDevice

```

// DecodeDevice()
// =====
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00'    return DeviceType\_nGnRnE;
        when '01'    return DeviceType\_nGnRE;
        when '10'    return DeviceType\_nGRE;
        when '11'    return DeviceType\_GRE;

```

### Library pseudocode for shared/translation/attrs/DecodeLDFAttr

```

// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if      attr IN {'x0xx'} then ldfattr.attrs = MemAttr\_WT; // Write-through
    elseif attr == '0100' then ldfattr.attrs = MemAttr\_NC; // Non-cacheable
    elseif attr IN {'x1xx'} then ldfattr.attrs = MemAttr\_WB; // Write-back
    else                      Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr\_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint\_No; // No allocation hint
            when '01' ldfattr.hints = MemHint\_WA; // Write-allocate
            when '10' ldfattr.hints = MemHint\_RA; // Read-allocate
            when '11' ldfattr.hints = MemHint\_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some alignment requirements.
    if ldfattr.attrs != MemAttr\_NC && ldfattr.hints != MemHint\_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;

```

### Library pseudocode for shared/translation/attrs/DecodeSDFAttr

```

// DecodeSDFAttr()
// =====
// Decode memory attributes using SDF (Short Descriptor Format) mapping

```

```

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00'                                // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01'                                // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10'                                // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11'                                // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;

    sdfattr.transient = FALSE;

    return sdfattr;

```

### Library pseudocode for shared/translation/attrs/DecodeShareability

```

// DecodeShareability()
// =====
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability_OSH;
        when '11' return Shareability_ISH;
        when '00' return Shareability_NSH;
    otherwise
        case ConstrainUnpredictable(Unpredictable_Shareability) of
            when Constraint_OSH return Shareability_OSH;
            when Constraint_ISH return Shareability_ISH;
            when Constraint_NSH return Shareability_NSH;

```

### Library pseudocode for shared/translation/attrs/EffectiveShareability

```

// EffectiveShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
        (memattrs.inner.attrs == MemAttr_NC &&
         memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;

```

### Library pseudocode for shared/translation/attrs/NormalNCMemAttr

```

// NormalNCMemAttr()
// =====

```

```

// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;

    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype      = MemType_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;
    nc_memattrs.tags         = MemTag_Untagged;
    nc_memattrs.notagaccess  = FALSE;

    return nc_memattrs;

```

### Library pseudocode for shared/translation/attrs/ S1ConstrainUnpredictableRESMAIR

```

// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean slaarch64)
    case attr of
        when '0000xx01' return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
        when '10100000' return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
        when '11110000' return !(slaarch64 && IsFeatureImplemented(FEAT_XS));
        when 'xxxx0000' return TRUE;
        otherwise           return FALSE;

```

### Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

```

// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean
                                  S1TTWParams walkparams)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
        (-, attr) = ConstrainUnpredictableBits(Unpredictable RESMAIR, 8);

    MemoryAttributes memattrs;
    case attr of
        when '0000xxxx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device  = DecodeDevice(attr<3:2>);
            memattrs.xs      = if slaarch64 then NOT attr<0> else '1';
        when '01000000'
            assert slaarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs = MemAttr_NC;
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.xs       = '0';

```

```

when '10100000'
    assert slaarch64 && IsFeatureImplemented(FEAT_XS);
    memattrs.memtype = MemType Normal;
    memattrs.outer.attrs = MemAttr WT;
    memattrs.outer.hints = MemHint RA;
    memattrs.outer.transient = FALSE;
    memattrs.inner.attrs = MemAttr WT;
    memattrs.inner.hints = MemHint RA;
    memattrs.inner.transient = FALSE;
    memattrs.xs = '0';
when '11110000' // Tagged memory
    assert slaarch64 && IsFeatureImplemented(FEAT_MTE2);
    memattrs.memtype = MemType Normal;
    memattrs.outer.attrs = MemAttr WB;
    memattrs.outer.hints = MemHint RWA;
    memattrs.outer.transient = FALSE;
    memattrs.inner.attrs = MemAttr WB;
    memattrs.inner.hints = MemHint RWA;
    memattrs.inner.transient = FALSE;
    memattrs.xs = '0';
otherwise
    memattrs.memtype = MemType Normal;
    memattrs.outer = DecodeLDFAttr(attr<7:4>);
    memattrs.inner = DecodeLDFAttr(attr<3:0>);

    if (memattrs.inner.attrs == MemAttr WB &&
        memattrs.outer.attrs == MemAttr WB) then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    if slaarch64 && attr IN {'11110000'} then
        memattrs.tags = MemTag AllocationTagged;
    elseif slaarch64 && walkparams.mtx == '1' then
        memattrs.tags = MemTag CanonicallyTagged;
    else
        memattrs.tags = MemTag Untagged;

    memattrs.notagaccess = FALSE;

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;

```

## Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```

// S2CombineS1AttrHints()
// =====
// Determine resultant Normal memory cacheability and allocation hints
// combining stage 1 Normal memory attributes and stage 2 cacheability

