OVP VMI Morph Time Function Reference

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	7.42.0
Filename:	OVP_VMI_Morph_Time_Function_Reference.doc
Project:	OVP VMI Morph Time Function Reference
Last Saved:	Thursday, 25 March 2021
Keywords:	

Table of Contents

1 Introduction

This is reference documentation for **version 7.42.0** of the VMI *morph time* function interface, defined in `ImpPublic/include/host/vmi/vmiMt.h`.

It also gives details of the VMI *instruction fetch* interface, defined in `ImpPublic/include/host/vmi/vmiCxt.h`, and *instruction decoder* function interface which greatly simplifies the creation of robust and correct instruction decoders. This interface is defined in `ImpPublic/include/host/vmi/vmiDecode.h`.

The functions in the VMI morph time function interface are used to define instruction behavior of a simulated processor, and are callable only within or beneath the processor *morph callback* function (defined with the `VMI_MORPH_FN` macro, installed as the `morphCB` field of the processor `vmiIASAttr` structure).

The morph callback performs the following actions:

1. It fetches an instruction at a simulated address supplied as an argument.
2. It decodes the instruction (for example, by a cascaded `if` driven by bit fields extracted from the fetched instruction, or by using the decoder function interface);
3. It calls one or more of the routines specified here to describe the behavior of the instruction.

Functions in section 3 of this document show how the fetch and decode support routines are used to implement steps 1 and 2 above.

Remaining examples in this document describe step 3 only – the starting point for each is a small *emission* function that is assumed to be called with appropriate arguments extracted from a decoded instruction.

See the *Imperas Processor Modeling Guide* for a detailed explanation of the steps required to model a processor using the functions in this interface.

2 Interaction with Imperas Simulators

Processor models developed using this interface can be used with both Imperas OVP platforms and the Imperas Simulator (`imperas.exe`) simulation product.

It is important to understand at a high level how the simulators use the morph callback function, and what is happening when it is called. This is briefly described here.

1. When the simulator executes a branch to a simulated address that it has not previously encountered, it calls the morph callback to translate a sequence of simulated opcodes into native machine code. The code block is terminated when the simulator detects a subsequent branch or jump instruction[1]. It then executes that native code.
2. Previously-encountered translated sequences (*code blocks*) are cached in a *dictionary*. If the simulator executes the same code again, it will reuse the cached code block and not call the morph callback.
3. It is very important to understand that the morph callback does not execute simulated instructions: instead, **it describes the behavior of those instructions**, using a sequence of VMI morph time interface calls.
4. The VMI morph time interface routines generate an ordered list of *native machine interface* (NMI) nodes which, when processed in order, together describe the full behavior of an instruction.
5. When the simulator has assembled an NMI node list for a complete code block (which can contain many instructions), the list is passed to a compiler module which generates an equivalent native code block.

[1] Or by the `vmimtEndBlock` function, described later in this document.

3 Instruction Fetch and Decode Support Routines

The VMI morph-time routines described in this manual and processor model disassembler routines both require support routines for the fetch and decode of instructions.

File `ImpPublic/include/host/vmi/vmiCxt.h` provides an API for instruction fetch.

File `ImpPublic/include/host/vmi/vmiDecode.h` provides an API to simplify decode of fetched instructions.

3.1 vmicxtFetch[1248]Byte

Prototypes

```
Uns8  vmicxtFetch1Byte(vmiProcessorP processor, Addr simAddress);
Uns16 vmicxtFetch2Byte(vmiProcessorP processor, Addr simAddress);
Uns32 vmicxtFetch4Byte(vmiProcessorP processor, Addr simAddress);
Uns64 vmicxtFetch8Byte(vmiProcessorP processor, Addr simAddress);
```

Description

These four routines fetch (respectively) 1, 2, 4 and 8 byte instruction words from the passed address for the passed processor. The endianness of the fetch is specified by the current processor endianness.

Example

This example demonstrates usage of `vmicxtFetch4Byte` for the OR1K training examples.

```
//
// Decode the OR1K instruction at the passed address. If the decode succeeds,
// dispatch it to the corresponding function in the dispatch table and return
// True; otherwise, dispatch using the defaultCB and return False.
//
Bool or1kDecode(
    or1kP               or1k,
    Uns32               thisPC,
    or1kDispatchTableCP table,
    or1kDispatchFn      defaultCB,
    void               *userData,
    Bool                inDelaySlot
) {
    // get the instruction at the passed address - always 4 bytes on OR1K
    vmiProcessorP       processor   = (vmiProcessorP)or1k;
    Uns32               instruction = vmicxtFetch4Byte(processor, thisPC);
    or1kInstructionType type        = decode(instruction);

    // apply the callback, or the default if no match
    if(type!=OR1K_IT_LAST) {
        ((*table)[type])(or1k, thisPC, instruction, userData, inDelaySlot);
        return True;
    } else {
        defaultCB(or1k, thisPC, instruction, userData, inDelaySlot);
        return False;
    }
}
```

Notes and Restrictions

1. Multiple calls to `vmicxt` routines may be used to fetch parts of a single instruction. For example, a CISC processor mode (such as an x86) can use `vmicxtFetch1Byte` to get the first instruction byte and then, depending on the value fetched, use further `vmicxt` functions calls to get subsequent instruction bytes.

3.2 vmidNewDecodeTable

Prototype

```
vmidDecodeTableP vmidNewDecodeTable(Uns32 bits, Uns32 defaultValue);
```

Description

This function returns a new *decode table* object, that is used to construct robust and efficient instruction decoders. The decode table decodes instructions of width `bits`. `defaultValue` specifies a value that is returned by function `vmidDecode` if an unrecognized instruction is encountered.

Example

This example is part of the OR1K training examples.

```
//
// This macro adds a decode table entry for a specific instruction class
//
#define DECODE_ENTRY(_PRIORITY, _NAME) \
    vmidNewEntry(           \
        table,              \
        #_NAME,             \
        OR1K_IT_##_NAME,    \
        MASK_##_NAME,       \
        OP_##_NAME,         \
        _PRIORITY           \
    )

//
// Create the OR1K decode table
//
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(OR1K_BITS, OR1K_IT_LAST);

    // handle movhi instruction
    DECODE_ENTRY(0, MOVHI);

    // handle arithmetic instructions (second argument constant)
    DECODE_ENTRY(0, ADDI);
    DECODE_ENTRY(0, ADDIC);
    DECODE_ENTRY(0, ANDI);
    DECODE_ENTRY(0, ORI);
    DECODE_ENTRY(0, XORI);
    DECODE_ENTRY(0, MULI);
    … etc …
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64 currently.

3.3 *vmidNewEntry*

Prototype

```
Bool vmidNewEntry(
    vmidDecodeTableP table,
    const char      *name,
    Uns32            matchValue,
    Uns64            mask,
    Uns64            value,
    Int32            priority
);
```

Description

Given a previously-created decode table object, this function adds a new decode entry to that table. Each decode entry decodes a single instruction type. The name of the entry is given by the `name` argument (this is informative only and used in error messages).

An instruction matches the new entry if:

```
(instruction & mask) == value
```

If this decode entry matches an instruction, `vmidDecode` will return `matchValue` (which is typically a processor-model-specific enumeration member).

It is possible that multiple entries in a decode table match the same instruction pattern – for example, often a RISC move instruction is a special case of an arithmetic instruction (such as an add). If such conflicting entries are required, they must be given distinct `priority` values, and the entry with greatest priority is deemed to match. A good default method to specify a reasonable priority for an instruction is to *use the number of non-zero bits in the `mask`*: this can be specified using the special value `VMID_DERIVE_PRIORITY` for the mask. To distinguish the kinds of conflict described above, it is possible to use an expression such as `VMID_DERIVE_PRIORITY+1` (indicating a priority one higher than the automatically-derived priority based on non-zero `mask` bits). If two entries with the same priority both match a candidate instruction, a decode table entry conflict error will be generated when `vmidDecode` is first called for the table.

If the decode entry was successfully created, `vmidNewEntry` returns `True`. Otherwise (if the decode table is already in use or `value` specifies bits that are not selected by `mask`) it returns `False`.

Typically, calls to `vmidNewEntry` are used within a macro as in the example below.

See also function `vmidNewEntryFmtBin`, which enables decode table entries to be created from format strings.

Example

This example is part of the OR1K training examples. The decode for each opcode is specified by patterns in file `or1kInstructions.h`:

```
////////////////////////////////////////////////////////////////////////////////
// OPCODE FORM 4
// OPCODE(6) D(5) UNUSED(4) OPCODE(1) I(16)
////////////////////////////////////////////////////////////////////////////////
#define OP4(_OP1, _OP2) (Uns32)((WIDTH(6,_OP1)<<26) | (WIDTH(1,_OP2)<<16))

#define OP4_MASK        OP4(-1, -1)
#define OP4_D(_I)       WIDTH(5,(_I)>>21)
#define OP4_I(_I)       WIDTH(16,(_I)>>0)
#define MASK_MOVHI      OP4_MASK
#define OP_MOVHI        OP4(0x06, 0x0)
```

An enumeration specifying the different instruction types is in `or1kDecode.h`:

```
//
// Instruction type enumeration
//
typedef enum or1kInstructionTypeE {

    // movhi instruction
    OR1K_IT_MOVHI,

    // arithmetic instructions (second argument constant)
    OR1K_IT_ADDI,
    OR1K_IT_ADDIC,
    OR1K_IT_ANDI,
    OR1K_IT_ORI,
    OR1K_IT_XORI,
    OR1K_IT_MULI,
    … etc …
}
```

Then the decode table is filled by function `createDecodeTable` in file `or1kDecode.c`:

```
//
// This macro adds a decode table entry for a specific instruction class
//
#define DECODE_ENTRY(_PRIORITY, _NAME) \
    vmidNewEntry(           \
        table,              \
        #_NAME,             \
        OR1K_IT_##_NAME,    \
        MASK_##_NAME,       \
        OP_##_NAME,         \
        _PRIORITY           \
    )

//
// Create the OR1K decode table
//
static vmidDecodeTableP createDecodeTable(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(OR1K_BITS, OR1K_IT_LAST);

    // handle movhi instruction
    DECODE_ENTRY(0, MOVHI);

    // handle arithmetic instructions (second argument constant)
    DECODE_ENTRY(0, ADDI);
    DECODE_ENTRY(0, ADDIC);
    DECODE_ENTRY(0, ANDI);
    DECODE_ENTRY(0, ORI);
    DECODE_ENTRY(0, XORI);
    DECODE_ENTRY(0, MULI);
    … etc …
}
```

Notes and Restrictions

1. Entries may not be added to a decode table after `vmidDecode` has been called on that table.
2. `mask` must select all non-zero bits in `value` (i.e. `(mask&~value)` must be zero).

3.4 vmidNewEntryFmtBin

Prototype

```
Bool vmidNewEntryFmtBin(
    vmidDecodeTableP table,
    const char      *name,
    Uns32            matchValue,
    const char      *format,
    Int32            priority
);
```

Description

Given a previously-created decode table object, this function adds a new decode entry to that table. Each decode entry decodes a single instruction type. The name of the entry is given by the `name` argument (this is informative only and used in error messages).

The instruction format string, `format`, contains three kinds of characters:

1. *constrained* characters (either '0' or '1') – the corresponding bit in the instruction must have the same value;
2. *spacer* characters (any of '|', '/', comma, space or tab) – these are ignored and can be freely used to improve readability of the format string;
3. *don't care* characters (any character not listed above) – these characters can be either 0 or 1 in the instruction and it will still match.

The format specifies bits in the instruction in most-significant bit to least-significant bit order. For example, the following pattern could be used to create a decode table entry that matches a 16-bit instruction with the five most-significant bits `01001` and the three least significant bits `110`:

```
"01001........110"
```

The case above uses the dot character as a *don't care* character. Here is another example that matches exactly the same instruction pattern, but using *x* as a don't care character and using vertical-bar spacer characters to improve readability:

```
"|01001|xxxxxxxx|110|"
```

It is possible that multiple entries in a decode table match the same instruction pattern – for example, often a RISC move instruction is a special case of an arithmetic instruction (such as an add). If such conflicting entries are required, they must be given distinct `priority` values, and the entry with greatest priority is deemed to match. A good default method to specify a reasonable priority for an instruction is to *use the number of 1 characters in the `format`* string: this can be specified using the special value `VMID_DERIVE_PRIORITY` for the mask. To distinguish the kinds of conflict described above, it is possible to use an expression such as `VMID_DERIVE_PRIORITY+1` (indicating a priority one higher than the automatically-derived priority based on 1 characters in the `format` string). If two entries with the same priority both match a candidate instruction, a decode table entry conflict error will be generated when `vmidDecode` is first called for the table.

If the decode entry was successfully created, `vmidNewEntryFmtBin` returns `True`. Otherwise (if the decode table is already in use, or the pattern string has the wrong number of characters) it returns `False`.

Typically, calls to `vmidNewEntryFmtBin` are used within a macro as in the example below.

See also function `vmidNewEntry`, which enables decode table entries to be created from mask/value pairs.

Example

This example is part of the ARC processor model example. An enumeration specifying the different instruction types is in `arcDecodeTypes.h`:

```
typedef enum arcInstructionTypeE {

    ////////////////////////////////////////////////////////////////////////////
    // 32-BIT INSTRUCTIONS
    ////////////////////////////////////////////////////////////////////////////

    // nonary instructions
    ITYPE_SET_32_0 (SWI),
    ITYPE_SET_32_0 (SYNC),
    ITYPE_SET_32_0 (RTIE),
    ITYPE_SET_32_0 (BRK),
    ITYPE_SET_32_0 (NOP),

    // unary instructions (with major opcode 0x04)
    ITYPE_SET_32_1 (ASL),
    ITYPE_SET_32_1 (ASR),
    ITYPE_SET_32_1 (LSR),
    ITYPE_SET_32_1 (ROR),
    ITYPE_SET_32_1 (RRC),
    ITYPE_SET_32_1 (SEXB),
    ITYPE_SET_32_1 (SEXW),
    ITYPE_SET_32_1 (EXTB),
    ITYPE_SET_32_1 (EXTW),
    ITYPE_SET_32_1 (ABS),
    ITYPE_SET_32_1 (NOT),
    ITYPE_SET_32_1 (RLC),

    … etc …
}
```

Macros are used to specify the decode entries for each instruction type, for example:

```
#define ITYPE_SET_32_1(_NAME)       \
    ARC_IT_##_NAME##_B_C,           \
    ARC_IT_##_NAME##_B_U6,          \
    ARC_IT_##_NAME##_B_LIMM,        \
    ARC_IT_##_NAME##_0_C,           \
    ARC_IT_##_NAME##_0_U6,          \
    ARC_IT_##_NAME##_0_LIMM
```

Decode entries corresponding to each instruction type are specified in file `arcDecodeEntries.h` using `vmidNewEntryFmtBin`, as follows:

```
#define DECODE_ENTRY(_PRIORITY, _NAME, _PATTERN) \
    vmidNewEntryFmtBin(table, #_NAME, ARC_IT_##_NAME, _PATTERN, _PRIORITY)

#define DECODE_SET_32_1(_NAME, _F1, _F2) \
    DECODE_ENTRY(0, _NAME##_B_C,    "|"_F1"|...|00|101111|.|...|......|"_F2"|"); \
    DECODE_ENTRY(0, _NAME##_B_U6,   "|"_F1"|...|01|101111|.|...|......|"_F2"|"); \
    DECODE_ENTRY(1, _NAME##_B_LIMM, "|"_F1"|...|00|101111|.|...|111110|"_F2"|"); \
    DECODE_ENTRY(2, _NAME##_0_C,    "|"_F1"|110|00|101111|.|111|......|"_F2"|"); \
    DECODE_ENTRY(2, _NAME##_0_U6,   "|"_F1"|110|01|101111|.|111|......|"_F2"|"); \
    DECODE_ENTRY(3, _NAME##_0_LIMM, "|"_F1"|110|00|101111|.|111|111110|"_F2"|")
```

Decode tables are filled by functions `createDecodeTable16` and `createDecodeTable32` in file `arcDecode.c`:

```
static vmidDecodeTableP createDecodeTable32(void) {

    vmidDecodeTableP table = vmidNewDecodeTable(32, ARC_IT_LAST);

    // nonary instructions
    DECODE_SET_32_0 (SWI,  "010", "01", "101111", "000", "111111");
    DECODE_SET_32_0 (SYNC, "011", "01", "101111", "000", "111111");
    DECODE_SET_32_0 (RTIE, "100", "00", "101111", "000", "111111");
    DECODE_SET_32_0 (BRK,  "101", "01", "101111", "000", "111111");
    DECODE_SET_32_0 (NOP,  "110", "01", "001010", "111", "000000");

    // unary instructions (with major opcode 0x04)
    DECODE_SET_32_1 (ASL,      "00100", "000000");
    DECODE_SET_32_1 (ASR,      "00100", "000001");
    DECODE_SET_32_1 (LSR,      "00100", "000010");
    DECODE_SET_32_1 (ROR,      "00100", "000011");
    DECODE_SET_32_1 (RRC,      "00100", "000100");
    DECODE_SET_32_1 (SEXB,     "00100", "000101");
    DECODE_SET_32_1 (SEXW,     "00100", "000110");
    DECODE_SET_32_1 (EXTB,     "00100", "000111");
    DECODE_SET_32_1 (EXTW,     "00100", "001000");
    DECODE_SET_32_1 (ABS,      "00100", "001001");
    DECODE_SET_32_1 (NOT,      "00100", "001010");
    DECODE_SET_32_1 (RLC,      "00100", "001011");

    … etc …
}
```

Notes and Restrictions

1. Entries may not be added to a decode table after `vmidDecode` has been called on that table.
2. The number of constrained and don’t care characters added together must equal the `bits` argument given when the decode table was created.

3.5 vmidDecode

Prototype

```
Uns32 vmidDecode(vmidDecodeTableP table, Uns64 instr);
```

Description

This function decodes the passed instruction value `instr` using the decode table. If the instruction matches some entry in the decode table, the `matchValue` associated with that entry is returned. Otherwise, the `defaultValue` specified when the table was created is returned.

Example

This example is part of the OR1K training examples.

```
//
// Decode the instruction and return an enum describing it
//
static or1kInstructionType decode(Uns32 instruction) {

    // get the OR1K decode table
    static vmidDecodeTableP decodeTable;
    if(!decodeTable) {
        decodeTable = createDecodeTable();
    }

    // decode the instruction to get the type
    or1kInstructionType type = vmidDecode(decodeTable, instruction);

    // some arguments to l.sf and l.sfi are invalid: filter them here
    if((type==OR1K_IT_SF) && !getCmpInfo(OP5_CMPOP(instruction))->name) {
        type = OR1K_IT_LAST;
    } else if((type==OR1K_IT_SFI) && !getCmpInfo(OP6_CMPOP(instruction))->name) {
        type = OR1K_IT_LAST;
    }

    return type;
}
```

Notes and Restrictions

None.

Basic Register Operations

This section describes emission functions for basic register operations: moves, unary operations, binary operations and comparisons.

3.6 Simulated Register Specification Using `vmiReg`

Most functions in this API require use of the `vmiReg` type to specify the location of source and target registers in a structure representing a simulated processor. A short introduction to usage of this type is given here; for a more detailed description, refer to the *Imperas Processor Modeling Guide*.

As an example, the OVP OR1K processor is represented using a structure of type or1k, defined as follows:

```
#define OR1K_REGS 32

typedef struct or1kS {

    Bool          carryFlag;        // carry flag
    Bool          overflowFlag;     // overflow flag
    Bool          branchFlag;       // branch flag

    Uns32         regs[OR1K_REGS];  // basic registers

    . . . fields omitted for clarity . . .

