SETGPN, SETGMN, SETGEN

Memory Set with tag setting, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and performs an implementation defined amount of the memory set. SETGMN performs an implementation defined amount of the memory set. SETGEN performs the last part of the memory set.

Note

The inclusion of implementation defined amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is implementation defined.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an implementation defined number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an implementation defined number of bytes set.
- Xn holds the saturated Xn - an implementation defined number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

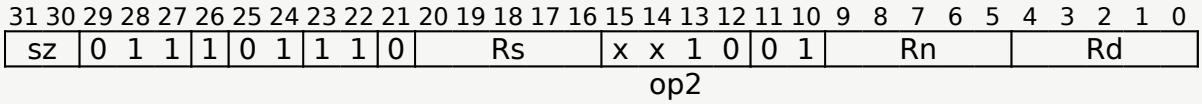
For SETGEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)



Epilogue (op2 == 1010)

```
SETGEN [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0110)

```
SETGMN [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0010)

```
SETGPN [<Xd>]!, <Xn>!, <Xs>

if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) then

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

For information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

	For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.
<Xs>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field. For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
boolean is_setg = TRUE;
integer B;

boolean implements_option_a = SETGOptionA();
boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPr

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !Is
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

    if implements_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';
else
    CheckMemSetParams(stage, implements_option_a, nzcv, options, d, s,

    if ((!IsZero(setsize) && !IsAligned(toaddress, TAG_GRANULE)) || !Is
        AArch64.Abort(toaddress, AlignmentFault(accdesc));

bits(64) stagesetsize = MemSetStageSize(stage, toaddress, setsize, is_s

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if implements_option_a then
```

```
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While
        // implementations might make this constant, that is not assume
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, accdesc] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While
        // implementations might make this constant, that is not assume
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, accdesc] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, accdesc] = tag;
            tagstep = tagstep - 1;

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```


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