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr NC || s2_attrhints.attrs == MemAttr NC
        attrhints.attrs = MemAttr NC;
    elseif s1_attrhints.attrs == MemAttr WT || s2_attrhints.attrs == MemAttr WT
        attrhints.attrs = MemAttr WT;
    else
        attrhints.attrs = MemAttr NC;

```

```

        attrhints.attrs = MemAttr_WT;
else
    attrhints.attrs = MemAttr_WB;

// Stage 2 does not assign any allocation hints
// Instead, they are inherited from stage 1
if attrhints.attrs != MemAttr_NC then
    attrhints.hints      = s1_attrhints.hints;
    attrhints.transient = s1_attrhints.transient;

return attrhints;

```

### Library pseudocode for shared/translation/attrs/S2CombineS1Device

```

// S2CombineS1Device()
// =====
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE
        return DeviceType_nGnRnE;
    elseif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE
        return DeviceType_nGnRE;
    elseif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;

```

### Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```

// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_memattrs, MemoryAttributes s2_memattrs, boolean s2aarch64)
    MemoryAttributes memattrs;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype ==
        memattrs.memtype = MemType_Device;
        memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elseif s1_memattrs.memtype == MemType_Device then // S2 Normal, S2 Device
        memattrs = s1_memattrs;
    elseif s2_memattrs.memtype == MemType_Device then // S2 Device, S2 Normal
        memattrs = s2_memattrs;
    else // S2 Normal, S2 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);

    if !IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = FALSE;
    else
        memattrs.notagaccess = (s2_memattrs.notagaccess &&

```

```

        s1_memattrs.tags == MemTag_AllocationTag
memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                         s2_memattrs.shareability)

if (memattrs.memtype == MemType_Normal &&
    memattrs.inner.attrs == MemAttr_WB &&
    memattrs.outer.attrs == MemAttr_WB) then
    memattrs.xs = '0';
elsif s2aarch64 then
    memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
else
    memattrs.xs = s1_memattrs.xs;

memattrs.shareability = EffectiveShareability(memattrs);
return memattrs;

```

### Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```

// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                    Shareability s2_shareability)

if (s1_shareability == Shareability_OSH ||
    s2_shareability == Shareability_OSH) then
    return Shareability_OSH;
elsif (s1_shareability == Shareability_ISH ||
      s2_shareability == Shareability_ISH) then
    return Shareability_ISH;
else
    return Shareability_NSH;

```

### Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```

// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
MemAttrHints s2attr;

case attr of
    when '01' s2attr.attrs = MemAttr_NC; // Non-cacheable
    when '10' s2attr.attrs = MemAttr_WT; // Write-through
    when '11' s2attr.attrs = MemAttr_WB; // Write-back
    otherwise // Constrained unpredictable
        case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
            when Constraint_NC s2attr.attrs = MemAttr_NC;
            when Constraint_WT s2attr.attrs = MemAttr_WT;
            when Constraint_WB s2attr.attrs = MemAttr_WB;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination
    s2attr.hints      = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

```

```
    return s2attr;
```

## Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh, boolean s2aarch64)
{
    MemoryAttributes memattrs;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DecodeDevice(attr<1:0>);
        when '0100' // Normal, Inner+Outer WB cacheable NoTagAccess memory
            if s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM) then
                memattrs.memtype      = MemType_Normal;
                memattrs.outer         = S2DecodeCacheability('11'); //
                memattrs.inner         = S2DecodeCacheability('11'); //
            else
                memattrs.memtype      = MemType_Normal;
                memattrs.outer         = S2DecodeCacheability(attr<3:2>);
                memattrs.inner         = S2DecodeCacheability(attr<1:0>);
        otherwise // Normal memory
            memattrs.memtype      = MemType_Normal;
            memattrs.outer         = S2DecodeCacheability(attr<3:2>);
            memattrs.inner         = S2DecodeCacheability(attr<1:0>);

    memattrs.shareability = DecodeShareability(sh);

    if s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = attr == '0100';
    else
        memattrs.notagaccess = FALSE;

    return memattrs;
}
```

## Library pseudocode for shared/translation/attrs/S2MemTagType

```
// S2MemTagType()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

MemTagType S2MemTagType(MemoryAttributes s2_memattrs, MemTagType s1_tagtype)
{
    if !IsFeatureImplemented(FEAT_MTE2) then
        return MemTag_Untagged;

    if ((s1_tagtype == MemTag_AllocationTagged) &&
        (s2_memattrs.memtype == MemType_Normal) &&
        (s2_memattrs.inner.attrs == MemAttr_WB) &&
        (s2_memattrs.inner.hints == MemHint_RWA) &&
        (!s2_memattrs.inner.transient) &&
        (s2_memattrs.outer.attrs == MemAttr_WB)) &&
```

```

        (s2_memattrs.outer.hints == MemHint_RWA) &&
        (!s2_memattrs.outer.transient)) then
            return MemTag_AllocationTagged;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag_AllocationTagged then
        return s1_tagtype;

    return MemTag_Untagged;

```

### Library pseudocode for shared/translation/attrs/WalkMemAttrs