} or1k, *or1kP;
```

Here, for example, the `regs` member holds the value of each of the 32 GPRs. The location of a register (for example, a GPR) is specified to the simulator using the `vmiReg` type, defined in file `vmiTypes.h`. A `vmiReg` structure can be created for any field in a processor structure using the `VMI_CPU_REG` macro, which takes a type pointer and a field name argument. Typically, the processor header files will contain further macros that encapsulate usage of the `VMI_CPU_REG` macro appropriately for that processor: for example, the OVP OR1K model contains these macro definitions:

```
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY              OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW           OR1K_CPU_REG(overflowFlag)
#define OR1K_BRANCH             OR1K_CPU_REG(branchFlag)
```

As an example, this code could now be used to specify the location of the OR1K `branchFlag` register to a morph-time API function:

```
vmiReg bf = OR1K_BRANCH;
```

Typically, usage of registers such as GPRs is encapsulated by sugar routines that handle special values. In the case of the OR1K processor, GPR 0 is always zero and unwritable. Therefore, the following sugar function is used to return an appropriate `vmiReg` for an operation, given a GPR index:

```
static vmiReg getGPR(Uns32 r) {
    return r ? OR1K_REG(r) : VMI_NOREG;
}
```

For conciseness and clarity, Examples listed in this manual will typically refer to `vmiReg` structures without giving details of the processor structure that contains those registers.

3.7 Unary Operation Types

The available unary operations are described by the `vmiUnop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // MOVE OPERATIONS
    vmi_MOV,        // d <- a
    vmi_SWP,        // d <- byteswap(a)

                    // ARITHMETIC OPERATIONS
    vmi_NEG,        // d <- -a
    vmi_ABS,        // d <- (a<0) ? -a : a

                    // SATURATED ARITHMETIC OPERATIONS
    vmi_NEGSQ,      // d <- saturate_signed(-a)
    vmi_ABSSQ,      // d <- (a<0) ? saturate_signed(-a) : a

                    // BITWISE OPERATIONS
    vmi_NOT,        // d <- ~a
    vmi_RBIT,       // d <- bit_reverse(a)

                    // MISCELLANEOUS OPERATIONS
    vmi_CNTZ,       // d <- count_zeros(a)
    vmi_CNTO,       // d <- count_ones(a)
    vmi_CLS,        // d <- count_leading_sign(a)
    vmi_CLZ,        // d <- count_leading_zeros(a)
    vmi_CLO,        // d <- count_leading_ones(a)
    vmi_CTZ,        // d <- count_trailing_zeros(a)
    vmi_CTO,        // d <- count_trailing_ones(a)
    vmi_BSFZ,       // d <- least_significant_zero_index(a)
    vmi_BSFO,       // d <- least_significant_one_index(a)
    vmi_BSRZ,       // d <- most_significant_zero_index(a)
    vmi_BSRO,       // d <- most_significant_one_index(a)

                    // AES ENCRYPTION OPERATIONS
    vmi_AESMC,      // d <- AES_mix_columns(a)
    vmi_AESIMC,     // d <- AES_inverse_mix_columns(a)

    vmi_UNOP_LAST   // KEEP LAST

} vmiUnop;
```

Signed saturation instructions clamp overflowing values to the smallest negative or largest positive value.

Operation `vmi_RBIT` reverses the bit order of the operand.

Operations `vmi_CNTZ` and `vmi_CNTO` count the number of zero and one bits in the argument value, respectively.

Operation `vmi_CLS` counts the number of leading bits that are equal to the sign bit (the most significant bit). The sign bit is not included in this count. Operations `vmi_CLZ`, `vmi_CLO`, `vmi_CTZ` and `vmi_CTO` count the number of leading (most significant) or trailing (least significant) one or zero bits in the argument value. If there are no bits of the required type, the result value is the size of the type in bits. For example, for an 8-bit type, the result will be 8 if there are no bits of the required type in the argument; otherwise it will be a value in the range 0-7.

Operations `vmi_BSFZ`, `vmi_BSFO`, `vmi_BSRZ` and `vmi_BSRO` return the bit index of the most significant (`BSR`) or least significant (`BSF`) one or zero bit in the argument value, where the least significant bit of a value is index 0. If there are no bits of the required type, the result value is the size of the type in bits.

Operations `vmi_AESMC` and `vmi_AESIMC` implement the `MixColumns` and `InvMixColumns` transformations described by the Advanced Encryption Standard (AES) specification. Both take a 64-bit operand and generate a 64-bit result.

3.8 Binary Operation Types

The available binary operations are described by the `vmiBinop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // ARITHMETIC OPERATIONS
    vmi_ADD,        // d <- a + b
    vmi_ADC,        // d <- a + b + C
    vmi_SUB,        // d <- a - b
    vmi_SBB,        // d <- a - b - C
    vmi_RSBB,       // d <- b - a - C
    vmi_RSUB,       // d <- b - a
    vmi_IMUL,       // d <- a * b (signed)
    vmi_MUL,        // d <- a * b (unsigned)
    vmi_IDIV,       // d <- a / b (signed)
    vmi_DIV,        // d <- a / b (unsigned)
    vmi_IREM,       // d <- a % b (signed)
    vmi_REM,        // d <- a % b (unsigned)
    vmi_CMP,        // a - b

                    // SATURATED ARITHMETIC OPERATIONS
    vmi_ADDSQ,      // d <- saturate_signed(a + b)
    vmi_SUBSQ,      // d <- saturate_signed(a - b)
    vmi_RSUBSQ,     // d <- saturate_signed(b - a)
    vmi_ADDUQ,      // d <- saturate_unsigned(a + b)
    vmi_SUBUQ,      // d <- saturate_unsigned(a - b)
    vmi_RSUBUQ,     // d <- saturate_unsigned(b - a)

                    // HALVING ARITHMETIC OPERATIONS
    vmi_ADDSH,      // d <- ((signed)(a + b)) / 2
    vmi_SUBSH,      // d <- ((signed)(a - b)) / 2
    vmi_RSUBSH,     // d <- ((signed)(b - a)) / 2
    vmi_ADDUH,      // d <- ((unsigned)(a + b)) / 2
    vmi_SUBUH,      // d <- ((unsigned)(a - b)) / 2
    vmi_RSUBUH,     // d <- ((unsigned)(b - a)) / 2
    vmi_ADDSHR,     // d <- round(((signed)(a + b)) / 2)
    vmi_SUBSHR,     // d <- round(((signed)(a - b)) / 2)
    vmi_RSUBSHR,    // d <- round(((signed)(b - a)) / 2)
    vmi_ADDUHR,     // d <- round(((unsigned)(a + b)) / 2)
    vmi_SUBUHR,     // d <- round(((unsigned)(a - b)) / 2)
    vmi_RSUBUHR,    // d <- round(((unsigned)(b - a)) / 2)

                    // BITWISE OPERATIONS
    vmi_OR,         // d <- a | b
    vmi_AND,        // d <- a & b
    vmi_XOR,        // d <- a ^ b
    vmi_ORN,        // d <- a | ~b
```

```
    vmi_ANDN,       // d <- a & ~b
    vmi_XORN,       // d <- a ^ ~b
    vmi_NOR,        // d <- ~(a | b)
    vmi_NAND,       // d <- ~(a & b)
    vmi_XNOR,       // d <- ~(a ^ b)

                    // SHIFT/ROTATE OPERATIONS
    vmi_ROL,        // d <- a << b | a >> <bits>-b
    vmi_ROR,        // d <- a >> b | a << <bits>-b
    vmi_RCL,        // (d,c) <- (a,c) << b | (a,c) >> <bits>-b
    vmi_RCR,        // (d,c) <- (a,c) >> b | (a,c) << <bits>-b
    vmi_SHL,        // d <- a << b
    vmi_SHR,        // d <- (unsigned)a >> b
    vmi_SAR,        // d <- (signed)a >> b

                    // SATURATED SHIFT OPERATIONS
    vmi_SHLSQ,      // d <- saturate_signed(a << b)
    vmi_SHLUQ,      // d <- saturate_unsigned(a << b)

                    // ROUNDING SHIFT OPERATIONS
    vmi_SHRR,       // d <- round((unsigned)a >> b)
    vmi_SARR,       // d <- round((signed)a >> b)

                    // MIN/MAX OPERATIONS
    vmi_IMIN,       // d <- min_signed(a, b)
    vmi_MIN,        // d <- min_unsigned(a, b)
    vmi_IMAX,       // d <- max_signed(a, b)
    vmi_MAX,        // d <- max_unsigned(a, b)

                    // WIDENING ARITHMETIC OPERATIONS
    vmi_IMULSU,     // d <- a (signed) * b (unsigned)
    vmi_IMULUS,     // d <- a (unsigned) * b (signed)

                    // POLYNOMIAL ARITHMETIC OPERATIONS
    vmi_PMUL,       // d <- a * b (carryless)

                    // AES ENCRYPTION OPERATIONS
    vmi_AESENC1,    // d <- AES_encrypt1(a), not last round
    vmi_AESENC1L,   // d <- AES_encrypt1(a), last round
    vmi_AESDEC1,    // d <- AES_decrypt1(a), not last round
    vmi_AESDEC1L,   // d <- AES_decrypt1(a), last round
    vmi_AESENC2,    // d <- AES_encrypt2(a), not last round
    vmi_AESENC2L,   // d <- AES_encrypt2(a), last round
    vmi_AESDEC2,    // d <- AES_decrypt2(a), not last round
    vmi_AESDEC2L,   // d <- AES_decrypt2(a), last round

    vmi_BINOP_LAST  // KEEP LAST

} vmiBinop;
```

Signed saturation instructions clamp overflowing values to the smallest negative or largest positive value. *Unsigned saturation* instructions clamp overflowing values to zero or the largest value.

Operations `vmi_AESENC1`, `vmi_AESENC1L`, `vmi_AESDEC1`, `vmi_AESDEC1L`, `vmi_AESENC2`, `vmi_AESENC2L`, `vmi_AESDEC2` and `vmi_AESDEC2L` implement primitives of the Advanced Encryption Standard (AES) algorithm, taking two 64-bit operands and generating a 64-bit result. In the terms of that specification, the operations are as follows:

vmi_AESENC1

```
result = ShiftRows(src1);
result = SubBytes(result);
result = MixColumns(result);
result = result xor src2;
```

vmi_AESENC1L

```
result = ShiftRows(src1);
result = SubBytes(result);
result = result xor src2;
```

vmi_AESDEC1

```
result = InvShiftRows(src1);
result = InvSubBytes(result);
result = InvMixColumns(result);
result = result xor src2;
```

vmi_AESDEC1L

```
result = InvShiftRows(src1);
result = InvSubBytes(result);
result = result xor src2;
```

vmi_AESENC2

```
result = result xor src2;
result = ShiftRows(src1);
result = SubBytes(result);
result = MixColumns(result);
```

vmi_AESENC2L

```
result = result xor src2;
result = ShiftRows(src1);
result = SubBytes(result);
```

vmi_AESDEC2

```
result = result xor src2;
result = InvShiftRows(src1);
result = InvSubBytes(result);
result = InvMixColumns(result);
```

vmi_AESDEC2L

```
result = result xor src2;
result = InvShiftRows(src1);
result = InvSubBytes(result);
```

3.9 Handling Instruction Flags

Several functions in this section use a `vmiFlags` structure to indicate how processor flags are used and affected by translated native code. There is one input flag (carry) and five output flags (carry, parity, zero, sign, and overflow), the behavior of which for any instruction is specified by entries in the structure.

3.9.1 Carry In Flag

The carry in flag is used by arithmetic operations `vmi_ADC` and `vmi_SBB` to provide the input carry or borrow value. For rotate-with-carry operations, the carry participates in an N+1 bit rotation with the N bit operand value.

3.9.2 Carry Out Flag

The carry out flag is set by arithmetic operations to indicate a carry out or a borrow. It is also set by shift and rotate operations to indicate the last bit shifted or rotated out of the operand. For rotate-with-carry operations, the carry participates in an N+1 bit rotation with the N bit operand value; for shifts or rotates of zero, the carry is unchanged. For an

N-bit multiply operation, the carry flag is set if the result was truncated in order to fit into N bits.

3.9.3 Parity Flag

The parity flag indicates the parity of the least significant byte of the result of an operation. If the least significant byte contains an even number of one bits, the flag is set; otherwise, it is cleared.

3.9.4 Zero Flag

The zero flag is set if an operation result is zero and cleared otherwise.

3.9.5 Sign Flag

The sign flag is set if the most significant bit of an operation result is set and cleared otherwise.

3.9.6 Overflow Flag

The overflow flag is set if an arithmetic operation overflowed (the carry into the most significant bit of the result is different to the carry out). For an N-bit multiply operation, the overflow flag is set if the result was truncated in order to fit into N bits.

3.9.7 `vmiFlags` Structure Usage

The carry input flag is be represented in a processor structure by an `Uns8` entry. Each of the five output flags may also be represented by an `Uns8` entry, if required by that model. In order to tell the simulator how to use the flags, each emission function that uses them is passed a pointer to a `vmiFlags` structure:

```
typedef enum {
    vmi_CF=0,           // carry flag
    vmi_PF=1,           // parity flag
    vmi_ZF=2,           // zero flag
    vmi_SF=3,           // sign flag
    vmi_OF=4,           // overflow flag
    vmi_LF=5            // KEEP LAST
} vmiFlag;

//
// Bitmask indicating whether particular flags should be negated
//
typedef enum {
    vmi_FN_NONE  =0x00, // empty negate mask
    vmi_FN_CF_IN =0x01, // negate carry in flag
    vmi_FN_CF_OUT=0x02, // negate carry out flag
    vmi_FN_PF    =0x04, // negate parity flag
    vmi_FN_ZF    =0x08, // negate zero flag
    vmi_FN_SF    =0x10, // negate sign flag
    vmi_FN_OF    =0x20, // negate overflow flag
} vmiFlagNegate;

//
// Processor flag-related structures
//
typedef struct vmiFlagsS {
    vmiReg        cin;          // register specifying carry in
    vmiReg        f[vmi_LF];    // registers to hold operation results
    vmiFlagNegate negate;       // bitmask of negated flags
} vmiFlags;
```

If an emission function is passed a null pointer as its `flags` argument, then the function should neither use nor set any flags. Otherwise, the function should obtain the carry in (if required) from a register described by the `cin` field and write output flags to registers described in the `f` array. If any register is given as `VMI_NOFLAG`, it should be ignored or discarded by the emission function.

There is also a *negate mask* that allows the parity of flags in processor models to be inverted with respect to those in the native processor.

As an example, here is how flag settings could be used to use register `CPUX_CARRY` as an input flag (if required) and store the output carry to register `CPUX_CARRY` and the output overflow to `CPUX_OVERFLOW`:

```
// processor structure definition
```

```
typedef struct cpuxS {
    Uns8  carryFlag;        // carry flag
    Uns8  overflowFlag;     // overflow flag
} cpux, *cpuxP;

// structure field accessor macros
#define CPUX_OFFSET(_F) VMI_CPU_REG(cpuxP, _F)
#define CPUX_CARRY      CPUX_OFFSET(carryFlag);
#define CPUX_OVERFLOW   CPUX_OFFSET(overflowFlag);

const vmiFlags flagsCO = {
    cin : CPUX_CARRY,               // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,      // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,      // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,      // zero flag not used
        [vmi_SF] = VMI_NOFLAG,      // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW    // offset to overflow flag
    }
};
```

For some processors, model flags are required to have inverted polarity with respect to native flags. As an example, for ARM processors, `SBC` and `SUB` instructions use and emit a *borrow* instead of a carry. To simulate this, the carry in must be negated before use (by `SBC`) and the carry out must be negated before being written to the model structure. This can be specified by using the `negate` mask in the `vmiFlags` structure as follows:

```
const vmiFlags flagsCO = {
    cin : CPUX_CARRY,               // offset to carry in flag
    f   : {
        [vmi_CF] = CPUX_CARRY,      // offset to carry out flag
        [vmi_PF] = VMI_NOFLAG,      // parity flag not used
        [vmi_ZF] = VMI_NOFLAG,      // zero flag not used
        [vmi_SF] = VMI_NOFLAG,      // sign flag not used
        [vmi_OF] = CPUX_OVERFLOW    // offset to overflow flag
    },
    vmi_FN_CF_IN|vmi_FN_CF_OUT      // negate carry in and carry out
};
```

Not all operations use or set all the flags; the following table gives usage for unary operation types (defined in `vmiTypes.h`).

```
    ---------- ----- --------------------
    OPERATION  INPUT       OUTPUT
    ---------- ----- --------------------
                CY    CY   PF   ZF   SF   OF
    ---------- ----- --------------------
    vmi_MOV     -      -    A    A    A    -
    vmi_SWP     -      -    A    A    A    -
    vmi_NEG     -      A    A    A    A    A
    vmi_ABS     -      A    A    A    A    A
    vmi_NEGSQ   -      A    A    A    A    A
    vmi_ABSSQ   -      A    A    A    A    A
    vmi_NOT     -      -    A    A    A    -
    vmi_RBIT    -      0    A    A    A    0
    vmi_CNTZ    -      0    A    A    0    0
    vmi_CNTO    -      0    A    A    0    0
    vmi_CLS     -      0    A    A    0    0
    vmi_CLZ     -      0    A    A    0    0
    vmi_CLO     -      0    A    A    0    0
    vmi_CTZ     -      0    A    A    0    0
    vmi_CTO     -      0    A    A    0    0
    vmi_BSFZ    -      0    A    A    0    0
    vmi_BSFO    -      0    A    A    0    0
    vmi_BSRZ    -      0    A    A    0    0
    vmi_BSRO    -      0    A    A    0    0
```

```
vmi_AESMC   -     -   -   -   -   -
vmi_AESIMC  -     -   -   -   -   -
---------- ----- -------------------


------ ----------------------------
SYMBOL MEANING
------ ----------------------------
-      Unused or unaffected
A      Flag affected (output)
0      Flag cleared (output)
------ ----------------------------
```

The following table gives usage for binary operation types (defined in `vmiTypes.h`).

```
------------ ----- --------------------
OPERATION    INPUT       OUTPUT
------------ ----- --------------------
              CY    CY  PF  ZF  SF  OF
------------ ----- --------------------
vmi_ADD       -     A   A   A   A   A
vmi_ADC       U     A   A   A   A   A
vmi_SUB       -     A   A   A   A   A
vmi_SBB       U     A   A   A   A   A
vmi_RSBB      U     A   A   A   A   A
vmi_RSUB      -     A   A   A   A   A
vmi_IMUL      -     A   A   A   A   A
vmi_MUL       -     A   A   A   A   A
vmi_IDIV      -     -   A   A   A   -
vmi_DIV       -     -   A   A   A   -
vmi_IREM      -     -   A   A   A   -
vmi_REM       -     -   A   A   A   -
vmi_CMP       -     A   A   A   A   A
vmi_ADDSQ     -     A   A   A   A   A
vmi_SUBSQ     -     A   A   A   A   A
vmi_RSUBSQ    -     A   A   A   A   A
vmi_ADDUQ     -     A   A   A   A   A
vmi_SUBUQ     -     A   A   A   A   A
vmi_RSUBUQ    -     A   A   A   A   A
vmi_ADDSH     -     A   A   A   A   -
vmi_SUBSH     -     A   A   A   A   -
vmi_RSUBSH    -     A   A   A   A   -
vmi_ADDUH     -     A   A   A   A   -
vmi_SUBUH     -     A   A   A   A   -
vmi_RSUBUH    -     A   A   A   A   -
vmi_ADDSHR    -     A   A   A   A   -
vmi_SUBSHR    -     A   A   A   A   -
vmi_RSUBSHR   -     A   A   A   A   -
vmi_ADDUHR    -     A   A   A   A   -
vmi_SUBUHR    -     A   A   A   A   -
vmi_RSUBUHR   -     A   A   A   A   -
vmi_OR        -     0   A   A   A   0
vmi_AND       -     0   A   A   A   0
vmi_XOR       -     0   A   A   A   0
vmi_ORN       -     0   A   A   A   0
vmi_ANDN      -     0   A   A   A   0
vmi_XORN      -     0   A   A   A   0
vmi_NOR       -     0   A   A   A   0
vmi_NAND      -     0   A   A   A   0
vmi_XNOR      -     0   A   A   A   0
vmi_ROL       U0    A   A   A   A   -
vmi_ROR       U0    A   A   A   A   -
vmi_RCL       U     A   A   A   A   -
vmi_RCR       U     A   A   A   A   -
vmi_SHL       U0    A   A   A   A   -
vmi_SHR       U0    A   A   A   A   -
vmi_SAR       U0    A   A   A   A   -
vmi_SHLSQ     -     A   A   A   A   A
vmi_SHLUQ     -     A   A   A   A   A
vmi_SHRR      -     A   A   A   A   -
```

```
    vmi_SARR      -     A   A   A   A   -
    vmi_IMIN      -     -   A   A   A   -
    vmi_MIN       -     -   A   A   A   -
    vmi_IMAX      -     -   A   A   A   -
    vmi_MAX       -     -   A   A   A   -
    vmi_IMULSU    -     A   A   A   A   A
    vmi_IMULUS    -     A   A   A   A   A
    vmi_PMUL      -     -   A   A   A   -
    vmi_AESENC1   -     -   -   -   -   -
    vmi_AESENC1L  -     -   -   -   -   -
    vmi_AESDEC1   -     -   -   -   -   -
    vmi_AESDEC1L  -     -   -   -   -   -
    vmi_AESENC2   -     -   -   -   -   -
    vmi_AESENC2L  -     -   -   -   -   -
    vmi_AESDEC2   -     -   -   -   -   -
    vmi_AESDEC2L  -     -   -   -   -   -
    ------------ ----- --------------------