```

// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
    walkmemattrs.outer        = DecodeSDFAttr(orgn);
    walkmemattrs.tags         = MemTag_Untagged;
    if (walkmemattrs.inner.attrs == MemAttr_WB &&
        walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;

```

### Library pseudocode for shared/translation/faults/AlignmentFault

```

// AlignmentFault()
// =====
// Return a fault record indicating an Alignment fault not due to memory
// for a specific access

FaultRecord AlignmentFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode  = Fault_Alignment;
    fault.accessdesc  = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk  = FALSE;
    fault.write       = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf       = GPCNoFault();

    return fault;

```

### Library pseudocode for shared/translation/faults/ExclusiveFault

```

// ExclusiveFault()
// =====
// Return a fault record indicating an Exclusive fault for a specific access

FaultRecord ExclusiveFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode = Fault_Exclusive;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;

```

## Library pseudocode for shared/translation/faults/NoFault

```

// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

FaultRecord NoFault()
    FaultRecord fault;

    fault.statuscode = Fault_None;
    fault.accessdesc = AccessDescriptor UNKNOWN;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;
    fault.dirtybit = FALSE;
    fault.overlay = FALSE;
    fault.toplevel = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess = FALSE;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();

    return fault;

// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred for a specific access

FaultRecord NoFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode = Fault_None;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk = FALSE;
    fault.dirtybit = FALSE;
    fault.overlay = FALSE;
    fault.toplevel = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess = FALSE;

```

```

fault.write      = !accdesc.read && accdesc.write;
fault.gpcfs2walk = FALSE;
fault.gpcf      = GPCNoFault\(\);

return fault;

```

### Library pseudocode for shared/translation/gpc/AbovePPS

```

// AbovePPS()
// =====
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.

boolean AbovePPS(bits(56) address)
    constant integer pps = DecodePPS\(\);
    if pps >= 56 then
        return FALSE;

    return !IsZero(address<55:pps>);

```

### Library pseudocode for shared/translation/gpc/DecodeGPTBlock

```

// DecodeGPTBlock()
// =====
// Validate and decode a GPT Block descriptor

(GPCF, GPTEntry) DecodeGPTBlock(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT\_Block;
    GPTEntry result;

    if !IsZero(gpt_entry<63:8>) then
        return (GPCF\_Walk, GPTEntry UNKNOWN);

    if !GPIValid(gpt_entry<7:4>) then
        return (GPCF\_Walk, GPTEntry UNKNOWN);

    result.gpi    = gpt_entry<7:4>;
    result.level = 0;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS\_4KB   result.size = GPTRange\_4KB;
        when PGS\_16KB  result.size = GPTRange\_16KB;
        when PGS\_64KB  result.size = GPTRange\_64KB;
        otherwise Unreachable\(\);
    result.contig_size = GPTL0Size\(\);

    return (GPCF\_None, result);

```

### Library pseudocode for shared/translation/gpc/DecodeGPTContiguous

```

// DecodeGPTContiguous()
// =====
// Validate and decode a GPT Contiguous descriptor

```

```
(GPCF, GPTEntry) DecodeGPTContiguous(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Contig;
    GPTEntry result;

    if !IsZero(gpt_entry<63:10>) then
        return (GPCF_Walk, result);

    result.gpi = gpt_entry<7:4>;
    if !GPIValid(result.gpi) then
        return (GPCF_Walk, result);

    case pgs of
        when PGS_4KB result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    case gpt_entry<9:8> of
        when '01' result.contig_size = GPTRange_2MB;
        when '10' result.contig_size = GPTRange_32MB;
        when '11' result.contig_size = GPTRange_512MB;
        otherwise return (GPCF_Walk, GPTEntry UNKNOWN);

    result.level = 1;

    return (GPCF_None, result);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTGranules

```
// DecodeGPTGranules()
// =====
// Validate and decode a GPT Granules descriptor

(GPCF, GPTEntry) DecodeGPTGranules(PGSe pgs, integer index, bits(64) gp
GPTEntry result;

for i = 0 to 15
    if !GPIValid(gpt_entry<i*4 +:4>) then
        return (GPCF_Walk, result);

result.gpi = gpt_entry<index*4 +:4>;

case pgs of
    when PGS_4KB result.size = GPTRange_4KB;
    when PGS_16KB result.size = GPTRange_16KB;
    when PGS_64KB result.size = GPTRange_64KB;
    otherwise Unreachable();

result.contig_size = result.size; // No contiguity
result.level = 1;

return (GPCF_None, result);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTTable

```

// DecodeGPTTable()
// =====
// Validate and decode a GPT Table descriptor

(GPCF, GPTTable) DecodeGPTTable(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Table;
    GPTTable result;

    if !IsZero(gpt_entry<63:52,11:4>) then
        return (GPCF_Walk, GPTTable UNKNOWN);

    constant integer 10sz = GPTL0Size();
    constant integer p = DecodePGSRange(pgs);

    if !IsZero(gpt_entry<(10sz-p)-2:12>) then
        return (GPCF_Walk, GPTTable UNKNOWN);

    case pgs of
        when PGS_4KB result.address = gpt_entry<55:17>:Zeros(17);
        when PGS_16KB result.address = gpt_entry<55:15>:Zeros(15);
        when PGS_64KB result.address = gpt_entry<55:13>:Zeros(13);
        otherwise Unreachable();

    // The address must be within the range covered by the GPT
    if AbovePPS(result.address) then
        return (GPCF_AddressSize, GPTTable UNKNOWN);

    return (GPCF_None, result);

```

### Library pseudocode for shared/translation/gpc/DecodePGS

```

// DecodePGS()
// =====

PGSe DecodePGS(bits(2) pgs)
    case pgs of
        when '00' return PGS_4KB;
        when '10' return PGS_16KB;
        when '01' return PGS_64KB;
        otherwise Unreachable();

```

### Library pseudocode for shared/translation/gpc/DecodePGSRange

```

// DecodePGSRange()
// =====

integer DecodePGSRange(PGSe pgs)
    case pgs of
        when PGS_4KB return GPTRange_4KB;
        when PGS_16KB return GPTRange_16KB;
        when PGS_64KB return GPTRange_64KB;
        otherwise Unreachable();

```

### Library pseudocode for shared/translation/gpc/DecodePPS

```

// DecodePPS()
// =====
// Size of region protected by the GPT, in bits.

integer DecodePPS()
    case GPCCR_EL3.PPS of
        when '000' return 32;
        when '001' return 36;
        when '010' return 40;
        when '011' return 42;
        when '100' return 44;
        when '101' return 48;
        when '110' return 52;
        otherwise Unreachable\(\);

```

### Library pseudocode for shared/translation/gpc/GPCFault

```

// GPCFault()
// =====
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)
    GPCFRecord fault;
    fault.gpf = gpf;
    fault.level = level;
    return fault;

```

### Library pseudocode for shared/translation/gpc/GPCNoFault

```

// GPCNoFault()
// =====
// Returns the default properties of a GPCF that does not represent a f

GPCFRecord GPCNoFault()
    GPCFRecord result;
    result.gpf = GPCF\_None;
    return result;

```

### Library pseudocode for shared/translation/gpc/GPCRegistersConsistent

```

// GPCRegistersConsistent()
// =====
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

boolean GPCRegistersConsistent()

    // Check for Reserved register values
    if GPCCR_EL3.PPS == '111' || DecodePPS\(\) > AArch64.PAMax\(\) then
        return FALSE;
    if GPCCR_EL3.PGS == '11' then
        return FALSE;
    if GPCCR_EL3.SH == '01' then
        return FALSE;

```

```

// Inner and Outer Non-cacheable requires Outer Shareable
if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
    return FALSE;

return TRUE;

```

### Library pseudocode for shared/translation/gpc/GPICheck

```

// GPICheck()
// =====
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access

boolean GPICheck(PASpace paspace, bits(4) gpi)

    case gpi of
        when GPT_NoAccess      return FALSE;
        when GPT_Secure        assert IsFeatureImplemented(FEAT_SEL2); return
        when GPT_NonSecure     return paspace == PAS_NonSecure;
        when GPT_Root          return paspace == PAS_Root;
        when GPT_Realm         return paspace == PAS_Realm;
        when GPT_Any           return TRUE;
        otherwise                 Unreachable();

```

### Library pseudocode for shared/translation/gpc/GPIIndex

```

// GPIIndex()
// =====

integer GPIIndex(bits(56) pa)
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB   return UInt(pa<15:12>);
        when PGS_16KB  return UInt(pa<17:14>);
        when PGS_64KB  return UInt(pa<19:16>);
        otherwise          Unreachable();

```

### Library pseudocode for shared/translation/gpc/GPIValid

```

// GPIValid()
// =====
// Returns whether a given value is a valid encoding for a GPI value

boolean GPIValid(bits(4) gpi)
    if gpi == GPT_Secure then
        return IsFeatureImplemented(FEAT_SEL2);

    return gpi IN {GPT_NoAccess,
                  GPT_NonSecure,
                  GPT_Root,
                  GPT_Realm,
                  GPT_Any};

```

## Library pseudocode for shared/translation/gpc/GPTL0Size

```
// GPTL0Size()
// =====
// Returns number of bits covered by a level 0 GPT entry

integer GPTL0Size()
    case GPCCR_EL3.L0GPTSZ of
        when '0000' return GPTRange\_1GB;
        when '0100' return GPTRange\_16GB;
        when '0110' return GPTRange\_64GB;
        when '1001' return GPTRange\_512GB;
        otherwise Unreachable\(\);
    return 30;
```

## Library pseudocode for shared/translation/gpc/GPTLevel0Index

```
// GPTLevel0Index()
// =====
// Compute the level 0 index based on input PA.

integer GPTLevel0Index(bits(56) pa)
    // Input address and index bounds
    constant integer pps = DecodePPS\(\);
    constant integer l0sz = GPTL0Size\(\);
    if pps <= l0sz then
        return 0;

    return UInt(pa<ppsl0sz>);
```

## Library pseudocode for shared/translation/gpc/GPTLevel1Index