    ------ -----------------------------------------------------------------
    SYMBOL MEANING
    ------ -----------------------------------------------------------------
    -      Unused or unaffected
    U      Flag used (input)
    U0     Flag used only if shift/rotate is zero (input)
    A      Flag affected (output)
    0      Flag cleared (output)
    ------ -----------------------------------------------------------------
```

For signed saturating operations (with `SQ` suffix) the flags represent the computed value *before* saturation. It is therefore possible to tell whether signed saturation has occurred using the *overflow* flag.

For unsigned saturating operations (with `UQ` suffix) the flags represent the computed value *before* saturation. It is therefore possible to tell whether unsigned saturation has occurred using the *carry* flag.

3.10 Handling Exceptions

Integer divide and remainder operations can cause two kinds of exception: *integer overflow* (when the minimum negative integer is divided by -1) and *divide-by-zero*. It is possible to handle such exceptions using one of two distinct VMI function interfaces:

1. A model *arithmetic result handler*. This should be used when the operation could *either* cause a processor simulated exception *or* produce a known result.
2. A model *arithmetic exception handler*. This should be used when the operation can under some circumstances leave target registers of the divide or remainder unchanged. The interface is more general than the arithmetic result handler, but more complex: use the arithmetic result handler by preference if possible.

More information about each of these two function types is given below.

3.10.1 Arithmetic Result Handler

The arithmetic result handler is of type `vmiArithResultFn` and is specified using the `arithResultCB` field of the processor `vmiIASAttrs` structure. It is defined using the `VMI_ARITH_RESULT_FN` macro:

```
#define VMI_ARITH_RESULT_FN(_NAME) void _NAME( \
    vmiProcessorP    processor,     \
    vmiDivideInfoCP  divideInfo,    \
    vmiDivideResultP divideResults  \
```

```
)
typedef VMI_ARITH_RESULT_FN((*vmiArithResultFn));
```

The function is passed three arguments: the processor, a structure containing information about the inputs and type of the current operation (`divideInfo`) and a structure in which to return results (`divideResults`). The `vmiDivideInfo` type is defined in `vmiTypes.h` as follows:

```
typedef struct vmiDivideInfoS {
    Uns8  bits;         // bit size of operation (8, 16 ,32 or 64)
    Bool  isSigned;     // whether division is signed
    Uns64 dividendLSW;  // least-significant part of dividend
    Uns64 dividendMSW;  // most-significant part of dividend
    Uns64 divisor;      // divisor
} vmiDivideInfo;
```

This structure gives the operation size in bits, whether the division/remainder operation is signed or unsigned, and the divisor and dividend values. Because the VMI API supports division of up to 128 bit dividends, the value of the dividend is passed as two Uns64 values. There is no information about whether the faulting operation is a division or remainder operation: both results should be returned in the `vmiDivideResult` type (see below).

The `vmiDivideResult` type type is defined in `vmiTypes.h` as follows:

```
typedef struct vmiDivideResultS {
    Uns64 quotient;     // fill this with quotient
    Uns64 remainder;    // fill this with remainder
} vmiDivideResult;
```

The function may fill the `quotient` and `remainder` fields of the `divideResults` structure with appropriate values, given the function inputs in the `divideInfo` argument structure, or possibly use `vmirtSetPCException` to jump to a simulated exception vector. As an example, here is the arithmetic result handler from the OVP ARM model. This function usually returns a default result, but for ARMv7-R architecture processors can instead cause an exception (`armUndefined` eventually calls `vmirtSetPCException`):

```
VMI_ARITH_RESULT_FN(armArithExceptionCB) {

    armP arm = (armP)processor;

    if(divideInfo->divisor) {

        // integer overflow
        divideResults->quotient  = divideInfo->dividendLSW;
        divideResults->remainder = 0;

    } else if(!MMU_PRESENT(arm) && SYS_FIELD_ALT(arm, SCTLR, WXN_DZ)) {

        // divide-by-zero, ARMv7-R architecture - note that SCTLR.DZ is only
        // valid on ARMv7-R; on ARMv7-A, this bit is used for something
        // completely different (WXN) and there is no way to specify that an
        // undefined instruction exception should be taken
        armUndefined(arm, getPC(arm), 0, False);

    } else {

        // handle divide-by-zero, no exception
```

```
        divideResults->quotient  = 0;
        divideResults->remainder = 0;
    }
}
```

Note that there is no need to return a result if an exception is taken (the result is unused in this case). Divide-by-zero and integer-overflow cases can be distinguished by whether the divisor is zero or not.

3.10.2 Arithmetic Exception Handler

The arithmetic exception handler is of type `vmiArithExceptFn` and is specified using the `arithExceptCB` field of the processor `vmiIASAttrs` structure. It is defined using the `VMI_ARITH_EXCEPT_FN` macro:

```
#define VMI_ARITH_EXCEPT_FN(_NAME) vmiIntegerExceptionResult _NAME( \
    vmiProcessorP           processor,          \
    vmiNumericExceptionType exceptionType,      \
    vmiExceptionContext     exceptionContext    \
)
typedef VMI_ARITH_EXCEPT_FN((*vmiArithExceptFn));
```

The exact reason it is being called is indicated by the `exceptionType` argument:

```
typedef enum vmiIntegerExceptionTypeE {
    VMI_INTEGER_DIVIDE_BY_ZERO,
    VMI_INTEGER_OVERFLOW
} vmiIntegerExceptionType;
```

The handler may modify processor state to reflect the result of the faulting operation, or possibly use `vmirtSetPCException` to jump to a simulated exception vector. The return value of the handler is of type `vmiIntegerExceptionResult`:

```
typedef enum vmiIntegerExceptionResultE {
    VMI_NUMERIC_UNHANDLED,      // not handled
    VMI_NUMERIC_ABORT,          // handled, abort current instruction
    VMI_NUMERIC_CONTINUE,       // handled, continue current instruction
} vmiIntegerExceptionResult;
```

A result of `VMI_INTEGER_UNHANDLED` indicates that the exception condition is unexpected and simulation should terminate.

A result of `VMI_INTEGER_ABORT` indicates that the current simulated instruction should be terminated and simulation should resume with the *next* simulated instruction.

A result of `VMI_INTEGER_CONTINUE` indicates that simulation of the current instruction should be resumed after the faulting operation. In this case, all results of the faulting operation will be discarded and simulation will resume with the next VMI operation (which could be in the next simulated instruction or a later operation in the current simulated instruction).

As an example, here is the arithmetic result handler from the OVP ARC model. This function usually returns a default result, but may instead cause an exception for divide-by-zero (`arcTakeException0` eventually calls `vmirtSetPCException`):

```
VMI_ARITH_EXCEPT_FN(arcArithExceptionCB) {

    arcP arc = (arcP)processor;

    if(exceptionContext==VMI_EXCEPT_CXT_CALL) {

        // not expecting any arithmetic exceptions in calls from morphed code
        return VMI_INTEGER_UNHANDLED; // LCOV_EXCL_LINE

    } else switch(exceptionType) {

        case VMI_INTEGER_DIVIDE_BY_ZERO:
            // handle divide-by-zero
            if(AUX_FIELD(arc, status32, DZ)) {
                arcTakeException0(arc, EC_DivideByZero);
            } else if(arc->setFlags) {
                arc->aflags.VF = 1;
            }
            return VMI_INTEGER_ABORT;

        case VMI_INTEGER_OVERFLOW:
            // handle overflow (MIN_INT / -1)
            if(arc->setFlags) {
                arc->aflags.VF = 1;
            }
            return VMI_INTEGER_ABORT;

        default:
            // not expecting any other arithmetic exception types
            return VMI_INTEGER_UNHANDLED; // LCOV_EXCL_LINE
    }
}
```

Note that the simpler arithmetic exception result function cannot be used for the ARC model because when an exception occurs the target register is not updated.

3.11 vmimtGetSMPParentRegister

Prototype

```
vmiReg vmimtGetSMPParentRegister(vmiReg r, Uns32 level);
```

Description

The VMI interfaces allow the specification of SMP processor clusters. These clusters are implemented as a number of levels of container processor objects, each of which can contain further levels of container processors or leaf processors (which are actually simulated). As an example, the MIPS 1004K OVP processor model has these hierarchy levels:

1. a root level CMP processor object, containing:
2. a number of CPU processor objects, each containing:
3. a number of VPE (*virtual processing element*) processor objects, each containing:
4. a number of TC (*thread context*) objects, which are actually simulated.

The thread context objects each represent a microthread running on a VPE. Instructions on a TC can refer to registers:

1. local to the TC itself (for example, the GPRs);
2. at the VPE level (for example, most system control registers are implemented at the VPE level);
3. at the CPU level (a few system control registers are shared by all the VPEs).

It is possible for a TC to be moved to a different VPE on the same CPU at run time. For example, TC0 might start life bound to VPE0 on CPU0, but later on be dynamically rebound to VPE1 of CPU0 instead[2]. When this rebinding occurs, *any reference to a register at the VPE level must be sure to use the new VPE*.

Function `vmimtGetSMPParentRegister` takes as an argument a `vmiReg` representing a register in the processor and a `level` argument, indicating a parent level (level 0 is the leaf level processor itself, level 1 is its parent, level 2 is its grandparent, and so on). It returns a new `vmiReg` representing that register at the indicated parent level. The register description remains valid *even if the leaf level processor is subsequently relocated in the SMP cluster so that the parent at that level changes*.

Example

This example is taken from the MIPS processor model. In file `mips32Morph.c`, there is a function `mips32VPEReg` which returns the `vmiReg` description for a VPE-level register for the current TC. If the processor does not implement the multithreaded ASE, the TC and VPE levels are equivalent; otherwise, function `vmimtGetSMPParentRegister` is used to construct the correct `vmiReg` description for the parent VPE:

[2] See function `vmirtSetSMPParent` in the *VMI Run Time Function Reference*.

```
vmiReg mips32VPEReg(mips32P tc, vmiReg r) {
    mips32P vpe = VPE_FOR_TC(tc);
    return (tc==vpe) ? r : vmimtGetSMPParentRegister(r, 1);
}
```

This function is used, for example, in `mips32MorphCop0.c` to return the `vmiReg` description for a control register:

```
static vmiReg getCOP0Reg(mips32P tc, Uns32 reg, Uns32 sel) {

    mips32Cop0RegId id     = cop0RegInfo[reg][sel].id;
    vmiReg          result = MIPS32_CPU_REG(cop0.regs[id]);

    switch(getCOP0Level(reg, sel)) {

        case COP0_RL_CPU:
            return VMI_REG_DELTA(result, tc->cpuDelta);

        case COP0_RL_VPE:
            return mips32VPEReg(tc, result);

        default:
            return result;
    }
}
```

It is only necessary to use `vmimtGetSMPParentRegister` when *the parent at that level can change dynamically at run time*. In the case of the MIPS processor, a TC can be bound to a different VPE at run time, but that VPE must lie on the same CPU. Therefore, references to CPU level registers for a TC do not need to use `vmimtGetSMPParentRegister` but can use the macro `VMI_REG_DELTA` instead. This macro constructs a `vmiReg` value referencing a register at a constant offset from the current processor.

Notes and Restrictions

None.

3.12 vmimtMoveRC

Prototype

```
void vmimtMoveRC(
    Uns32  bits,
    vmiReg rd,
    Uns64  c
);
```

Description

Emit code to move a constant value `c` into target register `rd` within the processor. The register has size `bits` within the processor structure.

Example

The OVP OR1K model uses this function to implement the MOVHI instruction:

```
// Emit code for a movhi instruction
static OR1K_MORPH_FN(morphMOVHI) {

    vmiReg rd = getGPR(state->info.r1);
    Uns32  c  = state->info.c;

    vmimtMoveRC(OR1K_BITS, rd, c<<16);
}
```

Notes and Restrictions

1. `bits` can be any multiple of 8. If `bits` is greater than 64, the given constant is replicated to fill the target register.
2. For target registers less than 64 bits wide, the constant must be either a zero or sign extended pattern that can be represented in that number of bits.

3.13 vmimtMoveRSimPC

Prototype

```
void vmimtMoveRSimPC(
    Uns32  bits,
    vmiReg rd
);
```

Description

Emit code to move the current simulated program counter into target register `rd` within the processor. The register has size `bits` within the processor structure.

If a processor model does not use physically-mapped code dictionaries, then this is equivalent to using `vmimtMoveRC`, specifying the current program counter as the constant argument. However, when processor models do use physically-mapped code dictionaries, `vmimtMoveRSimPC` **must** be used to obtain the current simulated address, because the same JIT-compiled code block can be mapped at *different* simulated virtual addresses.

See the description of `vmirtAliasMemoryVM` in the *VMI Run Time Function Reference* and also the *Imperas Processor Modeling Guide* for more information about physically-mapped code dictionaries.

Example

The OVP MIPS model uses this function when calculating link addresses:

```
// Emit code to set link address to thisPC+8
static void emitSetLinkAddress(vmiReg rl) {

    Uns32 bits = MIPS32_GPR_BITS;

    vmimtMoveRSimPC(bits, rl);
    vmimtBinopRC(bits, vmi_ADD, rl, 8, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

3.14 vmimtMoveRR

Prototype

```
void vmimtMoveRR(
    Uns32  bits,
    vmiReg rd,
    vmiReg ra
);
```

Description

Emit code to move from source register `ra` to target register `rd` within the processor. Both registers are of size `bits` within the processor structure.

Example

The OVP RISC-V model uses this function to define register-to-register moves:

```
static RISCV_MORPH_FN(emitMoveRR) {

    vmiReg rd   = getReg(state, 0);
    vmiReg rs   = getReg(state, 1);
    Uns32  bits = getRegBits(state, 0);

    vmimtMoveRR(bits, rd, rs);

    writeReg(state, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

3.15 vmimtMoveExtendRR

Prototype

```
void vmimtMoveExtendRR(
    Uns32  destBits,
    vmiReg rd,
    Uns32  srcBits,
    vmiReg ra,
    Bool   signExtend
);
```

Description

Emit code to move from source register `ra` to target register `rd` within the processor. The destination register is of size `destBits` and the source register of size `srcBits` (`destBits` must be equal to or larger than `srcBits`). If source and destination sizes are unequal, then the source will be zero-extended (if `signExtend` is `False`) or sign-extended (if `signExtend` is `True`) to the full destination size.

If source and destination sizes match, this function is exactly equivalent to `vmimtMoveRR`.

Example

The OVP RISC-V model uses this function to calculate an exclusive access tag address:

```
static void generateEATag(riscvMorphStateP state, vmiReg rtag, vmiReg ra) {

    Uns32 bits   = getEABits(state);
    Uns32 raBits = getModeBits(state);

    vmimtMoveExtendRR(bits, rtag, raBits, ra, 0);
    vmimtBinopRC(bits, vmi_AND, rtag, state->riscv->exclusiveTagMask, 0);
}
```

Notes and Restrictions

1. `destBits` and `srcBits` must be 8, 16, 32, 64 or 128.
2. `destBits` must be equal to or greater than `srcBits`.

3.16 vmimtCondMoveRRR

Prototype

```
void vmimtCondMoveRRR(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    vmiReg ra,
    vmiReg rb
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move from source register `ra` to target register `rd` within the processor. Otherwise, the emitted code will move source register `rb` to target register `rd` within the processor. All registers are of size `bits` within the processor structure.

Example

The OVP RISC-V model uses this function to implement an atomic conditional select instruction:

```
static AMO_FN(emitAMOCmpopRRRCB) {

    vmiReg tf = getTmp(state, 2);

    vmimtCompareRR(bits, state->attrs->cond, ra, rb, tf);
    vmimtCondMoveRRR(bits, tf, True, rd, ra, rb);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

3.17 vmimtCondMoveRRC

Prototype

```
void vmimtCondMoveRRC(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    vmiReg ra,
    Uns64  c
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move from source register `ra` to target register `rd` within the processor. Otherwise, the emitted code will move constant `c` to target register `rd` within the processor. All registers are of size `bits` within the processor structure.

Example

The OVP RISC-V model uses this function to convert input floating point register values that are not NaN-boxed to a QNaN:

```
static vmiReg getRegFS(riscvMorphStateP state, Uns32 argNum) {

    riscvRegDesc r      = state->info.r[argNum];
    Uns32        bits   = getRegBits(state, argNum);
    vmiReg       result = getReg(state, argNum);

    if(isFReg(r)) {

        Uns32 archBits = riscvGetFlenArch(state->riscv);
        Uns32 fprMask  = getRegMask(r);

        if((archBits>bits) && !(state->blockState->fpNaNBoxMask&fprMask)) {

            // use temporary corresponding to the input argument
            vmiReg tmp   = getTmp(state, argNum);
            vmiReg upper = VMI_REG_DELTA(result,bits/8);

            // is the upper half all ones?
            vmimtCompareRC(bits, vmi_COND_EQ, upper, -1, tmp);

            // seed the apparent value, depending on whether the source is
            // correctly NaN-boxed
            vmimtCondMoveRRC(bits, tmp, True, tmp, result, FP32_DEFAULT_QNAN);

            // use the temporary as a source
            result = tmp;
        }
    }

    return result;
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

3.18 vmimtCondMoveRCR

Prototype

```
void vmimtCondMoveRCR(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    Uns64  c,
    vmiReg rb
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move constant `c` to target register `rd` within the processor. Otherwise, the emitted code will move source register `rb` to target register `rd` within the processor. All registers are of size `bits` within the processor structure.

Example

The OVP ARC model uses this function to implement a min/max operation taking a constant and register as arguments:

```
static void emitMinmaxopRCRInt(arcMorphStateP state, Int32 c1, vmiReg rs1) {

    vmiReg       rd          = GET_RD(state, rd);
    vmiFlagsCP   flags       = getFlagsOrNull(state);
    vmiCondition minmaxCCond = state->attrs->minmaxCCond;
    vmiReg       tf          = flags ? ARC_CF : getTemp(state);

    // generate flags if required
    if(flags) {
        vmimtBinopRCR(ARC_GPR_BITS, vmi_CMP, VMI_NOREG, c1, rs1, flags);
    }

    // generate the selection condition
    vmimtCompareCR(ARC_GPR_BITS, minmaxCCond, c1, rs1, tf);

    // do the conditional move
    vmimtCondMoveRCR(ARC_GPR_BITS, tf, False, rd, c1, rs1);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

3.19 vmimtCondMoveRCC

Prototype

```
void vmimtCondMoveRCC(
    Uns32  bits,
    vmiReg flag,
    Bool   select1,
    vmiReg rd,
    Uns64  c1,
    Uns64  c2
);
```

Description

Emit code to compare the (8-bit) register `flag` in the processor structure with `select1`. If `flag` equals select1, the emitted code will move constant `c1` to target register `rd` within the processor. Otherwise, the emitted code will move constant `c2` to target register `rd` within the processor. Register `rd` is of size `bits` within the processor structure.

Example

The OVP ARM model uses this function to implement some per-element vector compare instructions:

```
static SIMD_EL_OP_FN(simdVCmpSelBool_F) {

    vmiFPRelation cond      = state->attrs->fpRelation;
    Bool          allowQNaN = state->attrs->allowQNaN;
    Uns64         ones      = allOnes(resultSize);
    vmiReg        relation  = getTemp(state, 32);
    vmiFlags      flags     = getZFFlags(VMI_REG_DELTA(relation, 1));

    // Compare the floating point operands, getting vmiFPRelation result in the
    // register relation
    vmimtFCompareRR(bytesToFType(opSize/8), relation, r2, r3, allowQNaN);

    // Move zeros or ones to result depending on whether any cond bits are set in
    //  relation
    vmimtBinopRRC(8, vmi_AND, VMI_NOREG, relation, cond, &flags);
    vmimtCondMoveRCC(resultSize, flags.f[vmi_ZF], False, result, ones, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32, 64 or 128.