```
// GPTLevel1Index()
// =====
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(56) pa)
    // Input address and index bounds
    constant integer l0sz = GPTL0Size\(\);
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS\_4KB return UInt(pa<l0sz-1:16>);
        when PGS\_16KB return UInt(pa<l0sz-1:18>);
        when PGS\_64KB return UInt(pa<l0sz-1:20>);
        otherwise Unreachable\(\);
```

## Library pseudocode for shared/translation/gpc/GPTWalk

```
// GPTWalk()
// =====
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(56) pa, AccessDescriptor accdesc)

    // GPT base address
```

```

bits(56) base;
pgs = DecodePGS(GPCCR_EL3.PGS);

// The level 0 GPT base address is aligned to the greater of:
// * the size of the level 0 GPT, determined by GPCCR_EL3.{PPS, LOG
// * 4KB
base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
pps = DecodePPS();
l0sz = GPTL0Size();
integer alignment = Max((pps - l0sz) + 3, 12);
base = base AND NOT ZeroExtend(Ones(alignment), 56);

AccessDescriptor gptaccdesc = CreateAccDescGPTW(accdesc);

// Access attributes and address for GPT fetches
AddressDescriptor gptaddrdesc;
gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.ORGN, C
gptaddrdesc.fault = NoFault(gptaccdesc);

// Address of level 0 GPT entry
gptaddrdesc.paddress.paspace = PAS Root;
gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

// Fetch L0GPT entry
bits(64) level_0_entry;
PhysMemRetStatus memstatus;
(memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
if IsFault(memstatus) then
    return (GPCFault(GPCF_EABT, 0), GPTEntry UNKNOWN);

GPTEntry result;
GPTTable table;
GPCF gpf;
case level_0_entry<3:0> of
    when GPT Block
        // Decode the GPI value and return that
        (gpf, result) = DecodeGPTBlock(pgs, level_0_entry);
        result.pa = pa;
        return (GPCFault(gpf, 0), result);
    when GPT Table
        // Decode the table entry and continue walking
        (gpf, table) = DecodeGPTTable(pgs, level_0_entry);
        if gpf != GPCF None then
            return (GPCFault(gpf, 0), GPTEntry UNKNOWN);
    otherwise
        // GPF - invalid encoding
        return (GPCFault(GPCF_Walk, 0), GPTEntry UNKNOWN);

// Must be a GPT Table entry
assert level_0_entry<3:0> == GPT Table;

// Address of level 1 GPT entry
offset = GPTLevel1Index(pa) * 8;
gptaddrdesc.paddress.address = table.address + offset;

// Fetch L1GPT entry
bits(64) level_1_entry;
(memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
if IsFault(memstatus) then
    return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

```

```

        case level_1_entry<3:0> of
            when GPT_Contig
                (gpf, result) = DecodeGPTContiguous(pgs, level_1_entry);
            otherwise
                gpi_index = GPIIndex(pa);
                (gpf, result) = DecodeGPTGranules(pgs, gpi_index, level_1_e

        if gpf != GPCF_None then
            return (GPCFault(gpf, 1), GPTEntry UNKNOWN);

        result.pa = pa;
        return (GPCNoFault(), result);
    
```

## Library pseudocode for shared/translation/gpc/GranuleProtectionCheck

```

// GranuleProtectionCheck()
// =====
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDes

    assert IsFeatureImplemented(FEAT_RME);
    // The address to be checked
    address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3.GPC == '0' then
        return GPCNoFault();

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPC_Walk, 0);

    // Input address size check
    if AbovePPS(address.address) then
        if address.paspace == PAS_NonSecure then
            return GPCNoFault();
        else
            return GPCFault(GPC_Fail, 0);

    // GPT base address size check
    bits(56) gpt_base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF_AddressSize, 0);

    // GPT lookup
    (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF_None then
        return gpcf;

    // Check input physical address space against GPI
    permitted = GPICheck(address.paspace, gpt_entry.gpi);

    if !permitted then
        gpcf = GPCFault(GPCF_Fail, gpt_entry.level);
    
```

```

        return gpcf;

    // Check passed

    return GPCNoFault();

```

### **Library pseudocode for shared/translation/gpc/PGS**

```

// PGS
// ===
// Physical granule size

enumeration PGSe {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};

```

### **Library pseudocode for shared/translation/gpc/Table**

```

constant bits(4) GPT_NoAccess    = '0000';
constant bits(4) GPT_Table      = '0011';
constant bits(4) GPT_Block       = '0001';
constant bits(4) GPT_Contig      = '0001';
constant bits(4) GPT_Secure     = '1000';
constant bits(4) GPT_NonSecure  = '1001';
constant bits(4) GPT_Root       = '1010';
constant bits(4) GPT_Realm      = '1011';
constant bits(4) GPT_Any        = '1111';
constant integer GPTRange_4KB   = 12;
constant integer GPTRange_16KB  = 14;
constant integer GPTRange_64KB  = 16;
constant integer GPTRange_2MB   = 21;
constant integer GPTRange_32MB  = 25;
constant integer GPTRange_512MB = 29;
constant integer GPTRange_1GB   = 30;
constant integer GPTRange_16GB  = 34;
constant integer GPTRange_64GB  = 36;
constant integer GPTRange_512GB = 39;

type GPTTable is (
    bits(56) address           // Base address of next table
)

type GPTEntry is (
    bits(4) gpi,                // GPI value for this region
    integer size,               // Region size
    integer contig_size,        // Contiguous region size
    integer level,              // Level of GPT lookup
    bits(56) pa                 // PA uniquely identifying the GPT entry
)

```

### **Library pseudocode for shared/translation/translation/S1TranslationRegime**

```

// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====
// Returns the Exception level controlling the current Stage 1 translation
// part this is unused in code because the System register accessors (SREG)
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);

```

## Library pseudocode for shared/translation/vmsa/AddressDescriptor

```

constant integer FINAL_LEVEL = 3;

// AddressDescriptor
// =====
// Descriptor used to access the underlying memory array.

type AddressDescriptor is (
    FaultRecord      fault,          // fault.statuscode indicates whether
    MemoryAttributes memattrs,
    FullAddress     paddress,
    boolean           slassured,      // Stage 1 Assured Translation Property
    boolean           s2fs1mro,       // Stage 2 MRO permission for Stage 1
    bits(16)          mecid,          // FEAT_MEC: Memory Encryption Context ID
    bits(64)          vaddress
)

```

## Library pseudocode for shared/translation/vmsa/ContiguousSize

```

// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(bit d128, TGx tgx, integer level)
    if d128 == '1' then
        case tgx of
            when TGx_4KB
                assert level IN {1, 2, 3};
                return if level == 1 then 2 else 4;
            when TGx_16KB
                assert level IN {1, 2, 3};
                if level == 1 then
                    return 2;

```

```

        elseif level == 2 then
            return 4;
        else
            return 6;
    when TGx_64KB
        assert level IN {2, 3};
        return if level == 2 then 6 else 4;
else
    case tgx of
    when TGx_4KB
        assert level IN {1, 2, 3};
        return 4;
    when TGx_16KB
        assert level IN {2, 3};
        return if level == 2 then 5 else 7;
    when TGx_64KB
        assert level IN {2, 3};
        return 5;

```

## Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```

// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                         MemoryAttributes memattrs)
{
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault();
    addrdesc.s1assured = FALSE;

    return addrdesc;
}

```

## Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```

// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicated

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault,
                                                AddressDescriptor addrdesc);

    addrdesc.vaddress = va;
    addrdesc.fault    = fault;

    return addrdesc;
}

```

## Library pseudocode for shared/translation/vmsa/DecodePASpace

```

// DecodePASpace()
// =====
// Decode the target PA Space

PASpace DecodePASpace (bit nse, bit ns)
    case nse:ns of
        when '00'    return PAS_Secure;
        when '01'    return PAS_NonSecure;
        when '10'    return PAS_Root;
        when '11'    return PAS_Realm;

```

### Library pseudocode for shared/translation/vmsa/DescriptorType

```

// DescriptorType
// =====
// Translation table descriptor formats

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};

```

### Library pseudocode for shared/translation/vmsa/Domains

```

constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client    = '01';
constant bits(2) Domain_Manager  = '11';

```

### Library pseudocode for shared/translation/vmsa/FetchDescriptor

```

// FetchDescriptor()
// =====
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
                                         AccessDescriptor walkaccess, FaultRecord
                                         integer N)
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    // 128-bit descriptors for AArch64 when FEAT_D128 is set and {V}TCR.D128=1
    assert N == 32 || N == 64 || N == 128;
    bits(N) descriptor;
    FaultRecord fault = fault_in;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkaccess);

```

```

if IsFault(memstatus) then
    boolean iswrite = FALSE;
    fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                    walkaccess, N DIV 8, fault);
    if IsFault(fault.statuscode) then
        return (fault, bits(N) UNKNOWN);

if ee == '1' then
    descriptor = BigEndianReverse(descriptor);

return (fault, descriptor);

```

### Library pseudocode for shared/translation/vmsa/HasUnprivileged

```

// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });

```

### Library pseudocode for shared/translation/vmsa/Regime

```

// Regime
// =====
// Translation regimes

enumeration Regime {
    Regime_EL3,                      // EL3
    Regime_EL30,                     // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,                      // EL2
    Regime_EL20,                     // EL2&0
    Regime_EL10                      // EL1&0
};

```

### Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

```

// RegimeUsingAArch32()
// =====
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2 return ELUsingAArch32(EL2);
        when Regime_EL3 return FALSE;

```

## Library pseudocode for shared/translation/vmsa/S1TTWParams