3.20 vmimtUnopR

Prototype

```
void vmimtUnopR(
    Uns32      bits,
    vmiUnop    op,
    vmiReg     rd,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a unary operation on register `rd` in the processor structure, writing the result back to the same register. The argument `bits` gives the bit width for the operation.

Argument `op` is the unary operation to perform. Available unary operations are defined in `vmiTypes.h`: see *Simulated Register Specification* Using `vmiReg`
Most functions in this API require use of the `vmiReg` type to specify the location of source and target registers in a structure representing a simulated processor. A short introduction to usage of this type is given here; for a more detailed description, refer to the *Imperas Processor Modeling Guide.*

As an example, the OVP OR1K processor is represented using a structure of type or1k, defined as follows:

```
#define OR1K_REGS 32

typedef struct or1kS {

    Bool          carryFlag;        // carry flag
    Bool          overflowFlag;     // overflow flag
    Bool          branchFlag;       // branch flag

    Uns32         regs[OR1K_REGS];  // basic registers

    . . . fields omitted for clarity . . .

} or1k, *or1kP;
```

Here, for example, the `regs` member holds the value of each of the 32 GPRs. The location of a register (for example, a GPR) is specified to the simulator using the `vmiReg` type, defined in file `vmiTypes.h.` A `vmiReg` structure can be created for any field in a processor structure using the `VMI_CPU_REG` macro, which takes a type pointer and a field name argument. Typically, the processor header files will contain further macros that encapsulate usage of the `VMI_CPU_REG` macro appropriately for that processor: for example, the OVP OR1K model contains these macro definitions:

```
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY              OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW           OR1K_CPU_REG(overflowFlag)
#define OR1K_BRANCH             OR1K_CPU_REG(branchFlag)
```

As an example, this code could now be used to specify the location of the OR1K `branchFlag` register to a morph-time API function:

```
vmiReg bf = OR1K_BRANCH;
```

Typically, usage of registers such as GPRs is encapsulated by sugar routines that handle special values. In the case of the OR1K processor, GPR 0 is always zero and unwritable. Therefore, the following sugar function is used to return an appropriate `vmiReg` for an operation, given a GPR index:

```
static vmiReg getGPR(Uns32 r) {
    return r ? OR1K_REG(r) : VMI_NOREG;
}
```

For conciseness and clarity, Examples listed in this manual will typically refer to `vmiReg` structures without giving details of the processor structure that contains those registers. Unary Operation Types for more information about this.

If the flags argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP PowerPC model uses this function to implement negate instructions:

```
PPC32_MORPH_FN(morphSE_NEG_R1) {

    Uns8 RX = state->info.RX;
    vmiReg GPR_RX = PPC32_GPR_WR(RX);

    vmimtUnopR(PPC32_GPR_BITS, vmi_NEG, GPR_RX, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. The `vmi_SWP` unary operation entirely reverses the byte order of the argument. For example the 64-bit value `0x0102030405060708` becomes `0x0807060504030201`.

3.21 vmimtUnopRR

Prototype

```
void vmimtUnopRR(
    Uns32      bits,
    vmiUnop    op,
    vmiReg     rd,
    vmiReg     ra,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a unary operation on register `ra` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the unary operation to perform. Available unary operations are defined in `vmiTypes.h`: see *Simulated Register Specification* Using `vmiReg`

Most functions in this API require use of the `vmiReg` type to specify the location of source and target registers in a structure representing a simulated processor. A short introduction to usage of this type is given here; for a more detailed description, refer to the *Imperas Processor Modeling Guide.*

As an example, the OVP OR1K processor is represented using a structure of type or1k, defined as follows:

```
#define OR1K_REGS 32

typedef struct or1kS {

    Bool          carryFlag;        // carry flag
    Bool          overflowFlag;     // overflow flag
    Bool          branchFlag;       // branch flag

    Uns32         regs[OR1K_REGS];  // basic registers

    . . . fields omitted for clarity . . .

} or1k, *or1kP;
```

Here, for example, the `regs` member holds the value of each of the 32 GPRs. The location of a register (for example, a GPR) is specified to the simulator using the `vmiReg` type, defined in file `vmiTypes.h.` A `vmiReg` structure can be created for any field in a processor structure using the `VMI_CPU_REG` macro, which takes a type pointer and a field name argument. Typically, the processor header files will contain further macros that encapsulate usage of the `VMI_CPU_REG` macro appropriately for that processor: for example, the OVP OR1K model contains these macro definitions:

```
#define OR1K_CPU_REG(_F)        VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)            OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY              OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW           OR1K_CPU_REG(overflowFlag)
#define OR1K_BRANCH             OR1K_CPU_REG(branchFlag)
```

As an example, this code could now be used to specify the location of the OR1K `branchFlag` register to a morph-time API function:

```
vmiReg bf = OR1K_BRANCH;
```

Typically, usage of registers such as GPRs is encapsulated by sugar routines that handle special values. In the case of the OR1K processor, GPR 0 is always zero and unwritable. Therefore, the following sugar function is used to return an appropriate `vmiReg` for an operation, given a GPR index:

```
static vmiReg getGPR(Uns32 r) {
    return r ? OR1K_REG(r) : VMI_NOREG;
}
```

For conciseness and clarity, Examples listed in this manual will typically refer to `vmiReg` structures without giving details of the processor structure that contains those registers. Unary Operation Types for more information about this.

If the flags argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

If `rd` and `ra` are the same, this is equivalent to `vmimtUnopR`.

Example

The OVP ARM model uses this function to implement some per-element vector compare instructions:

```
static SIMD_EL_OP_FN(simdVCmp0) {

    vmiCondition cond = state->attrs->cond;

    // do the indicated comparison test on the operands
    vmimtCompareRC(opSize, cond, r2, 0, result);

    // extend to operand size
    vmimtMoveExtendRR(opSize, result, 8, result, False);

    // negate to fill with zeros or ones
    vmimtUnopRR(opSize, vmi_NEG, result, result, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. The `vmi_SWP` unary operation entirely reverses the byte order of the argument. For example the 64-bit value `0x0102030405060708` becomes `0x0807060504030201`.
3. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.22 vmimtUnopRC

Prototype

```
void vmimtUnopRC(
    Uns32      bits,
    vmiUnop    op,
    vmiReg     rd,
    Uns64      c,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a unary operation on constant `c`, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the unary operation to perform. Available unary operations are defined in `vmiTypes.h`: see *Simulated Register Specification* Using `vmiReg`

Most functions in this API require use of the `vmiReg` type to specify the location of source and target registers in a structure representing a simulated processor. A short introduction to usage of this type is given here; for a more detailed description, refer to the *Imperas Processor Modeling Guide.*

As an example, the OVP OR1K processor is represented using a structure of type or1k, defined as follows:

```
#define OR1K_REGS 32

typedef struct or1kS {

    Bool          carryFlag;        // carry flag
    Bool          overflowFlag;     // overflow flag
    Bool          branchFlag;       // branch flag

    Uns32         regs[OR1K_REGS];  // basic registers

    . . . fields omitted for clarity . . .

} or1k, *or1kP;
```

Here, for example, the `regs` member holds the value of each of the 32 GPRs. The location of a register (for example, a GPR) is specified to the simulator using the `vmiReg` type, defined in file `vmiTypes.h.` A `vmiReg` structure can be created for any field in a processor structure using the `VMI_CPU_REG` macro, which takes a type pointer and a field name argument. Typically, the processor header files will contain further macros that encapsulate usage of the `VMI_CPU_REG` macro appropriately for that processor: for example, the OVP OR1K model contains these macro definitions:

```
#define OR1K_CPU_REG(_F)         VMI_CPU_REG(or1kP, _F)
#define OR1K_REG(_R)             OR1K_CPU_REG(regs[_R])
#define OR1K_CARRY               OR1K_CPU_REG(carryFlag)
#define OR1K_OVERFLOW            OR1K_CPU_REG(overflowFlag)
#define OR1K_BRANCH              OR1K_CPU_REG(branchFlag)
```

As an example, this code could now be used to specify the location of the OR1K `branchFlag` register to a morph-time API function:

```
vmiReg bf = OR1K_BRANCH;
```

Typically, usage of registers such as GPRs is encapsulated by sugar routines that handle special values. In the case of the OR1K processor, GPR 0 is always zero and unwritable. Therefore, the following sugar function is used to return an appropriate `vmiReg` for an operation, given a GPR index:

```
static vmiReg getGPR(Uns32 r) {
    return r ? OR1K_REG(r) : VMI_NOREG;
}
```

For conciseness and clarity, Examples listed in this manual will typically refer to `vmiReg` structures without giving details of the processor structure that contains those registers. Unary Operation Types for more information about this.

If the flags argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP PowerPC model uses this function to implement some load-immediate instructions:

```
PPC32_MORPH_FN(morphLI_D2_1) {

    Uns8 RT = state->info.RT;
    vmiReg GPR_RT = PPC32_GPR_WR(RT);
    Int16 SI = state->info.SI;

    vmimtUnopRC(PPC32_GPR_BITS, vmi_ADD, GPR_RT, SI, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. The `vmi_SWP` unary operation entirely reverses the byte order of the argument. For example the 64-bit value `0x0102030405060708` becomes `0x0807060504030201`.
3. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.23 vmimtBinopRR

Prototype

```
void vmimtBinopRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    vmiReg     ra,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on registers `rd` and `ra` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP ARM model uses this function to implement some instructions that set a cumulative saturation flag:

```
static void emitOpSetQ(
    armMorphStateP state,
    Uns32          bits,
    vmiBinop       op,
    vmiReg         rd,
    vmiReg         rs1,
    vmiReg         rs2
) {
    vmiReg   tf    = getTemp(state, 32);
    vmiFlags flags = getOFFlags(tf);

    // do the operation, setting flags
    vmimtBinopRRR(bits, op, rd, rs1, rs2, &flags);

    // set the sticky Q flag of there was overflow
    vmimtBinopRR(8, vmi_OR, ARM_QF, tf, 0);
}
```

Notes and Restrictions

1. The `bits` argument must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
3. For shift/rotate operations (`vmi_ROL`, `vmi_ROR`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` and `vmi_SAR`) the shift/rotate amount b is by default masked using

bits-1 before use. For example, if bits is 32 then the shift/rotate amount will be masked to the range 0..31 before use. This default behavior can be overridden by vmimtSetShiftMask.

4. rd may be VMI_NOREG, in which case the operation result is discarded. This is useful if only flag values are required.

3.24 vmimtBinopRRR

Prototype

```
void vmimtBinopRRR(
    Uns32       bits,
    vmiBinop    op,
    vmiReg      rd,
    vmiReg      ra,
    vmiReg      rb,
    vmiFlagsCP  flags
);
```

Description

Emit code to perform a binary operation on registers `ra` and `rb` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

If `rd` and `ra` are equal, this is equivalent to `vmimtBinopRR`.

Example

The OVP RISC-V model uses this function to implement generic binops:

```
static RISCV_MORPH_FN(emitBinopRRR) {

    vmiReg rd   = getReg(state, 0);
    vmiReg rs1  = getReg(state, 1);
    vmiReg rs2  = getReg(state, 2);
    Uns32  bits = getRegBits(state, 0);

    vmimtBinopRRR(bits, state->attrs->binop, rd, rs1, rs2, 0);

    writeReg(state, 0);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
3. For shift/rotate operations (`vmi_ROL`, `vmi_ROR`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` and `vmi_SAR`) the shift/rotate amount `b` is by default masked using `bits-1` before use. For example, if `bits` is 32 then the shift/rotate amount will be masked to the range 0..31 before use. This default behavior can be overridden by `vmimtSetShiftMask`.

4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.25 vmimtBinopRC

Prototype

```
void vmimtBinopRC(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    Uns64      c,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on register `rd` in the processor structure and constant `c`, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP RISC-V model uses this function to calculate an exclusive access tag address:

```
static void generateEATag(riscvMorphStateP state, vmiReg rtag, vmiReg ra) {

    Uns32 bits   = getEABits(state);
    Uns32 raBits = getModeBits(state);

    vmimtMoveExtendRR(bits, rtag, raBits, ra, 0);
    vmimtBinopRC(bits, vmi_AND, rtag, state->riscv->exclusiveTagMask, 0);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c` are silently discarded.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.26 vmimtBinopRCR

Prototype

```
void vmimtBinopRCR(
    Uns32       bits,
    vmiBinop    op,
    vmiReg      rd,
    Uns64       c,
    vmiReg      rb,
    vmiFlagsCP  flags
);
```

Description

Emit code to perform a binary operation on constant `c` and register `rb` in the processor structure, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP ARC model uses this function to implement a logical operation taking a constant and register as arguments:

```
static void emitLogopRCRInt(arcMorphStateP state, Uns32 c1, vmiReg rs1) {

    vmiReg     rd    = GET_RD(state, rd);
    vmiBinop   op    = state->attrs->binop;
    vmiFlagsCP flags = getFlagsOrNull(state);

    // emit code to create an appropriate mask in temp
    vmiReg temp = emitMakeBitMask(state, rs1);

    vmimtBinopRCR(ARC_GPR_BITS, op, rd, c1, temp, flags);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64; for AES operations, `bits` must be 64.
2. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
3. For shift/rotate operations (`vmi_ROL`, `vmi_ROR`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` and `vmi_SAR`) the shift/rotate amount `b` is by default masked using `bits-1` before use. For example, if `bits` is 32 then the shift/rotate amount will be masked to the range 0..31 before use. This default behavior can be overridden by `vmimtSetShiftMask`.

4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.27 vmimtBinopRCC

Prototype

```
void vmimtBinopRCC(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rd,
    Uns64      c1,
    Uns64      c2,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on constants `c1` and `c2`, writing the result to register `rd`.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP ARC model uses this function to implement a logical operation taking two constants as arguments:

```
static void emitLogopRCCInt(arcMorphStateP state, Uns32 c1, Uns32 c2) {

    vmiReg     rd    = GET_RD(state, rd);
    vmiBinop   op    = state->attrs->binop;
    vmiFlagsCP flags = getFlagsOrNull(state);

    // invert the constant and subtract 1 if required
    c2 = makeBitMask(state, c2);

    vmimtBinopRCC(ARC_GPR_BITS, op, rd, c1, c2, flags);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c1` and `c2` are silently discarded prior to use.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.28 vmimtBinopRRC

Prototype

```
void vmimtBinopRRC(
    Uns32       bits,
    vmiBinop    op,
    vmiReg      rd,
    vmiReg      ra,
    Uns64       c,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a binary operation on register `ra` in the processor structure and constant `c`, writing the result to register `rd`. Argument `bits` gives the bit width for the operation.

Argument `op` is the binary operation to perform. Available binary operations are defined in `vmiTypes.h`: see *Binary Operation Types* for more information about this.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP ARC model uses this function to implement a logical operation taking a register and constant as arguments:

```
static void emitLogopRRCInt(arcMorphStateP state, vmiReg rs1, Uns32 c1) {

    vmiReg     rd    = GET_RD(state, rd);
    vmiBinop   op    = state->attrs->binop;
    vmiFlagsCP flags = getFlagsOrNull(state);

    // invert the constant and subtract 1 if required
    c1 = makeBitMask(state, c1);

    vmimtBinopRRC(ARC_GPR_BITS, op, rd, rs1, c1, flags);
}
```

Notes and Restrictions

1. Argument `bits` must be 8, 16, 32 or 64.
2. For target registers less than 64 bits wide, the unused most significant bits of `c` are silently discarded prior to use.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).
4. `rd` may be `VMI_NOREG`, in which case the operation result is discarded. This is useful if only flag values are required.

3.29 vmimtMulopRRR

Prototype

```
void vmimtMulopRRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rdh,
    vmiReg     rdl,
    vmiReg     ra,
    vmiReg     rb,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a multiply operation on registers `ra` and `rb` in the processor structure. Argument `bits` gives the bit width of these two registers. The result of the multiply has size `bits*2`; the most significant part of the result (size `bits`) is assigned to processor register `rdh`, and the least significant part of the result (also of size `bits`) is assigned to processor register `rdl`.

Either `rdh` or `rdl` may have the special value `VMI_NOREG`; this indicates that this part of the result is to be discarded (not saved in a processor register). If `rdh` is `VMI_NOREG`, this function is equivalent to `vmiBinopRRR`.

Available binary operations are this subset from the `vmiBinop` type defined in `vmiTypes.h`:

```
typedef enum {
    vmi_IMUL,       // d <- a * b (signed)
    vmi_MUL,        // d <- a * b (unsigned)
    vmi_IMULSU,     // d <- a (signed) * b (unsigned)
    vmi_IMULUS,     // d <- a (unsigned) * b (signed)
    vmi_PMUL,       // d <- a * b (carryless)
} vmiBinop;
```

The first four operations are normal multiplications taking every combination of signed and unsigned arguments. Operation `vmi_PMUL` is a carryless multiplication in which terms are combined using exclusive-or instead of by addition; this is often required in cryptographic instructions.

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP RISC-V model uses this function to implement a multiply operation that selects the upper half of the result:

```
static RISCV_MORPH_FN(emitMulopHRRR) {

    vmiReg rd   = getReg(state, 0);
    vmiReg rs1  = getReg(state, 1);
```

```
    vmiReg rs2  = getReg(state, 2);
    Uns32  bits = getRegBits(state, 0);

    vmimtMulopRRR(bits, state->attrs->binop, rd, VMI_NOREG, rs1, rs2, 0);

    writeReg(state, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. Operations other than the subset listed above not supported by this function.

3.30 vmimtDivopRRR

Prototype

```
void vmimtDivopRRR(
    Uns32      bits,
    vmiBinop   op,
    vmiReg     rdd,
    vmiReg     rdr,
    vmiReg     rah,
    vmiReg     ral,
    vmiReg     rb,
    vmiFlagsCP flags
);
```

Description

Emit code to perform a divide operation, producing both result and remainder. The dividend is of size `bits*2` and is constructed from the register pair `rah:ral`. The divisor is in register `rb`. The result of the division is assigned to processor register `rdd`. The remainder is assigned to processor register `rdr`.

Either `rdd` or `rdr` may have the special value `VMI_NOREG`; this indicates that the result or remainder (as appropriate) is to be discarded (not saved in a processor register).

Available binary operations are this subset from the `vmiBinop` type defined in `vmiTypes.h`:

```
typedef enum {
    vmi_IDIV,           // d <- a / b (signed)
    vmi_DIV,            // d <- a / b (unsigned)
} vmiBinop;
```

If the `flags` argument is non-null, it defines how any flags should be handled by this operation: see *Handling Instruction Flags* for more information about this.

Example

The OVP MIPS model uses this function to implement double-width divide operations:

```
static void emitDivRR(mipsInstructionInfoP info, vmiBinop op) {

    Uns32        bits    = getOpBits(info);
    vmiReg       lo      = MIPS_REG_LO(0);
    vmiReg       hi      = MIPS_REG_HI(0);
    vmiReg       rs      = getR1(info);
    vmiReg       rt      = getR2(info);
    mipsIDivType divType = (op == vmi_IDIV) ? MIPS_IDIV_DIV : MIPS_IDIV_DIVU;

    // save operation size (in bytes) in divType
    divType |= (bits/8) << MIPS_IDIV_SIZE_SHIFT;

    // be ready for exceptions (divide by zero or overflow)
    vmimtMoveRC(8, MIPS_TMP_DIV_TYPE, divType);
    vmimtMoveRR(bits, MIPS_TMP_DIVIDEND, rs);

    vmimtDivopRRR(bits, op, lo, hi, VMI_NOREG, rs, rt, 0);

    // sign-extend to 64 bits if required
```

```
    MIPS_MT_SIGN_EXTEND(lo, bits);
    MIPS_MT_SIGN_EXTEND(hi, bits);

    // Clear divide type setting
    vmimtMoveRC(8, MIPS_TMP_DIV_TYPE, MIPS_IDIV_NONE);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. Operations other than the subset listed above not supported by this function.
3. Arithmetic exceptions can be generated by some operations (for example, integer divide by zero or integer overflow). A handler for these arithmetic exceptions can be supplied if required (defined with the `VMI_ARITH_EXCEPT_FN` macro, installed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).

3.31 vmimtCompareRR

Prototype

```
void vmimtCompareRR(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare the two processor registers `ra` and `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by subtracting the second argument from the first using twos complement arithmetic and discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                         (OF==1)
    vmi_COND_NO  = 1,   // no overflow                      (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)                 (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)             (CF==0)
    vmi_COND_Z   = 4,   // zero                             (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)            (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                         (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)    (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)        (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)    (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                         (SF==1)
    vmi_COND_NS  = 9,   // not negative                     (SF==0)
    vmi_COND_P   = 10,  // parity even                      (PF==1)
    vmi_COND_NP  = 11,  // not parity even                  (PF==0)
    vmi_COND_L   = 12,  // less (signed)                    (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)                (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)           (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)       (ZF==0 && SF==OF)
} vmiCondition;
```

Example

The OVP RISC-V model uses this function to implement register-register compare operations:

```
static RISCV_MORPH_FN(emitCmpopRRR) {

    vmiReg rd   = getReg(state, 0);
    vmiReg rs1  = getReg(state, 1);
    vmiReg rs2  = getReg(state, 2);
    Uns32  bits = getRegBits(state, 0);

    vmimtCompareRR(bits, state->attrs->cond, rs1, rs2, rd);

    writeRegSize(state, 0, 8);
}
```


Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

3.32 vmimtCompareCR

Prototype

```
void vmimtCompareCR(
    Uns32        bits,
    vmiCondition cond,
    Uns64        c,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare constant `c` and register `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by subtracting the second argument from the first using twos complement arithmetic and discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                         (OF==1)
    vmi_COND_NO  = 1,   // no overflow                      (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)                 (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)             (CF==0)
    vmi_COND_Z   = 4,   // zero                             (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)            (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                         (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)    (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)        (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)    (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                         (SF==1)
    vmi_COND_NS  = 9,   // not negative                     (SF==0)
    vmi_COND_P   = 10,  // parity even                      (PF==1)
    vmi_COND_NP  = 11,  // not parity even                  (PF==0)
    vmi_COND_L   = 12,  // less (signed)                    (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)                (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)           (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)       (ZF==0 && SF==OF)
} vmiCondition;
```

Example

The OVP ARC model uses this function to modify results of some binary operations taking a constant and register as arguments:

```
static void emitBinopRCRInt(arcMorphStateP state, Uns32 c1, vmiReg rs1) {

    Uns32        bits  = ARC_GPR_BITS;
    vmiReg       rd    = GET_RD(state, rd);
    vmiBinop     op    = state->attrs->binop;
    vmiFlagsCP   flags = getFlagsOrCIn(state);
    vmiCondition cond;

    // emit boolean indicating whether flags should be updated
    emitRequireSetFlags(state, op);

    // invert, scale and mask the variable second argument if required
    rs1 = emitInvertScaleMaskR(state, rs1);

    // do the operation
```

```
    vmimtBinopRCR(bits, op, rd, c1, rs1, flags);

    // handle any result condition
    if(getResultCondition(state, rd, &cond)) {
        vmimtCompareCR(bits, cond, c1, rs1, rd);
        emitExtendFlag(state, rd);
    }
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

3.33 vmimtCompareRC

Prototype

```
void vmimtCompareRC(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    Uns64        c,
    vmiReg       flag
);
```

Description

Emit code to compare the processor register `ra` and constant `c` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by subtracting the second argument from the first using twos complement arithmetic and discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h`:

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                          (OF==1)
    vmi_COND_NO  = 1,   // no overflow                       (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)                  (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)              (CF==0)
    vmi_COND_Z   = 4,   // zero                              (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)             (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                          (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)     (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)         (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)     (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                          (SF==1)
    vmi_COND_NS  = 9,   // not negative                      (SF==0)
    vmi_COND_P   = 10,  // parity even                       (PF==1)
    vmi_COND_NP  = 11,  // not parity even                   (PF==0)
    vmi_COND_L   = 12,  // less (signed)                     (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)                 (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)            (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)        (ZF==0 && SF==OF)
} vmiCondition;
```

Example

The OVP RISC-V model uses this function to implement register-constant compare operations:

```
static RISCV_MORPH_FN(emitCmpopRRC) {

    vmiReg rd   = getReg(state, 0);
    vmiReg rs1  = getReg(state, 1);
    Uns32  bits = getRegBits(state, 0);
    Uns64  c    = state->info.c;

    vmimtCompareRC(bits, state->attrs->cond, rs1, c, rd);

    writeRegSize(state, 0, 8);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

3.34 vmimtTestRR

Prototype

```
void vmimtTestRR(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare the two processor registers `ra` and `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by a bitwise-and of the two arguments, discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h` (but note that CF and OF are always set to zero by this comparison):

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                         (OF==1)
    vmi_COND_NO  = 1,   // no overflow                      (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)                 (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)             (CF==0)
    vmi_COND_Z   = 4,   // zero                             (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)            (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                         (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)    (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)        (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)    (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                         (SF==1)
    vmi_COND_NS  = 9,   // not negative                     (SF==0)
    vmi_COND_P   = 10,  // parity even                      (PF==1)
    vmi_COND_NP  = 11,  // not parity even                  (PF==0)
    vmi_COND_L   = 12,  // less (signed)                    (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)                (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)           (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)       (ZF==0 && SF==OF)
} vmiCondition;
```

Example

The OVP ARM model uses this function to implement some per-element vector compare instructions:

```
static SIMD_EL_OP_FN(simdVTst) {

    // set result if bitwise-and of operands is non-zero
    vmimtTestRR(opSize, vmi_COND_NZ, r2, r3, result);

    // extend to operand size
    vmimtMoveExtendRR(opSize, result, 8, result, False);

    // negate to fill with zeros or ones
    vmimtUnopRR(opSize, vmi_NEG, result, result, 0);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

3.35 vmimtTestCR

Prototype

```
void vmimtTestCR(
    Uns32        bits,
    vmiCondition cond,
    Uns64        c,
    vmiReg       rb,
    vmiReg       flag
);
```

Description

Emit code to compare constant `c` and register `rb` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by a bitwise-and of the two arguments, discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h` (but note that CF and OF are always set to zero by this comparison):

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                        (OF==1)
    vmi_COND_NO  = 1,   // no overflow                     (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)                (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)            (CF==0)
    vmi_COND_Z   = 4,   // zero                            (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)           (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                        (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)   (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)       (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)   (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                        (SF==1)
    vmi_COND_NS  = 9,   // not negative                    (SF==0)
    vmi_COND_P   = 10,  // parity even                     (PF==1)
    vmi_COND_NP  = 11,  // not parity even                 (PF==0)
    vmi_COND_L   = 12,  // less (signed)                   (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)               (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)          (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)      (ZF==0 && SF==OF)
} vmiCondition;
```

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

3.36 vmimtTestRC

Prototype

```
void vmimtTestRC(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    Uns64        c,
    vmiReg       flag
);
```

Description

Emit code to compare the processor register `ra` and constant `c` of size `bits`. The comparison to perform is indicated by `cond` and is implemented by a bitwise-and of the two arguments, discarding the result. If the comparison is true assign 1 to the 8-bit processor register `flag`; otherwise, assign 0 to this register.

Available comparison operations are defined in `vmiTypes.h` (but note that CF and OF are always set to zero by this comparison):

```
typedef enum {
    vmi_COND_O   = 0,   // overflow                       (OF==1)
    vmi_COND_NO  = 1,   // no overflow                    (OF==0)
    vmi_COND_B   = 2,   // below (unsigned)               (CF==1)
    vmi_COND_NB  = 3,   // not below (unsigned)           (CF==0)
    vmi_COND_Z   = 4,   // zero                           (ZF==1)
    vmi_COND_EQ  = 4,   // equal (alias of zero)          (ZF==1)
    vmi_COND_NZ  = 5,   // not zero                       (ZF==0)
    vmi_COND_NE  = 5,   // not equal (alias of not zero)  (ZF==0)
    vmi_COND_BE  = 6,   // below or equal (unsigned)      (CF==1 || ZF==1)
    vmi_COND_NBE = 7,   // not below or equal (unsigned)  (CF==0 && ZF==0)
    vmi_COND_S   = 8,   // negative                       (SF==1)
    vmi_COND_NS  = 9,   // not negative                   (SF==0)
    vmi_COND_P   = 10,  // parity even                    (PF==1)
    vmi_COND_NP  = 11,  // not parity even                (PF==0)
    vmi_COND_L   = 12,  // less (signed)                  (SF!=OF)
    vmi_COND_NL  = 13,  // not less (signed)              (SF==OF)
    vmi_COND_LE  = 14,  // less or equal (signed)         (ZF==1 || SF!=OF)
    vmi_COND_NLE = 15   // not less or equal (signed)     (ZF==0 && SF==OF)
} vmiCondition;
```

Example

The OVP ARM model uses this function to implement table branch instructions:

```
ARM_MORPH_FN(armEmitTBZ) {

    Uns32  bits = ARM_GPR_BITS(state);
    vmiReg rn   = GET_RS(state, r1);
    vmiReg tf   = getTemp(state, 32);
    Uns32  bit  = state->info.c;

    // do the comparison
    vmimtTestRC(state, bits, vmi_COND_NZ, rn, (1ULL<<bit), tf);

    // get information about the jump
    armJumpInfo ji;
    seedJumpInfo(&ji, state, False, False, True);

    // do the jump
```

```
    armEmitCondJump(state, &ji, tf, state->attrs->jumpIfTrue);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

3.37 vmimtSetShiftMask

Prototype

```
void vmimtSetShiftMask(Uns8 mask);
```

Description

The shift/rotate amount for shift/rotate binops is normally masked to `bits-1`, where `bits` is the operand size. It is possible to override this default shift mask with any mask in the range 1..255 by immediately preceding the binop with a call to `vmimtSetShiftMask` giving the required shift mask.

Example 1

On the Intel x86 processor, all shifts, including byte and word size shifts, are masked to the range 0..31. This behavior can be specified as follows:

```
vmimtSetShiftMask(31);
vmimtBinopRR(CPUX_GBITS, vmi_ROR, CPUX_REG(rd), CPUX_REG(ra), 0);
```

Example 2

On ARM processors, the shift amount is byte sized. This behavior can be specified as follows:

```
vmimtSetShiftMask(255);
vmimtBinopRRR(CPUX_GBITS, vmi_ROR, CPUX_REG(rd), CPUX_REG(ra) , CPUX_REG(rb), 0);
```

Notes and Restrictions

1. The call to `vmimtSetShiftMask` must immediately precede the `vmimtBinop*` call to which the shift mask must be applied.
2. If `vmimtSetShiftMask` is called before a `vmimtBinop*` call that is not one of the shift/rotate opcodes (`vmi_ROR`, `vmi_ROL`, `vmi_RCL`, `vmi_RCR`, `vmi_SHL`, `vmi_SHR` or `vmi_SAR`) it is ignored.
3. If `vmimtSetShiftMask` is called before any VMI morph time interface function that is not a `vmimtBinop*` call, it is ignored.

4 Memory Operations

This section describes emission functions for memory operations (loads and stores).

In VMI versions prior to 4.3.0, only the *current processor data domain* could be targeted by load and store primitives.

From VMI version 4.3.0 onwards *any domain* can be targeted by a load or store primitive. This is useful in situations where loads or stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

From VMI version 6.2.0 onwards, load and store operations can handle any operand size from 1 to 128 bytes.

From VMI version 6.3.0 onwards, load and store functions can have *constraints* applied to their operation, as documented below.

4.1 Memory Constraints

The action of functions that emit code to load and store from memory can be refined using a *memory constraint*. Available constraints are defined by the `memConstraint` type in `vmiTypes.h`:

```
typedef enum memConstraintE {

    MEM_CONSTRAINT_NONE    = 0x0, // no constraint
    MEM_CONSTRAINT_ALIGNED = 0x1, // access must not be misaligned
    MEM_CONSTRAINT_USER1   = 0x2, // no access when privilege is MEM_PRIV_USER1
    MEM_CONSTRAINT_USER2   = 0x4, // no access when privilege is MEM_PRIV_USER2

    // legacy alias
    MEM_CONSTRAINT_NO_DEVICE = MEM_CONSTRAINT_USER1

} memConstraint;
```

The `memConstraint` type is a bitmask, so constraints may be combined.

Constraint `MEM_CONSTRAINT_ALIGNED` specifies that the load or store operation must use an address that is aligned to the size of the data element. If the address is misaligned, a simulated exception will be taken using the `rdAlignExceptCB` or `wrAlignExceptCB` callback functions specified in the attribute structure of the processor. See the *Imperas Processor Modeling Guide* for more information about the attributes structure.

Constraints `MEM_CONSTRAINT_USER1` and `MEM_CONSTRAINT_USER2` specify that the load or store operation must not access a memory region with one of two user-defined privilege constraints. Such regions are specified by using a `memPriv` including `MEM_PRIV_USER1` or `MEM_PRIV_USER2`, respectively, when a memory region is defined using `vmirtAliasMemoryVM` or modified using `vmirtProtectMemory` (refer to the *VMI*

Run Time Function Reference for more information about these functions). If the memory region is of type MEM_PRIV_USER1 and the access specifies constraint MEM_CONSTRAINT_USER1, or the memory region is of type MEM_PRIV_USER2 and the access specifies constraint MEM_CONSTRAINT_USER2, then a simulated exception will be taken using the rdDeviceExceptCB or wrDeviceExceptCB callback functions specified in the attribute structure of the processor. See the *Imperas Processor Modeling Guide* for more information about the attributes structure.

Note that in versions of the VMI interface prior to version 6.45.1 there was a single user-specified constraint, indicated by the MEM_PRIV_DEVICE / MEM_CONSTRAINT_NO_DEVICE pair. This has been generalized to allow two distinct constraints to be specified.

4.2 vmimtStoreRRO

Prototype

```
void vmimtStoreRRO(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    vmiReg        rb,
    memEndian     endian,
    memConstraint constraint
);
```

Description

Emit code to store the value of processor register `rb` (of size `bits`) to an address calculated from the value of `ra` plus the fixed displacement `offset`.

The size of the address register `ra` is derived from the size of the *current processor data domain*, as follows:

1. If the domain is up to 16 bits: `ra` is 16 bits (i.e. 2 byte address);
2. If the domain is 17-32 bits: `ra` is 32 bits (i.e. 4 byte address);
3. If the domain is 33-63 bits: `ra` is 64 bits (i.e. 8 byte address).

If `ra` has the value `VMI_NOREG`, the address at which to store is `offset` alone. If the store address calculated by `ra+offset` is invalid (the platform defines no writable entity at that address), the simulator will call the store privilege exception handler (`wrPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtStoreRRODomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `endian` is `MEM_ENDIAN_LITTLE`, then the store is performed little-endian. If it is `MEM_ENDIAN_BIG`, the store is performed big-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any store address snap handler (`wrSnapCB`) defined for the processor model. If there is no store address snap handler, or the store address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any store alignment exception handler (`wrAlignExceptCB`) defined for the processor model.

`bits` not equal to 8, 16, 32 or 64

The `endian` argument is ignored. Data is always stored little-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte store) then the simulator will verify that the address is aligned to an 8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

When the store is executed, it is broken down into individual transactions of 1, 2, 4 or 8 bytes in size, starting with the largest possible size. For example, a 10-byte store will be broken down into an 8-byte store followed by a 2-byte store.

Example

The OVP RISC-V model uses this function to implement store instructions:

```
static void emitStoreCommon(
    riscvMorphStateP state,
    vmiReg           rs,
    vmiReg           ra,
    memConstraint    constraint
) {
    Uns32     memBits = state->info.memBits;
    Uns64     offset  = state->info.c;
    memEndian endian  = getDataEndian(state->riscv);

    vmimtStoreRRO(memBits, offset, ra, rs, endian, constraint);
}
```

Notes and Restrictions

1. When the the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024..

4.3 vmimtStoreRCO

Prototype

```
void vmimtStoreRCO(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    Uns64         c,
    memEndian     endian,
    memConstraint constraint
);
```

Description

Emit code to store the constant value `c` (of size `bits`) to an address calculated from the value of `ra` plus the fixed displacement `offset`.

The size of the address register `ra` is derived from the size of the *current processor data domain*, as follows:

1. If the domain is up to 16 bits: `ra` is 16 bits (i.e. 2 byte address);
2. If the domain is 17-32 bits: `ra` is 32 bits (i.e. 4 byte address);
3. If the domain is 33-63 bits: `ra` is 64 bits (i.e. 8 byte address).

If `ra` has the value `VMI_NOREG`, the address at which to store is `offset` alone. If the store address calculated by `ra+offset` is invalid (the platform defines no writable entity at that address), the simulator will call the store privilege exception handler (`wrPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtStoreRCODomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `endian` is `MEM_ENDIAN_LITTLE`, then the constant value is stored little-endian. If it is `MEM_ENDIAN_BIG`, the constant value is stored big-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any store address snap handler (`wrSnapCB`) defined for the processor model. If there is no store address snap handler, or the store address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any store alignment exception handler (wrAlignExceptCB) defined for the processor model.

bits not equal to 8, 16, 32 or 64

If endian is MEM_ENDIAN_LITTLE, then the constant value is stored little-endian. If it is MEM_ENDIAN_BIG, the constant value is stored big-endian.

If constraint does not include MEM_CONSTRAINT_ALIGNED, the simulator does not perform alignment checking for the store address. If constraint includes MEM_CONSTRAINT_ALIGNED, the simulator verifies that the address calculated by ra+offset is a multiple of the byte size implied by bits; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if bits is 80 (implying a 10-byte store) then the simulator will verify that the address is aligned to an 8-byte boundary. Actions for misaligned address are the same as for bits of 8, 16, 32 or 64, as described above.

When the store is executed, it is broken down into individual transactions of 1, 2, 4 or 8 bytes in size, starting with the largest possible size. For example, a 10-byte store will be broken down into an 8-byte store followed by a 2-byte store.

When bits is greater than 64, the constant value is written repeatedly into each 64-bit (8 byte) element of the written memory.

Example

The OVP ARC model uses this function to implement constant store instructions:

```
void arcEmitStoreRCO(
    arcMorphStateP state,
    Uns32          bits,
    Uns32          offset,
    vmiReg         ra,
    Uns32          c,
    Bool           checkAlign
) {
    // emit stack check prologue if required
    Bool      doStackCheck = emitSCPrologue(state, offset, ra);
    memEndian endian       = state->arc->endian;

    if(doStackCheck) {

        // try-store is required if stack checking is enabled (otherwise a stack
        // check error will not prevent the store from happening)
        vmimtTryStoreRC(bits, offset, ra, checkAlign);

        // emit stack check epilogue if required
        emitSCEpilogue(state, False);
    }

    // do the store if stack check succeeds
    vmimtStoreRCO(bits, offset, ra, c, endian, checkAlign);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024..
5. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

4.4 vmimtLoadRRO

Prototype

```
void vmimtLoadRRO(
    Uns32         destBits,
    Uns32         memBits,
    Addr          offset,
    vmiReg        rd,
    vmiReg        ra,
    memEndian     endian,
    Bool          signExtend,
    memConstraint constraint
);
```

Description

Emit code to load the value of processor register `rd` (of size `destBits`) from an address calculated from the value of `ra` plus the fixed displacement `offset`.

The size of the address register `ra` is derived from the size of the *current processor data domain*, as follows:

1. If the domain is up to 16 bits: `ra` is 16 bits (i.e. 2 byte address);
2. If the domain is 17-32 bits: `ra` is 32 bits (i.e. 4 byte address);
3. If the domain is 33-63 bits: `ra` is 64 bits (i.e. 8 byte address).

If `ra` has the value `VMI_NOREG`, the address from which to load is `offset` alone. If the load address calculated by `ra+offset` is invalid (the platform defines no readable entity at that address), the simulator will call the load privilege exception handler (`rdPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtLoadRRODomain`.

Arguments `memBits` and `destBits` must be a multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when both values are 8, 16, 32 or 64 to all other cases; see the following subsections.

`destBits` and `memBits` both equal to 8, 16, 32 or 64

The size of the value to load from memory is given by `memBits`, which must be less than or equal to `destBits`. If `memBits` is less than `destBits`, the value will be sign-extended to `destBits` (if `signExtend` is `True`) or zero-extended (if `signExtend` is `False`).

If `endian` is `MEM_ENDIAN_LITTLE`, then the load is performed little-endian. If it is `MEM_ENDIAN_BIG`, the load is performed big-endian. Any required sign extension is done after endian swapping.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any load address snap handler (`rdSnapCB`) defined for the processor model. If there is no load address snap handler, or the load address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any load alignment exception handler (`rdAlignExceptCB`) defined for the processor model.

`destBits` and `memBits` not equal to 8, 16, 32 or 64

The size of the value to load from memory is given by `memBits`, which must be equal to `destBits`. The `signExtend` argument is ignored

The `endian` argument is ignored. Data is always loaded little-endian.

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte load) then the simulator will verify that the address is aligned to an 8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

When the load is executed, it is broken down into individual transactions of 1, 2, 4 or 8 bytes in size, starting with the largest possible size. For example, a 10-byte load will be broken down into an 8-byte load followed by a 2-byte load.