```
// S1TTWParams
// =====
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor)

type S1TTWParams is (
    // A64-VMSA exclusive parameters
    bit          ha,           // TCR_ELx.HA
    bit          hd,           // TCR_ELx.HD
    bit          tbi,          // TCR_ELx.TBI{x}
    bit          tbid,         // TCR_ELx.TBID{x}
    bit          nfd,          // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EI
    bit          e0pd,         // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EI
    bit          d128,         // TCR_ELx.D128
    bit          aie,          // (TCR2_ELx/TCR_EL3).AIE
    MAIRType    mair2,        // MAIR2_ELx
    bit          ds,           // TCR_ELx.DS
    bits(3)     ps,           // TCR_ELx.{I}PS
    bits(6)     txsz,         // TCR_ELx.TxSZ
    bit          epan,         // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EI
    bit          dct,          // HCR_EL2.DCT
    bit          nv1,          // HCR_EL2.NV1
    bit          cmow,         // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EI
    bit          pnch,         // TCR{2}_ELx.PnCH
    bit          disch,        // TCR{2}_ELx.DisCH
    bit          haft,         // TCR{2}_ELx.HAFT
    bit          mtx,          // TCR_ELx.MTX{y}
    bits(2)     skl,          // TCR_ELx.SKL
    bit          pie,          // TCR2_ELx.PIE or TCR_EL3.PIE
    S1PIRTypel pir,          // PIR_ELx
    S1PIRTypel pire0,        // PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H
    bit          emec,         // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    bit          amec,         // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EI

    // A32-VMSA exclusive parameters
    bits(3)     t0sz,         // TTBCR.T0SZ
    bits(3)     t1sz,         // TTBCR.T1SZ
    bit          uwxn,         // SCTLR.UWXN

    // Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx         tgx,          // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)     irgn,         // TCR_ELx.IRGNx   / TTBCR.IRGNx or HTCR.IRGNx
    bits(2)     orgn,         // TCR_ELx.ORGNx  / TTBCR.ORGNx or HTCR.ORGNx
    bits(2)     sh,           // TCR_ELx.SHx     / TTBCR.SHx or HTCR.SHx
    bit          hpd,          // TCR_ELx.HPD{x}  / TTBCR2.HPDx or HTCR.HPDx
    bit          ee,           // SCTLR_ELx.EE    / SCTLR.EE or HSCTLR.EE
    bit          wxn,          // SCTLR_ELx.WXN   / SCTLR.WXN or HSCTLR.WXN
    bit          ntlsmd,       // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit          dc,            // HCR_EL2.DC     / HCR.DC
    bit          sif,           // SCR_EL3.SIF    / SCR.SIF
    MAIRType    mair,          // MAIR_ELx       / MAIR1:MAIR0 or HMAIR1
)
```

## Library pseudocode for shared/translation/vmsa/S2TTWParams

```

// S2TTWParams
// =====
// Register fields corresponding to stage 2 translation.

type S2TTWParams is (
    // A64-VMSA exclusive parameters
    bit          ha,           // VTCR_EL2.HA
    bit          hd,           // VTCR_EL2.HD
    bit          sl2,          // V{S}TCR_EL2.SL2
    bit          ds,            // VTCR_EL2.DS
    bit          d128,         // VTCR_ELx.D128
    bit          sw,            // VSTCR_EL2.SW
    bit          nsw,           // VTCR_EL2.NSW
    bit          sa,            // VSTCR_EL2.SA
    bit          nsa,           // VTCR_EL2.NSA
    bits(3)      ps,            // VTCR_EL2.PS
    bits(6)      txsz,          // V{S}TCR_EL2.T0SZ
    bit          fwb,           // HCR_EL2.PTW
    bit          cmow,          // HCRX_EL2.CMOW
    bits(2)      skl,           // VTCR_EL2.SKL
    bit          s2pie,         // VTCR_EL2.S2PIE
    S2PIRType     s2pir,          // S2PIR_EL2
    bit          t10,           // VTCR_EL2.TL0
    bit          t11,           // VTCR_EL2.TL1
    bit          assuredonly, // VTCR_EL2.AssuredOnly
    bit          haft,           // VTCR_EL2.HAFT
    bit          emec,           // SCTLR2_EL2.EMEC

    // A32-VMSA exclusive parameters
    bit          s,              // VTCR.S
    bits(4)      t0sz,          // VTCR.T0SZ

    // Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    TGx        tgx,           // V{S}TCR_EL2.TG0 / Always TGx_4KB
    bits(2)      sl0,           // V{S}TCR_EL2.SL0 / VTCR.SL0
    bits(2)      irgn,          // VTCR_EL2.IRGN0 / VTCR.IRGN0
    bits(2)      orgn,          // VTCR_EL2.ORGN0 / VTCR.ORGN0
    bits(2)      sh,             // VTCR_EL2.SH0 / VTCR.SH0
    bit          ee,             // SCTLR_EL2.EE / HSCTLR.EE
    bit          ptw,            // HCR_EL2.PTW / HCR.PTW
    bit          vm,             // HCR_EL2.VM / HCR.VM
)

```

## Library pseudocode for shared/translation/vmsa/SDFType