Example

The OVP RISC-V model uses this function to implement load instructions:

```
static void emitLoadCommon(
    riscvMorphStateP state,
    vmiReg           rd,
    vmiReg           ra,
    memConstraint    constraint
) {
    Uns32     bits    = getRegBits(state, 0);
    Uns32     memBits = state->info.memBits;
    Uns64     offset  = state->info.c;
    Bool      sExtend = !state->info.unsExt;
    memEndian endian  = getDataEndian(state->riscv);

    vmimtLoadRRO(bits, memBits, offset, rd, ra, endian, sExtend, constraint);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.

2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `destBits` and `memBits` `bits` must be a multiple of 8 in the range 8 to 1024.
5. `destBits` must be equal to or greater than `memBits` (if both are 8, 16, 32 or 64). Both values must be equal in all other cases.

4.5 vmimtTryStoreRC

Prototype

```
void vmimtTryStoreRC(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

Emit code to trigger any exceptions that would be generated if a store of the passed bit size was made to the passed address. If no exceptions would be generated, the function has no effect.

If `ra` has the value `VMI_NOREG`, the address at which to store is `offset` alone. If the store address calculated by `ra+offset` is invalid (the platform defines no writable entity at that address), the simulator will call the store privilege exception handler (`wrPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtTryStoreRCDomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

`bits` equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any store address snap handler (`wrSnapCB`) defined for the processor model. If there is no store address snap handler, or the store address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any store alignment exception handler (`wrAlignExceptCB`) defined for the processor model.

`bits` not equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the store address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte store) then the simulator will verify that the address is aligned to an

8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

Example

The OVP RISC-V model uses this function with atomic instructions to ensure a Store/AMO exception is taken in preference to a Load exception:

```
static void emitAMOCommonRRR(riscvMorphStateP state, amoCB opCB) {

    vmiReg        rd         = getReg(state, 0);
    vmiReg        rs         = getReg(state, 1);
    vmiReg        ra         = getReg(state, 2);
    Uns32         bits       = getRegBits(state, 0);
    memConstraint constraint = getLoadStoreConstraintA(state);
    vmiReg        tmp1       = getTmp(state, 0);
    vmiReg        tmp2       = getTmp(state, 1);

    // for this instruction, memBits is bits
    state->info.memBits = bits;

    // this is an atomic operation
    vmimtAtomic();

    // generate Store/AMO exception in preference to Load exception
    vmimtTryStoreRC(bits, 0, ra, constraint);

    // generate results using tmp1 and tmp2
    emitLoadCommon(state, tmp1, ra, constraint);
    opCB(state, bits, tmp2, tmp1, rs);
    emitStoreCommon(state, tmp2, ra, constraint);
    vmimtMoveRR(bits, rd, tmp1);

    writeReg(state, 0);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

4.6 vmimtTryLoadRC

Prototype

```
void vmimtTryLoadRC(
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

Emit code to trigger any exceptions that would be generated if a load of the passed bit size was made to the passed address. If no exceptions would be generated, the function has no effect.

If `ra` has the value `VMI_NOREG`, the address from which to load is `offset` alone. If the load address calculated by `ra+offset` is invalid (the platform defines no readable entity at that address), the simulator will call the load privilege exception handler (`rdPrivExceptCB`) defined for the processor model.

This function emits code that targets the current processor data domain. From VMI version 4.3.0, it is possible to specify *any* target domain – see related function `vmimtTryLoadRCDomain`.

Argument `bits` can be any multiple of eight between 8 (i.e. 1 byte) and 1024 (i.e. 128 bytes). Behavior is different when `bits` is 8, 16, 32 or 64 to all other cases; see the following subsections.

bits equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`. If the address is misaligned, the simulator will first call any load address snap handler (`rdSnapCB`) defined for the processor model. If there is no load address snap handler, or the load address snap handler returns zero (indicating no address snap is to be performed), the simulator will then call any load alignment exception handler (`rdAlignExceptCB`) defined for the processor model.

bits not equal to 8, 16, 32 or 64

If `constraint` does not include `MEM_CONSTRAINT_ALIGNED`, the simulator does not perform alignment checking for the load address. If `constraint` includes `MEM_CONSTRAINT_ALIGNED`, the simulator verifies that the address calculated by `ra+offset` is a multiple of the byte size implied by `bits`; if the implied byte size is not a power of two, then the next smallest power of two is chosen. For example, if `bits` is 80 (implying a 10-byte load) then the simulator will verify that the address is aligned to an

8-byte boundary. Actions for misaligned address are the same as for `bits` of 8, 16, 32 or 64, as described above.

Example

The OVP ARC model uses this function to ensure correct behavior when stack checking is enabled:

```
void arcEmitLoadRRO(
    arcMorphStateP state,
    Uns32          destBits,
    Uns32          memBits,
    Uns32          offset,
    vmiReg         rd,
    vmiReg         ra,
    Bool           signExtend,
    Bool           checkAlign
) {
    // emit stack check prologue if required
    Bool      doStackCheck = emitSCPrologue(state, offset, ra);
    memEndian endian       = state->arc->endian;

    if(doStackCheck) {

        // try-load is required if stack checking is enabled (otherwise a stack
        // check error will not prevent the load from happening)
        vmimtTryLoadRC(memBits, offset, ra, checkAlign);

        // emit stack check epilogue if required
        emitSCEpilogue(state, True);
    }

    // do the load if stack check succeeds
    vmimtLoadRRO(destBits, memBits, offset, rd, ra, endian, signExtend, checkAlign);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

4.7 vmimtPreLoadRC

Prototype

```
void vmimtPreLoadRC(
    Addr           offset,
    vmiReg         ra,
    memPreloadType type
);
```

Description

Emit code indicate a *preload* of an address calculated by `ra+offset`. Preloads are hints to cache and memory subsystems that data at the given address will be accessed soon and should be moved closer to the processor in the memory hierarchy. The class of preload is indicated by the `type` argument of type `memPreloadType`, defined in `vmiTypes.h` as follows:

```
typedef enum memPreloadTypeE {
    MEM_PLT_LOAD,       // preload for likely load
    MEM_PLT_STORE,      // preload for likely store
    MEM_PLT_FETCH,      // preload for likely execute
} memPreloadType;
```

For simulation purposes, preloads have no effect; the only observable effect is when the *instruction attributes API* is being used. This API is designed for use in intercept libraries that are intended to monitor the executed instruction stream to emulate the effects of pipelines and memory hierarchy. See the *OVP VMI Run Time Function Reference* manual for more details.

Example

The OVP ARM model uses this function to implement prefetch instructions:

```
ARM_MORPH_FN(armEmitPRFML) {

    vmiReg base   = getTemp(state, 64);
    Uns64  offset = state->info.t - state->info.thisPC;

    // get base address for load
    vmimtMoveRSimPC(64, base);

    // do the prefetch
    vmimtPreLoadRC(offset, base, state->info.prfm);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.

4.8 vmimtStoreRRODomain

Prototype

```
void vmimtStoreRRODomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    vmiReg        rb,
    memEndian     endian,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtStoreRRO`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the store is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtStoreRRO`).

If domain is non-`NULL`, the store is directed to the *specified domain*.

This function is useful in situations where stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtStoreRRO` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtStoreRRODomain`:

```
void armEmitStoreRRO(
    armMorphStateP state,
    Uns32          bits,
    Uns32          offset,
    vmiReg         ra,
    vmiReg         rb
) {
    memDomainP    domain     = getDomain(state);
    memEndian     endian     = getEndian(state, bits);
    memConstraint constraint = getConstraint(state, bits);
    vmimtStoreRRODomain(domain, bits, offset, ra, rb, endian, constraint);
}
```

Notes and Restrictions

1. When the the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

4.9 *vmimtStoreRCODomain*

Prototype

```
void vmimtStoreRCODomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    Uns64         c,
    memEndian     endian,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtStoreRCO`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the store is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtStoreRCO`).

If domain is non-`NULL`, the store is directed to the *specified domain*.

This function is useful in situations where stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtStoreRCO` for details of other arguments to this function.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.
5. For `bits` less than 64, the unused most significant bits of `c` are silently discarded.

4.10 vmimtLoadRRODomain

Prototype

```
void vmimtLoadRRODomain(
    memDomainP    domain,
    Uns32         destBits,
    Uns32         memBits,
    Addr          offset,
    vmiReg        rd,
    vmiReg        ra,
    memEndian     endian,
    Bool          signExtend,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtLoadRRO`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the load is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtLoadRRO`).

If domain is non-`NULL`, the load is directed to the *specified domain*.

This function is useful in situations where loads are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtLoadRRO` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtLoadRRODomain`:

```
void armEmitLoadRRO(
    armMorphStateP state,
    Uns32          destBits,
    Uns32          memBits,
    Uns32          offset,
    vmiReg         rd,
    vmiReg         ra,
    Bool           signExtend,
    Bool           isReturn
) {
```

```
    memDomainP    domain     = getDomain(state);
    memEndian     endian     = getEndian(state, memBits);
    memConstraint constraint = getConstraint(state, bits);

    // emit single load
    vmimtLoadRRODomain(
        domain, destBits, memBits, offset, rd, ra, endian, signExtend, constraint
    );
    setVariable(state, rd, isReturn);
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `destBits` and `memBits bits` must be a multiple of 8 in the range 8 to 1024.
5. `destBits` must be equal to or greater than `memBits` (if both are 8, 16, 32 or 64). Both values must be equal in all other cases.

4.11 vmimtTryStoreRCDomain

Prototype

```
void vmimtTryStoreRCDomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtTryStoreRC`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the store is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtTryStoreRC`).

If domain is non-`NULL`, the store is directed to the *specified domain*.

This function is useful in situations where stores are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtTryStoreRC` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtTryStoreRCDomain`:

```
void armEmitTryStoreRC(
    armMorphStateP state,
    Uns32          bits,
    Addr           offset,
    vmiReg         ra
) {
    memDomainP domain = getDomain(state);
    vmimtTryStoreRCDomain(domain, bits, offset, ra, getConstraint(state, bits));
}
```


Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

4.12 vmimtTryLoadRCDomain

Prototype

```
void vmimtTryLoadRCDomain(
    memDomainP    domain,
    Uns32         bits,
    Addr          offset,
    vmiReg        ra,
    memConstraint constraint
);
```

Description

This function is similar to `vmimtTryLoadRC`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the load is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtTryLoadRC`).

If domain is non-`NULL`, the load is directed to the *specified domain*.

This function is useful in situations where loads are targeted at a different domain to the default domain for the current processor mode. For example, the `LDRT` and `STRT` instructions in the ARM processor perform user-level loads and stores when executing in privileged mode.

See the description of `vmimtTryLoadRC` for details of other arguments to this function.

Example

This example is from the OVP ARM model. This processor has *translating* load/store instructions that allow the user address space to be read or written from privileged mode. A utility functions selects either the user address space or the default address space, depending on attributes of the decoded instruction:

```
inline static memDomainP getDomain(armMorphStateP state) {
    return doTranslate(state) ? state->arm->dds.vmUser : 0;
}
```

This domain is then specified to `vmimtTryLoadRCDomain`:

```
void armEmitTryLoadRC(
    armMorphStateP state,
    Uns32          bits,
    Addr           offset,
    vmiReg         ra
) {
    memDomainP domain = getDomain(state);
    vmimtTryLoadRCDomain(domain, bits, offset, ra, getConstraint(state, bits));
}
```

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.
4. `bits` must be a multiple of 8 in the range 8 to 1024.

4.13 vmimtPreLoadRCDomain

Prototype

```
void vmimtPreLoadRCDomain(
    memDomainP     domain,
    Addr           offset,
    vmiReg         ra,
    memPreloadType type
);
```

Description

This function is similar to `vmimtPreLoadRC`, except for the `domain` argument, which allows the memory domain for the store to be specified.

If `domain` is `NULL`, the preload is directed to the *current processor data domain* (in other words, behavior is identical to `vmimtTryLoadRC`).

If domain is non-`NULL`, the preload is directed to the *specified domain*.

This function is useful in situations where preloads are targeted at a different domain to the default domain for the current processor mode.

See the description of `vmimtPreLoadRC` for details of other arguments to this function.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. When the data address bus width is 32 bits or less, the appropriate type for the address register `ra` in the processor structure is a 32-bit unsigned (`Uns32`), unless modified by a preceding `vmimtSetAddressMask` call.
2. When the data address bus width is 33 to 64 bits, the appropriate type for the address register `ra` in the processor structure is a 64-bit unsigned (`Uns64`), unless modified by a preceding `vmimtSetAddressMask` call.
3. Data address bus widths greater than 64 bits are not supported.

5 Control Flow Operations

This section describes emission functions for control flow operations: inter-instruction and intra-instruction unconditional and conditional jumps.

Inter-instruction jumps correspond to control transfers in the simulated processor instruction set.

Intra-instruction jumps are required when the translation of a simulated instruction requires loops or conditional branches (although if the control behavior is complicated, it is usually easier and more efficient to use an embedded call instead - see `vmimtCallResultAttrs` and related functions).

5.1 *vmimtSetAddressMask*

Prototype

```
void vmimtSetAddressMask(Uns64 mask);
```

Description

This call can be used *immediately prior* to jump, load or store morph-time operations to specify masking of addresses.

When used immediately prior to `vmimt*Jump*` calls, this specifies required masking of target and link addresses. For example, specifying an address mask of `-2` will cause the least significant bit of target and return addresses to be ignored.

When used immediately prior to `vmimt*Load*` or `vmimt*Store*` calls, this specifies the effective number of bits used to calculate the load/store address. Any computed address will be modified so that all bits above the most-significant non-zero bit in the mask are zeroed. Note that more advanced load/store address masking is also possible using run-time function `vmirtSetLoadStoreMask`.

Example

The OVP ARM model uses this function to implement indirect jumps in AArch32 state. In this state, the least-significant bit determines the execution mode at the target address (ARM or Thumb):

```
void armEmitUncondJumpReg(
    armMorphStateP state,
    armJumpInfoP   ji,
    vmiReg         toReg
) {
    emitClearITState(state);

    vmimtSetAddressMask(-2);

    vmimtUncondJumpReg(
        ji->linkPC,
        toReg,
        ji->linkReg,
        ji->hint
    );
}
```

Notes and Restrictions

1. If `vmimtSetAddressMask` is not called immediately prior to a `vmimt*Jump*`, `vmimt*Load*` or `vmimt*Store*` function, it is ignored.

5.2 vmimtUncondJump

Prototype

```
void vmimtUncondJump(
    Addr        linkPC,
    Addr        toAddress,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform an unconditional inter-instruction branch to `toAddress`.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                        // no jump hint

    vmi_JH_CALL      = 0x01,                        // call
    vmi_JH_RETURN    = 0x02,                        // return
    vmi_JH_INT       = 0x04,                        // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,      // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,    // interrupt return

    vmi_JH_RELATIVE  = 0x08                         // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Example

The OVP RISC-V model uses this function to implement jump-and-link instructions:

```
static RISCV_MORPH_FN(emitJAL) {

    vmiReg      lr     = getReg(state, 0);
    Uns64       tgt    = state->info.c;
    Uns64       linkPC = getLinkPC(state);
    vmiJumpHint hint   = isLR(lr) ? vmi_JH_CALL : vmi_JH_NONE;

    vmimtUncondJump(linkPC, tgt, lr, hint|vmi_JH_RELATIVE);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

5.3 vmimtUncondJumpDelaySlot

Prototype

```
void vmimtUncondJumpDelaySlot(
    Uns32          slotOps,
    Addr           linkPC,
    Addr           toAddress,
    vmiReg         linkReg,
    vmiJumpHint    hint,
    vmiPostSlotFn slotCB
);
```

Description

Emit code to perform an unconditional inter-instruction branch to `toAddress` with `slotOps` subsequent delay slot instructions, which will be executed prior to taking the branch.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                         // no jump hint

    vmi_JH_CALL      = 0x01,                         // call
    vmi_JH_RETURN    = 0x02,                         // return
    vmi_JH_INT       = 0x04,                         // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,       // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,     // interrupt return

    vmi_JH_RELATIVE  = 0x08                          // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken (if, for example, if there is a simulated exception in the delay slot instruction) the function is *not* called. The callback function is passed the processor as its only argument.

If `slotOps` and `slotCB` are 0, this function is equivalent to `vmiUncondJump`.

Example

The OVP MIPS model uses this function to implement unconditional branch instructions. In this processor, the link address is updated even if the branch fails (because memory at the target address is not executable, for example) so link register update is done explicitly before the jump:

```
static void emitBranchU(
    mipsInstructionInfoP info,
    Uns32                slotInsns,
    mipsP                mips,
    Bool                 link
) {
    mipsUnsArch dst  = info->c;
    vmiJumpHint hint = getBranchHint(link);

    // set up the link return address
    if(link) {
        emitSetLinkAddress(info, slotInsns, mips, MIPS_REG_RA);
    }

    // do the jump
    vmimtUncondJumpDelaySlot(slotInsns, 0, dst, VMI_NOREG, hint, 0);
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0, 1, 2 or 3.
2. When the the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtUncondJumpDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

5.4 vmimtUncondJumpReg

Prototype

```
void vmimtUncondJumpReg(
    Addr        linkPC,
    vmiReg      toReg,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform an unconditional indirect jump to the address in processor register `toReg`. This function is typically used to generate code for calls through function pointers and return instructions.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                        // no jump hint

    vmi_JH_CALL      = 0x01,                        // call
    vmi_JH_RETURN    = 0x02,                        // return
    vmi_JH_INT       = 0x04,                        // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,      // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,    // interrupt return

    vmi_JH_RELATIVE  = 0x08                         // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Example

The OVP RISC-V model uses this function to implement unconditional branch-and-link instructions:

```
static RISCV_MORPH_FN(emitJALR) {

    vmiReg      lr     = getReg(state, 0);
    vmiReg      ra     = getReg(state, 1);
    Uns32       bits   = getRegBits(state, 0);
    Uns64       offset = state->info.c;
    Uns64       linkPC = getLinkPC(state);
    vmiJumpHint hint;

    // calculate target address if required
    if(offset) {
        vmiReg tmp = getTmp(state, 0);
        vmimtBinopRRC(bits, vmi_ADD, tmp, ra, offset, 0);
        ra = tmp;
    }

    // derive jump hint
    if(isLR(ra)) {
        hint = vmi_JH_RETURN;
    } else if(isLR(lr)) {
        hint = vmi_JH_CALL;
    } else {
        hint = vmi_JH_NONE;
    }

    vmimtUncondJumpReg(linkPC, ra, lr, hint|vmi_JH_RELATIVE);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

5.5 vmimtUncondJumpRegDelaySlot

Prototype

```
void vmimtUncondJumpRegDelaySlot(
    Uns32          slotOps,
    Addr           linkPC,
    vmiReg         toReg,
    vmiReg         linkReg,
    vmiJumpHint    hint,
    vmiPostSlotFn slotCB
);
```

Description

Emit code to perform an unconditional indirect jump to the address in processor register `toReg` with `slotOps` subsequent delay slot instructions, which will be executed prior to taking the branch. This function is typically used to generate code for calls through function pointers and return instructions.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` as the branch is taken – this allows *branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                          // no jump hint

    vmi_JH_CALL      = 0x01,                          // call
    vmi_JH_RETURN    = 0x02,                          // return
    vmi_JH_INT       = 0x04,                          // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,        // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,      // interrupt return

    vmi_JH_RELATIVE  = 0x08                           // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken (if, for example, if there is a simulated exception in the delay slot instruction) the function is *not* called. The callback function is passed the processor as its only argument.

If `slotOps` and `slotCB` are 0, this function is equivalent to `vmiUncondJumpReg`.