```

// SDFTyp
// =====
// Short-descriptor format type

enumeration SDFTyp {
    SDFTyp_Table,
    SDFTyp_Invalid,
    SDFTyp_Supersection,
    SDFTyp_Section,
    SDFTyp_LargePage,
    SDFTyp_SmallPage
};

```

## Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()
// =====
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime EL3          return SecurityStateAtEL(EL3);
        when Regime EL30         return SS_Secure; // A32 EL3 is always Secure
        when Regime EL2          return SecurityStateAtEL(EL2);
        when Regime EL20         return SecurityStateAtEL(EL2);
        when Regime EL10         return SecurityStateAtEL(EL1);
```

## Library pseudocode for shared/translation/vmsa/StageOA

```
// StageOA()
// =====
// Given the final walk state (a page or block descriptor), map the untrusted
// input address bits to the output address

FullAddress StageOA(bits(64) ia, bit d128, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    integer csize;

    tsize = TranslationSize(d128, tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize = ContiguousSize(d128, tgx, walkstate.level);
    else
        csize = 0;

    constant integer ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<55:ia_msb>:ia<ia_msb-1:0>

    return oa;
```

## Library pseudocode for shared/translation/vmsa/TGx

```
// TGx
// ===
// Translation granules sizes

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

## Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// =====
```

```

// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx 4KB return 12;
        when TGx 16KB return 14;
        when TGx 64KB return 16;

```

### Library pseudocode for shared/translation/vmsa/TLBContext

```

// TLBContext
// =====
// Translation context compared on TLB lookups and invalidations, promoted
// by the TLB

type TLBContext is (
    SecurityState ss,
    Regime          regime,
    bits(16)          vmid,
    bits(16)          asid,
    bit               nG,
    PASpace         ipaspace, // Used in stage 2 lookups & invalidations
    boolean           includes_s1,
    boolean           includes_s2,
    boolean           includes_gpt,
    bits(64)          ia,        // Input Address
    TGx             tg,
    bit               cnp,
    integer           level,     // Assist TLBI level hints (FEAT_TTL)
    boolean           isd128,
    bit               xs         // XS attribute (FEAT_XS)
)

```

### Library pseudocode for shared/translation/vmsa/TLBRecord

```

// TLBRecord
// =====
// Translation output as a TLB payload

type TLBRecord is (
    TLBContext   context,
    TTWState      walkstate,
    integer          blocksize,    // Number of bits directly mapped from IA
    integer          contigsize,   // Number of entries log 2 marking a contiguous
    bits(128)        s1descriptor, // Stage 1 leaf descriptor in memory (valid)
    bits(128)        s2descriptor // Stage 2 leaf descriptor in memory (valid)
)

```

### Library pseudocode for shared/translation/vmsa/TTWState

```

// TTWState
// =====
// Translation table walk state

type TTWState is (
    boolean          istable,

```

```

        integer          level,
        FullAddress    baseaddress,
        bit              contiguous,
        boolean         s1assured,      // Stage 1 Assured Translation
        bit              s2assuredonly, // Stage 2 AssuredOnly attribut
        bit              disch,         // Stage 1 Disable Contiguous H
        bit              nG,
        bit              guardedpage,
        SDFType       sdftype,        // AArch32 Short-descriptor format
        bits(4)         domain,        // AArch32 Short-descriptor format
        MemoryAttributes memattrs,
        Permissions   permissions
)

```

### Library pseudocode for shared/translation/vmsa/TranslationRegime

```

// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el)
  if el == EL3 then
    return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
  elseif el == EL2 then
    return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
  elseif el == EL1 then
    return Regime_EL10;
  elseif el == EL0 then
    if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3)
      return Regime_EL30;
    elseif ELIsInHost(EL0) then
      return Regime_EL20;
    else
      return Regime_EL10;
  else
    Unreachable();

```

### Library pseudocode for shared/translation/vmsa/TranslationSize

```

// TranslationSize()
// =====
// Compute the number of bits directly mapped from the input address
// to the output address

integer TranslationSize(bit d128, TGx tgx, integer level)
  granulebits = TGxGranuleBits(tgx);
  descsizeilog2 = if d128 == '1' then 4 else 3;
  blockbits   = (FINAL LEVEL - level) * (granulebits - descsizeilog2);

  return granulebits + blockbits;

```

### Library pseudocode for shared/translation/vmsa/UseASID

```

// UseASID()
// =====

```

```
// Determine whether the translation context for the access requires ASID  
  
boolean UseASID(TLBContext accesscontext)  
    return HasUnprivileged(accesscontext.regime);
```

## Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()  
// ======  
// Determine whether the translation context for the access requires VMID  
  
boolean UseVMID(TLBContext accesscontext)  
    return accesscontext.regime == Regime\_EL10 && EL2Enabled();
```

---

[Base Instructions](#)

[SIMD&FP Instructions](#)

[SVE Instructions](#)

[SME Instructions](#)

[Index by Encoding](#)

Sh  
Pseu

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode no\_diffs\_2023\_09\_RC2, sve v2023-06\_rel ; Build timestamp: 2023-09-18T17:56

Copyright © 2010-2023 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.