Example

The OVP ARC model uses this function to implement unconditional indirect jump instructions:

```
void arcEmitUncondJumpRegDelaySlot(
    arcMorphStateP state,
    arcJumpInfoP   ji,
    vmiReg         toReg
) {
    Uns32 bits    = ARC_GPR_BITS;
    Uns32 BTAMask = getBTAMask(state);

    if(state->info.ds) {

        // delay-slot jumps
        arcBlockStateP blockState = state->blockState;
        vmiReg         bta        = ARC_AUX_REG(bta);
        vmiReg         de         = ARC_DE;

        // set status32.DE to indicate branch is to be taken
        vmimtMoveRC(8, de, 1);

        // set BTA with target address
        vmimtBinopRRC(bits, vmi_AND, bta, toReg, BTAMask, 0);
    }

    // mask jump target to implemented bits
    vmimtSetAddressMask(BTAMask);

    // emit the jump
    vmimtUncondJumpRegDelaySlot(
        state->info.ds,
        ji->linkPC,
        toReg,
        ji->linkReg,
        ji->hint,
        0
    );
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0, 1, 2 or 3.
2. When the the instruction address bus width is 32 bits or less, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address registers `toReg` and `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. If register `toReg` is updated by delay slot instructions, the target address is not affected (it is the original value of `toReg`).
5. `vmimtUncondJumpRegDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

5.6 vmimtCondJump

Prototype

```
void vmimtCondJump(
    vmiReg      flag,
    Bool        jumpIfTrue,
    Addr        linkPC,
    Addr        toAddress,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform a conditional inter-instruction branch. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to target address `toAddress` if the value of the `flag` register is non-zero and execution will continue with the next `vmi` operation if the `flag` register is zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target address `toAddress` if the value of the `flag` register is zero and execution will continue with the next `vmi` operation if the `flag` register is non-zero.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                         // no jump hint

    vmi_JH_CALL      = 0x01,                         // call
    vmi_JH_RETURN    = 0x02,                         // return
    vmi_JH_INT       = 0x04,                         // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,       // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,     // interrupt return

    vmi_JH_RELATIVE  = 0x08                          // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

The OVP RISC-V model uses this function to implement conditional branch instructions:

```
static RISCV_MORPH_FN(emitBranchRR) {

    vmiReg rs1  = getReg(state, 0);
    vmiReg rs2  = getReg(state, 1);
    Uns32  bits = getRegBits(state, 0);
    Uns64  tgt  = state->info.c;
    vmiReg tmp  = getTmp(state, 0);

    vmimtCompareRR(bits, state->attrs->cond, rs1, rs2, tmp);
    vmimtCondJump(tmp, True, 0, tgt, VMI_NOREG, vmi_JH_RELATIVE);
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

5.7 vmimtCondJumpDelaySlot

Prototype

```
void vmimtCondJumpDelaySlot(
    Uns32         slotOps,
    vmiReg        flag,
    Bool          jumpIfTrue,
    Addr          linkPC,
    Addr          toAddress,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB
);
```

Description

Emit code to perform a conditional inter-instruction branch with `slotOps` subsequent delay slot instructions. The processor flag register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is zero.

`slotOps` instructions after the current instruction will be executed prior to taking the branch. These instructions will be executed whether or not the branch is taken. If `slotOps` is 0, this function is equivalent to `vmiCondJump`.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                         // no jump hint

    vmi_JH_CALL      = 0x01,                         // call
    vmi_JH_RETURN    = 0x02,                         // return
    vmi_JH_INT       = 0x04,                         // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,       // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,     // interrupt return

    vmi_JH_RELATIVE  = 0x08                          // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.
Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

The OVP MIPS model uses this function to implement conditional branch instructions:

```
static void emitBranch(
    mipsInstructionInfoP info,
    Uns32                slotInsns,
    mipsP                mips,
    vmiCondition         cond,
    Bool                 link,
    Bool                 annul
) {
    Uns32       bits     = MIPS_ARCH_BITS;
    vmiReg      rt       = getR1(info);
    vmiReg      rs       = getR2(info);
    mipsUnsArch dst      = info->c;
    vmiJumpHint hint     = getBranchHint(link);
    vmiReg      tempFlag = MIPS_TEMPFLAG(0);

    // do the required comparison, setting MIPS_TEMPFLAG
    vmimtCompareRR(bits, cond, rs, rt, tempFlag);

    // set up the link return address (even if the condition is false and we
    // don't actually call the function)
    if(link) {
        emitSetLinkAddress(info, slotInsns, mips, MIPS_REG_RA);
    }

    // do the jump
    if(annul) {
        vmimtCondJumpDelaySlotAnnul(
            slotInsns, tempFlag, True, 0, dst, VMI_NOREG, hint, 0
        );
    } else {
        vmimtCondJumpDelaySlot(
            slotInsns, tempFlag, True, 0, dst, VMI_NOREG, hint, 0
        );
    }
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 0, 1, 2 or 3.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

4. `vmimtCondJumpDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

5.8 vmimtCondJumpDelaySlotAnnul

Prototype

```
void vmimtCondJumpDelaySlotAnnul(
    Int32         slotOps,
    vmiReg        flag,
    Bool          jumpIfTrue,
    Addr          linkPC,
    Addr          toAddress,
    vmiReg        linkReg,
    vmiJumpHint   hint,
    vmiPostSlotFn slotCB
);
```

Description

Emit code to perform a conditional inter-instruction branch with the absolute value of `slotOps` subsequent delay slot instructions. The processor flag register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target address `toAddress` only if the value of the `flag` register is zero.

The absolute value of `slotOps` specifies the number of instructions after the current instruction that will be executed prior to taking the branch. If `slotOps` is *positive*, then if the branch is *not taken* some or all of the instruction actions may be annulled. If `slotOps` is *negative*, then if the branch is *taken* some or all of the instruction actions may be annulled. The precise actions that are annulled are identified by a call to `vmimtSkipIfAnnul`, described elsewhere in this section.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                        // no jump hint

    vmi_JH_CALL      = 0x01,                        // call
    vmi_JH_RETURN    = 0x02,                        // return
    vmi_JH_INT       = 0x04,                        // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,      // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,    // interrupt return
```

```
    vmi_JH_RELATIVE  = 0x08                         // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

The OVP MIPS model uses this function to implement conditional branch instructions:

```
static void emitBranch(
    mipsInstructionInfoP info,
    Int32                slotInsns,
    mipsP                mips,
    vmiCondition         cond,
    Bool                 link,
    Bool                 annul
) {
    Uns32       bits     = MIPS_ARCH_BITS;
    vmiReg      rt       = getR1(info);
    vmiReg      rs       = getR2(info);
    mipsUnsArch dst      = info->c;
    vmiJumpHint hint     = getBranchHint(link);
    vmiReg      tempFlag = MIPS_TEMPFLAG(0);

    // do the required comparison, setting MIPS_TEMPFLAG
    vmimtCompareRR(bits, cond, rs, rt, tempFlag);

    // set up the link return address (even if the condition is false and we
    // don't actually call the function)
    if(link) {
        emitSetLinkAddress(info, slotInsns, mips, MIPS_REG_RA);
    }

    // do the jump
    if(annul) {
        vmimtCondJumpDelaySlotAnnul(
            slotInsns, tempFlag, True, 0, dst, VMI_NOREG, hint, 0
        );
    } else {
        vmimtCondJumpDelaySlot(
            slotInsns, tempFlag, True, 0, dst, VMI_NOREG, hint, 0
        );
    }
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to values in the range -3 to 3.
2. When the the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).

3. When the the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtCondJumpDelaySlotAnnul` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

5.9 vmimtCondJumpReg

Prototype

```
void vmimtCondJumpReg(
    vmiReg      flag,
    Bool        jumpIfTrue,
    Addr        linkPC,
    vmiReg      toReg,
    vmiReg      linkReg,
    vmiJumpHint hint
);
```

Description

Emit code to perform a conditional indirect inter-instruction branch. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to the address in processor register `toReg` if the value of the `flag` register is non-zero and execution will continue with the next `vmi` operation if the `flag` register is zero.

If `jumpIfTrue` is `False`, then a jump will be taken to the address in processor register `toReg` if the value of the `flag` register is zero and execution will continue with the next `vmi` operation if the `flag` register is non-zero.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *indirect conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                        // no jump hint

    vmi_JH_CALL      = 0x01,                        // call
    vmi_JH_RETURN    = 0x02,                        // return
    vmi_JH_INT       = 0x04,                        // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,      // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,    // interrupt return

    vmi_JH_RELATIVE  = 0x08                         // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

The OVP ARC model uses this function to terminate zero-overhead loops:

```
void arcEmitEndZOL(arcMorphStateP state) {

    if(state->atZOL) {
        vmimtCondJumpReg(
            ARC_ZOL_BRANCH,
            True,
            0,
            ARC_AUX_REG(lp_start),
            VMI_NOREG,
            vmi_JH_NONE
        );
    }
}
```

Notes and Restrictions

1. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
2. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).

5.10 vmimtCondJumpRegDelaySlot

Prototype

```
void vmimtCondJumpRegDelaySlot(
    Uns32          slotOps,
    vmiReg         flag,
    Bool           jumpIfTrue,
    Addr           linkPC,
    vmiReg         toReg,
    vmiReg         linkReg,
    vmiJumpHint    hint,
    vmiPostSlotFn slotCB

);
```

Description

Emit code to perform a conditional indirect inter-instruction branch with `slotOps` subsequent delay slot instructions. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is zero.

`slotOps` instructions after the current instruction will be executed prior to taking the branch. These instructions will be executed whether or not the branch is taken. If `slotOps` is 0, this function is equivalent to `vmiCondJump`.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *indirect conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                        // no jump hint

    vmi_JH_CALL      = 0x01,                        // call
    vmi_JH_RETURN    = 0x02,                        // return
    vmi_JH_INT       = 0x04,                        // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,      // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,    // interrupt return

    vmi_JH_RELATIVE  = 0x08                         // target is relative

} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.
Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. `slotOps` is currently restricted to 0, 1, 2 or 3.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtCondJumpRegDelaySlot` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

5.11 vmimtCondJumpRegDelaySlotAnnul

Prototype

```
void vmimtCondJumpRegDelaySlotAnnul(
    Int32          slotOps,
    vmiReg         flag,
    Bool           jumpIfTrue,
    Addr           linkPC,
    vmiReg         toReg,
    vmiReg         linkReg,
    vmiJumpHint    hint,
    vmiPostSlotFn slotCB

);
```

Description

Emit code to perform a conditional indirect inter-instruction branch with the absolute value of `slotOps` subsequent delay slot instructions. The processor `flag` register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is non-zero.

If `jumpIfTrue` is `False`, then a jump will be taken to the address in processor register `toReg` only if the value of the `flag` register is zero.

The absolute value of `slotOps` specifies the number of instructions after the current instruction that will be executed prior to taking the branch. If `slotOps` is *positive*, then if the branch is *not taken* some or all of the instruction actions may be annulled. If `slotOps` is *negative*, then if the branch is *taken* some or all of the instruction actions may be annulled. The precise actions that are annulled are identified by a call to `vmimtSkipIfAnnul`, described elsewhere in this section.

If `linkReg` is not `VMI_NOREG`, then address `linkPC` will be loaded into the processor register specified by `linkReg` if the branch is taken – this allows *indirect conditional branch and link* instructions to be easily specified. If `linkReg` is `VMI_NOREG`, the value of `linkPC` is ignored.

Argument `hint` is used to indicate to the simulator the kind of branch taking place. The type is defined in `vmiTypes.h`:

```
typedef enum vmiJumpHintE {

    vmi_JH_NONE      = 0x00,                         // no jump hint

    vmi_JH_CALL      = 0x01,                         // call
    vmi_JH_RETURN    = 0x02,                         // return
    vmi_JH_INT       = 0x04,                         // interrupt
    vmi_JH_CALLINT   = vmi_JH_INT|vmi_JH_CALL,       // interrupt call
    vmi_JH_RETURNINT = vmi_JH_INT|vmi_JH_RETURN,     // interrupt return
```

```
    vmi_JH_RELATIVE  = 0x08                          // target is relative
} vmiJumpHint;
```

Simulator performance is much improved if appropriate hints are given as to whether an instruction is a call, a return, or a simple control transfer because it is then able to match up calls and returns in much the same way as they are optimized in hardware.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. If the branch is not taken the function is *not* called. The callback function is passed the processor as its only argument.

This function is often used in conjunction with `vmimtCompareRR` (and related functions) as shown in the example.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. `slotOps` is currently restricted to values in the range -3 to 3.
2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `linkReg` in the processor structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `linkReg` in the processor structure is a 64-bit unsigned (`Uns64`).
4. `vmimtCondJumpRegDelaySlotAnnul` must be the *last* morph time call issued for a simulated instruction. Attempting to make further `vmimt` calls will cause a simulator fatal error message and terminate simulation.

5.12 vmimtSkipIfAnnul

Prototype

```
void vmimtSkipIfAnnul(void);
```

Description

This routine has no action unless the current instruction is a delay slot instruction of a conditional jump specified by a call to `vmimtCondJumpDelaySlotAnnul` or `vmimtCondJumpRegDelaySlotAnnul`, described elsewhere in this section.

If the current instruction is indeed such a delay slot instruction, this routine causes all behavior following the call to `vmimtSkipIfAnnul` to be skipped if the branch is annulled (not taken).

`vmimtSkipIfAnnul` is typically called once in the pre-morph callback function (defined using the `preMorphCB` field in the `vmiIASAttr` structure).

Example

The OVP ARC model uses this function in the pre-morph callback as follows:

```
VMI_MORPH_FN(arcPreMorphInstruction) {

    arcP           arc   = (arcP)processor;
    arcMorphStateP state = instrState;

    // get instruction and instruction type
    arcDecode(arc, thisPC, &state->info);

    // get morpher attributes for the decoded instruction and initialize other
    // state fields
    state->attrs       = &dispatchTable[state->info.type];
    state->arc         = arc;
    state->nextPC      = (state->info.thisPC + state->info.bytes) & arc->pcMask;
    state->inDelaySlot = inDelaySlot;

    // skip actions after this point if annulling
    vmimtSkipIfAnnul();
}
```

Notes and Restrictions

1. This function is normally used in the *pre-morph* callback instead of the *morph* callback because it should have an effect even if the instruction is implemented in an extension library. See the *OVP Processor Modeling Guide* for more information about extension libraries.

5.13 vmimtGetDelaySlotNextPC

Prototype

```
void vmimtGetDelaySlotNextPC(vmiReg targetReg, Bool getNextPC);
```

Description

This function is useful for determining an instruction address after a delay slot instruction or the branch target of a delay slot instruction. It can be used in processor models, but is more often used in intercept libraries that monitor program control flow.

If `getNextPC` is `True`, then register `targetReg` will be loaded with the address of the *next instruction to be executed after the delay slot instruction*. In the case of a conditional branch that is taken, this will be the branch address; in the case of a conditional branch that is *not* taken, this will be the address of the instruction following the delay slot instruction.

If `getNextPC` is `False`, then register `targetReg` will be loaded with the branch target address irrespective of whether the branch is taken.

Example

This example is extracted from an intercept library used to monitor control flow in MIPS processors:

```
typedef struct vmiosObjectS {
    Addr dsTarget;
} vmiosObject;

static VMIOS_MORPH_FN(mipsMorphCallback) {

    // record branch target address if this is a delay slot instruction
    if(inDelaySlot) {

        // get offset to intercept structure from processor
        UnsPS dstDelta = (UnsPS)object - (UnsPS)processor;

        // get VMI reg for dsTarget field in intercept structure
        vmiReg dsTarget = VMI_CPU_REG_DELTA(vmiosObjectP, dsTarget, dstDelta);

        // assign dsTarget register in intercept structure
        vmimtGetDelaySlotNextPC(dsTarget, False);
    }

    . . . lines omitted for clarity . . .
}
```

Notes and Restrictions

1. This function may only be used in the context of a delay slot instruction. Use outside this context will generate an assertion:
   `vmimtGetDelaySlotNextPC used outside delay slot context`
   The `inDelaySlot` parameter to the functions declared with the `VMI_MORPH_FN` or `VMIOS_MORPH_FN` macros specifies whether the current instruction is a delay slot instruction.

2. When the instruction address bus width is 32 bits or less, the appropriate type for the address register `targetReg` in the processor or intercept library structure is a 32-bit unsigned (`Uns32`).
3. When the instruction address bus width is 33 to 64 bits, the appropriate type for the address register `targetReg` in the processor or intercept library structure is a 64-bit unsigned (`Uns64`).

5.14 vmimtEnterDelaySlotC

Prototype

```
void vmimtEnterDelaySlotC(
    Uns32          slotOps,
    Addr           simPC1,
    Addr           simPC2,
    vmiPostSlotFn slotCB
);
```

Description

This function is used to implement a special form of jump. Firstly, a jump is made to address `simPC1`, and address `simPC2` is scheduled as a delay slot instruction address. After `slotOps` instructions have been executed, control resumes at `simPC2`.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. The callback function is passed the processor as its only argument.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. `slotOps` is currently restricted to 1, 2 or 3.

5.15 vmimtEnterDelaySlotR

Prototype

```
void vmimtEnterDelaySlotR(
    Uns32         slotOps,
    vmiReg        toReg,
    Addr          simPC2,
    vmiPostSlotFn slotCB
);
```

Description

This function is used to implement a special form of jump. Firstly, a jump is made to the address held in register `toReg`, and address `simPC2` is scheduled as a delay slot instruction address. After `slotOps` instructions have been executed, control resumes at `simPC2`.

Argument `slotCB`, if non-`NULL`, specifies a function that is called just before the delayed branch is taken. The callback function is passed the processor as its only argument.

Example

This example is taken from the OVP ARC processor model. This processor has an *execute indexed* instruction (`EI_S`) that executes a single instruction at a computed address before resuming execution at the instruction after the `EI_S` instruction. The required target address is in register `rt`:

```
void arcEnterDelaySlotR(arcMorphStateP state, vmiReg rt) {

    Uns32 bits = ARC_GPR_BITS;

    // set BTA with target address
    vmimtMoveRC(bits, ARC_AUX_REG(bta), state->nextPC);

    // enter delay-slot block
    vmimtEnterDelaySlotR(1, rt, state->nextPC, 0);
}
```

Notes and Restrictions

1. `slotOps` is currently restricted to 1, 2 or 3.

5.16 vmimtNewLabel

Prototype

```
vmiLabelP vmimtNewLabel(void);
```

Description

This function is used to define a label for use with *intra-instruction jumps*, described in following sections. The label indicates a position in the JIT-translated code stream to which execution should branch.

Example

The OVP ARM model uses this function to control whether a processor should stop in a `WFI` instruction:

```
ARM_MORPH_FN(armEmitWFI) {

    vmiLabelP noWait = vmimtNewLabel();

    // don't stop if there are pending interrupts
    vmimtCompareRCJumpLabel(8, vmi_COND_NZ, ARM_PENDING, 0, noWait);

    // halt the processor at the end of this instruction
    armEmitWait(state, AD_WFI);

    // here if interrupt is currently pending
    vmimtInsertLabel(noWait);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump` or `vmimtUncondJump`.
2. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

5.17 vmimtInsertLabel

Prototype

```
void vmimtInsertLabel(vmiLabelP label);
```

Description

This function is used to insert a label previously defined with `vmimtNewLabel` at the current position in the NMI node list. Labels are used to implement *intra-instruction jumps*.

Example

The OVP ARM model uses this function to control whether a processor should stop in a `WFI` instruction:

```
ARM_MORPH_FN(armEmitWFI) {

    vmiLabelP noWait = vmimtNewLabel();

    // don't stop if there are pending interrupts
    vmimtCompareRCJumpLabel(8, vmi_COND_NZ, ARM_PENDING, 0, noWait);

    // halt the processor at the end of this instruction
    armEmitWait(state, AD_WFI);

    // here if interrupt is currently pending
    vmimtInsertLabel(noWait);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump` or `vmimtUncondJump`.
2. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

5.18 vmimtUncondJumpLabel

Prototype

```
void vmimtUncondJumpLabel(vmiLabelP toLabel);
```

Description

This function is used to perform an unconditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

Example

The OVP ARM model uses this function to control whether a processor should stop in a `WFE` instruction:

```
ARM_MORPH_FN(armEmitWFE) {

    vmiLabelP wait = vmimtNewLabel();
    vmiLabelP done = vmimtNewLabel();

    // jump to wait code if no event registered
    vmimtCondJumpLabel(ARM_EVENT, False, wait);

    // clear event register and finish
    vmimtMoveRC(8, ARM_EVENT, 0);
    vmimtUncondJumpLabel(done);

    // here if halt is required
    vmimtInsertLabel(wait);

    // wait for event
    armEmitWait(state, AD_WFE);

    // here when done
    vmimtInsertLabel(done);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such `vmimtUncondJump`.
2. The label argument to `vmimtUncondJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction.
   If the call to `vmimtInsertLabel` precedes the call to `vmimtUncondJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtUncondJumpLabel` (as in the above example), then this is a forward jump in the node list.
3. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
```

```
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

5.19 vmimtCondJumpLabel

Prototype

```
void vmimtCondJumpLabel(
    vmiReg    flag,
    Bool      jumpIfTrue,
    vmiLabelP toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The processor flag register to test to determine whether to take the branch is specified by `flag`: this is an 8-bit register (declare it as an `Uns8` in the processor structure).

If `jumpIfTrue` is `True`, then a jump will be taken to label `toLabel` if the value of the `flag` register is non-zero and execution will continue with the next operation if the `flag` register is zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target label `toLabel` if the value of the `flag` register is zero and execution will continue with the next operation if the `flag` register is non-zero.

Example

The OVP ARM model uses this function to control whether a processor should stop in a `WFE` instruction:

```
ARM_MORPH_FN(armEmitWFE) {

    vmiLabelP wait = vmimtNewLabel();
    vmiLabelP done = vmimtNewLabel();

    // jump to wait code if no event registered
    vmimtCondJumpLabel(ARM_EVENT, False, wait);

    // clear event register and finish
    vmimtMoveRC(8, ARM_EVENT, 0);
    vmimtUncondJumpLabel(done);

    // here if halt is required
    vmimtInsertLabel(wait);

    // wait for event
    armEmitWait(state, AD_WFE);

    // here when done
    vmimtInsertLabel(done);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.

2. The label argument to `vmimtCondJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtCondJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCondJumpLabel` (as in the above example), then this is a forward jump in the node list.
3. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

5.20 vmimtCondJumpLabelFunctionResult

Prototype

```
void vmimtCondJumpLabelFunctionResult(Bool jumpIfTrue, vmiLabelP toLabel);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*. Whether the branch should be taken is determined by the value returned by an embedded function call made immediately prior to the call to `vmimtCondJumpLabelFunctionResult` – see section 7 for more information about embedded function calls.

If `jumpIfTrue` is `True`, then a jump will be taken to label `toLabel` if the embedded function call returned non-zero and execution will continue with the next operation if the embedded function call returned zero.

If `jumpIfTrue` is `False`, then a jump will be taken to target label `toLabel` if the embedded function call returned zero and execution will continue with the next operation if the embedded function call returned non-zero.

Example

The OVP MIPS model uses this function to control whether a processor should stop in a `WAIT` instruction. In this model, the state of pending interrupts is returned by a function, `mipsIsIntActiveTC`:

```
static void emitWAIT(mipsP tc) {

    mipsP     vpe    = GET_VPE(tc);
    vmiLabelP noWait = vmimtNewLabel();

    // if Config7.WII is set, the processor should not wait if there is an
    // active interrupt, even if it is not currently enabled
    if(COP0_FIELD(vpe, Config7, WII)) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)mipsIsIntActiveTC);
        vmimtCondJumpLabelFunctionResult(True, noWait);
    }

    // enter wait state
    mipsEmitHalt(tc, MD_WAIT);

    // here if a pending interrupt causes wait to be ignored
    vmimtInsertLabel(noWait);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.
2. The label argument to `vmimtCondJumpLabelFunctionResult` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction.
   If the call to `vmimtInsertLabel` precedes the call to

`vmimtCondJumpLabelFunctionResult`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCondJumpLabelFunctionResult` (as in the above example), then this is a forward jump in the node list.

3. Only use label-based jumps if *the correct branch to take is known only at run time*. If the correct branch is known at *morph time*, it is much more efficient to morph alternative code sequences instead. For example:

```
static void emitSys(Addr thisPC, Bool interceptTrap) {
    if(interceptTrap) {
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_InterceptTrap);
    } else {
        Addr nextAddress = thisPC + 4;
        vmimtArgProcessor();
        vmimtCall((vmiCallFn)vmic_EnterKernelMode);
        vmimtUncondJump(nextAddress, SYS_ADDRESS, CPUX_EPCR, vmi_JH_CALL);
    }
}
```

5.21 vmimtTestRRJumpLabel

Prototype

```
void vmimtTestRRJumpLabel(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    vmiReg       rb,
    vmiLabelP    toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The function emits code to compare registers `ra` and `rb` of size `bits`. If the condition `cond` is satisfied, control branches to `toLabel`; otherwise, execution continues with the next operation. The comparison is performed by performing a bitwise AND of the two registers.

Example

This function is not currently used in OVP models.

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.
2. The label argument to `vmimtTestRCJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtTestRCJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtTestRCJumpLabel` (as in the above example), then this is a forward jump in the node list.

5.22 vmimtTestRCJumpLabel

Prototype

```
void vmimtTestRCJumpLabel(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       r,
    Uns64        c,
    vmiLabelP    toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The function emits code to compare register `r` of size `bits` with constant `c`. If the condition `cond` is satisfied, control branches to `toLabel`; otherwise, execution continues with the next operation. The comparison is performed by performing a bitwise AND of the register and constant value.

Example

The OVP ARM model uses this function to implement a stack alignment check in AArch64 mode:

```
void armEmitCheckSA(armMorphStateP state) {

    armP           arm        = state->arm;
    armBlockStateP blockState = state->blockState;

    // emit blockMask check of SA state
    vmimtValidateBlockMask(ARM_BM_SA);

    // determine if alignment check is required
    if(!blockState->alignedSP && (arm->blockMask & ARM_BM_SA)) {

        vmiLabelP ok = vmimtNewLabel();

        // after this instruction, the stack pointer will be aligned (otherwise
        // an exception will have been taken)
        blockState->alignedSP = True;

        // skip mode switch unless stack is misaligned
        vmimtTestRCJumpLabel(64, vmi_COND_Z, ARM_SP64(0), 0xf, ok);

        // emit call to stack alignment exception routine
        vmimtArgProcessor();
        vmimtCallAttrs((vmiCallFn)armSA, VMCA_EXCEPTION);

        // here if no stack alignment exception
        vmimtInsertLabel(ok);
    }
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.

2. The label argument to `vmimtTestRCJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtTestRCJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtTestRCJumpLabel` (as in the above example), then this is a forward jump in the node list.

5.23 vmimtCompareRRJumpLabel

Prototype

```
void vmimtCompareRCJumpLabel(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       ra,
    vmiReg       rb,
    vmiLabelP    toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The function emits code to compare registers `ra` and `rb` of size `bits`. If the condition `cond` is satisfied, control branches to `toLabel`; otherwise, execution continues with the next operation. The comparison is performed by subtracting `rb` from `ra`.

Example

The OVP ARM processor model uses this in a routine to validate exclusive access addresses, as follows:

```
static vmiLabelP validateEA(
    armMorphStateP state,
    Int32          offset,
    vmiReg         ra,
    vmiReg         rd
) {
    Uns32     memBits = state->info.sz*8;
    Uns32     eaBits  = getEABits(state);
    vmiLabelP done    = vmimtNewLabel();
    vmiLabelP ok      = vmimtNewLabel();
    vmiReg    t       = getTemp(state, eaBits);

    // generate any store exception prior to exclusive access tag check
    vmimtTryStoreRC(memBits, offset, ra);

    // generate exclusive access tag for this address
    generateEATag(state, offset, t, ra);

    // do load and store tags match?
    vmimtCompareRRJumpLabel(eaBits, vmi_COND_EQ, ARM_EA_TAG, t, ok);

    // indicate store failed
    vmimtMoveRC(ARM_GPR_BITS(state), rd, 1);

    // jump to instruction end
    vmimtUncondJumpLabel(done);

    // here to commit store
    vmimtInsertLabel(ok);

    return done;
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.
2. The label argument to `vmimtCompareRCJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtCompareRCJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCompareRCJumpLabel` (as in the above example), then this is a forward jump in the node list.

5.24 vmimtCompareRCJumpLabel

Prototype

```
void vmimtCompareRCJumpLabel(
    Uns32        bits,
    vmiCondition cond,
    vmiReg       r,
    Uns64        c,
    vmiLabelP    toLabel
);
```

Description

This function is used to perform a conditional jump to a label previously defined with `vmimtNewLabel`. Labels are used to implement *intra-instruction jumps*.

The function emits code to compare register `r` of size `bits` with constant `c`. If the condition `cond` is satisfied, control branches to `toLabel`; otherwise, execution continues with the next operation. The comparison is performed by subtracting the constant from the register value.

Example

The OVP ARC processor model uses this to implement `FFS` and `FLS` instructions, as follows:

```
static void emitFFSFLSRR(arcMorphStateP state, vmiUnop op, Uns32 zValue) {

    Uns32      bits    = ARC_GPR_BITS;
    vmiReg     rd      = GET_RD(state, rd);
    vmiReg     rs1     = GET_RS(state, rs1);
    vmiFlagsCP flags   = getFlagsOrNull(state);
    vmiLabelP  nonZero = vmimtNewLabel();
    vmiLabelP  done    = vmimtNewLabel();

    // generate flags if required (based on source value)
    emitGenerateFlagsR(state, rs1, flags);

    // go if the argument is non-zero
    vmimtCompareRCJumpLabel(bits, vmi_COND_NZ, rs1, 0, nonZero);

    // special actions for zero argument
    vmimtMoveRC(bits, rd, zValue);

    // after special code
    vmimtUncondJumpLabel(done);

    // here for non-zero argument
    vmimtInsertLabel(nonZero);

    // scan for least-significant 1
    vmimtUnopRR(bits, op, rd, rs1, 0);

    // here when done
    vmimtInsertLabel(done);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.
2. The label argument to `vmimtCompareRCJumpLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` at some point during translation of the current instruction. If the call to `vmimtInsertLabel` precedes the call to `vmimtCompareRCJumpLabel`, then this is a backward jump in the node list; if the call to `vmimtInsertLabel` follows the call to `vmimtCompareRCJumpLabel` (as in the above example), then this is a forward jump in the node list.

6 Indexed and Vector Register Operations

Morph-time primitives have historically been restricted to accessing *fixed* register fields in processor structures. From VMI version 7.0.0, it is possible to access *indexed* members of vector registers or other similar structures.

This section describes functions designed to support indexed and vector register operations.

6.1 vmimtDJNZLabel

Prototype

```
void vmimtDJNZLabel(
    Uns32      bits,
    vmiReg     r,
    vmiLabelP toLabel
);
```

Description

This function is used to perform a conditional *backwards* jump to a label previously defined with `vmimtNewLabel`. Such jumps are intended to be used to implement *vector* instructions.

The function emits code to decrement register `r` of size `bits`. If the result is non-zero, control branches to `toLabel`; otherwise, execution continues with the next operation.

Example

This example shows how `vmimtDJNZLabel` can be used to implement a vector binary operation on integer data:

```
static void emitVectorOp(
    vmiBinop op,
    Uns32    bits,
    Uns32    rd,
    Uns32    ra,
    Uns32    rb
) {
    vmiLabelP repeat   = vmimtNewLabel();
    Uns32     vecSize  = sizeof(vector);
    Uns32     elemSize = bits/8;
    Uns32     elemNum  = vecSize/elemSize;
    vmiReg    base     = VR_BASE;
    vmiReg    index    = VR_INDEX;
    vmiReg    rdE      = VR(rd);
    vmiReg    raE      = VR(ra);
    vmiReg    rbE      = VR(rb);

    // prepare indexed registers
    vmimtGetIndexedRegister(&rdE, &base, vecSize);
    vmimtGetIndexedRegister(&raE, &base, vecSize);
    vmimtGetIndexedRegister(&rbE, &base, vecSize);

    // initialize repeat count
    vmimtMoveRC(32, index, elemNum);

    // loop to here
    vmimtInsertLabel(repeat);

    // do operation
    vmimtBinopRRR(bits, op, rdE, raE, rbE, 0);

    // prepare for next iteration
    vmimtAddBaseC(base, elemSize, 0);
    vmimtDJNZLabel(32, index, repeat);
}
```

Notes and Restrictions

1. Labels may only be used for intra-instruction jumps: for inter-instruction jumps, use functions such as `vmimtCondJump`.
2. The label argument to `vmimtDJNZLabel` must be inserted into the NMI list by a call to `vmimtInsertLabel` *before* calling `vmimtDJNZLabel` (these jumps may only be used to implement backwards-loop constructs).
3. The label argument to `vmimtDJNZLabel` must be referenced *only once*: it cannot be the target of another intra-instruction jump primitive.
4. `bits` can be 8, 16 or 32.

6.2 vmimtGetIndexedRegister

Prototype

```
void vmimtGetIndexedRegister(vmiReg *r, vmiReg *base, Uns32 bytes);
```

Description

This function is used to specify that the `vmiReg` descriptor addressed by `r` is to be accessed using the *base* register descriptor indexed by `base`. The full size of vector register `r` is `bytes`. The function is typically used in combination with function `vmimtDJNZLabel` to implement a vector instruction.

The full flow to use this primitive in a vector operation context is usually as follows:

1. Obtain `vmiReg` descriptors for all source and target vectors used by an instruction.
2. Use `vmimtGetIndexedRegister` to convert the `vmiReg` descriptors to *indexed* descriptors using one or more pointer-sized artifact *base* registers (declared using type `UnsPS` in the processor structure).
3. Initialize an artifact loop counter register with the number of vector elements.
4. Insert a loop label.
5. Emit operations to implement one vector operation.
6. Adjust to the base register or registers for the next iteration.
7. Use `vmimtDJNZLabel` to loop to the previously-inserted label.

It is also possible to use indexed registers in non-vector contexts. For example, they could be used to implement registers like the x87 floating point registers in an Intel IA-32 processor model, in which the registers used by an operation are identified by an index held in another control register.

Example

This example shows how indexed registers can be used to implement a vector binary operation.

```
typedef struct vectorS {
    Uns32 elem[4];
} vector;

typedef struct cpuS {

    // architectural registers
    vector vr[16];

    // artifact registers
    UnsPS vrBase;    // base pointer (must be declared as UnsPS type)
    Uns32 vrIndex;   // index register

} cpu, *cpuP;

#define CPU_REG(_F)   VMI_CPU_REG(cpuP, _F)
#define CPU_TEMP(_F)  VMI_CPU_TEMP(cpuP, _F)
#define VR(_R)        CPU_REG(vr[_R])
#define VR_BASE       CPU_TEMP(vrBase)
#define VR_INDEX      CPU_TEMP(vrIndex)

static void emitVectorOp(
```

```
    vmiBinop op,
    Uns32    bits,
    Uns32    rd,
    Uns32    ra,
    Uns32    rb
) {
    vmiLabelP repeat   = vmimtNewLabel();
    Uns32     vecSize  = sizeof(vector);
    Uns32     elemSize = bits/8;
    Uns32     elemNum  = vecSize/elemSize;
    vmiReg    base     = VR_BASE;
    vmiReg    index    = VR_INDEX;
    vmiReg    rdE      = VR(rd);
    vmiReg    raE      = VR(ra);
    vmiReg    rbE      = VR(rb);

    // prepare indexed registers
    vmimtGetIndexedRegister(&rdE, &base, vecSize);
    vmimtGetIndexedRegister(&raE, &base, vecSize);
    vmimtGetIndexedRegister(&rbE, &base, vecSize);

    // initialize repeat count
    vmimtMoveRC(32, index, elemNum);

    // loop to here
    vmimtInsertLabel(repeat);

    // do operation
    vmimtBinopRRR(bits, op, rdE, raE, rbE, 0);

    // prepare for next iteration
    vmimtAddBaseC(base, elemSize, 0);
    vmimtDJNZLabel(32, index, repeat);
}
```

Notes and Restrictions

1. Be careful to ensure that that the `base` argument size is correct: this should be the size of the *entire* target vector, in bytes. If this value is incorrect, instruction attributes will not be generated correctly (see section 13).
2. When using indexed registers, avoid using *unindexed* aliases of the same registers inside the loop for *written* registers. The simulator will not be able to detect that these are aliases, which could cause incorrect code to be generated.
3. One base register is required for each size of vector operand and result. In the example above, only one base register is required, because all operands and the result are the same size. For an operation that (for example) takes 1-byte operands and produces a 4-byte result, separate base registers will be required for the operands and the result.
4. See section 9.21 for information on the use of indexed registers in combination with compound floating point operations.

6.3 vmimtAddBaseC

Prototype

```
void vmimtAddBaseC(vmiReg base, Int32 bytes, Uns32 vectorBytes);
```

Description

This function is used to add offset `bytes` to a vector base register, of type `UnsPS`, initialized by a previous call to `vmimtGetIndexedRegister`. The `bytes` value may be positive or negative.

If argument `vectorBytes` is non-zero, then any increment or decrement of the base pointer that would take it outside this size range will result in the pointer value wrapping round to to other end of the vector. This allows modulo vector addressing to be implemented easily. If `vectorBytes` is zero, then no such range check is performed.

Example

This example shows how indexed registers can be used to implement a vector binary operation. Function `vmimtAddBaseC` is used at the loop end to prepare for the next iteration.

```
typedef struct vectorS {
    Uns32 elem[4];
} vector;

typedef struct cpuS {

    // architectural registers
    vector vr[16];

    // artifact registers
    UnsPS vrBase;    // base pointer (must be declared as UnsPS type)
    Uns32 vrIndex;   // index register

} cpu, *cpuP;

#define CPU_REG(_F)   VMI_CPU_REG(cpuP, _F)
#define CPU_TEMP(_F)  VMI_CPU_TEMP(cpuP, _F)
#define VR(_R)        CPU_REG(vr[_R])
#define VR_BASE       CPU_TEMP(vrBase)
#define VR_INDEX      CPU_TEMP(vrIndex)

static void emitVectorOp(
    vmiBinop op,
    Uns32    bits,
    Uns32    rd,
    Uns32    ra,
    Uns32    rb
) {
    vmiLabelP repeat   = vmimtNewLabel();
    Uns32     vecSize  = sizeof(vector);
    Uns32     elemSize = bits/8;
    Uns32     elemNum  = vecSize/elemSize;
    vmiReg    base     = VR_BASE;
    vmiReg    index    = VR_INDEX;
    vmiReg    rdE      = VR(rd);
    vmiReg    raE      = VR(ra);
    vmiReg    rbE      = VR(rb);

    // prepare indexed registers
```

```
    vmimtGetIndexedRegister(&rdE, &base, vecSize);
    vmimtGetIndexedRegister(&raE, &base, vecSize);
    vmimtGetIndexedRegister(&rbE, &base, vecSize);

    // initialize repeat count
    vmimtMoveRC(32, index, elemNum);

    // loop to here
    vmimtInsertLabel(repeat);

    // do operation
    vmimtBinopRRR(bits, op, rdE, raE, rbE, 0);

    // prepare for next iteration
    vmimtAddBaseC(base, elemSize, 0);
    vmimtDJNZLabel(32, index, repeat);
}
```

Notes and Restrictions

1. The base register must be declared using type `UnsPS`, and must have been previously initialized using a call to `vmimtGetIndexedRegister`.
2. See section 9.21 for information on the use of indexed registers in combination with compound floating point operations.

6.4 vmimtAddBaseR

Prototype

```
void vmimtAddBaseR(
    vmiReg base,
    vmiReg offset,
    Uns32  scale,
    Uns32  vectorBytes,
    Bool   checkWrapLow,
    Bool   checkWrapHigh
);
```

Description

This function is used to add a displacement held in register `offset` to a vector base register, initialized by a previous call to `vmimtGetIndexedRegister`. The base register is of type `UnsPS`, and the offset must be of type `IntPS`.

If argument `vectorBytes` is non-zero, then any increment or decrement of the base pointer that would take it outside this size range will result in the pointer value wrapping round to to other end of the vector. This allows modulo vector addressing to be implemented easily. Depending on the value held in register `offset`, parameters `checkWrapLow` and `checkWrapHigh` should be set as follows:

1. Value of `offset` known always to be >=0: set `checkWrapLow` to `False` and `checkWrapHigh` to `True`.
2. Value of `offset` known always to be <=0: set `checkWrapLow` to `True` and `checkWrapHigh` to `False`.
3. No known constraints on `offset`: set `checkWrapLow` to `True` and `checkWrapHigh` to `True`.

Example

This example shows how indexed registers can be used to implement a vector binary operation. Function `vmimtAddBaseR` is used at the loop end to prepare for the next iteration. The example uses two base registers: the source registers use a constant index across the vector, and the target register uses a variable offset that may either increase or decrease. In this simple example, there is no check for overlap of source registers and destination: typically, a vector temporary would be required in such cases.

```
typedef struct vectorS {
    Uns32 elem[4];
} vector;

typedef struct cpuS {

    // architectural registers
    vector vr[16];

    // artifact registers
    UnsPS vrBase[2]; // base pointers (must be declared as UnsPS type)
    IntPS vrOffset;  // byte offset (must be declared as IntPS type)
    Uns32 vrIndex;   // index register
```

```
} cpu, *cpuP;

#define CPU_REG(_F)   VMI_CPU_REG(cpuP, _F)
#define CPU_TEMP(_F)  VMI_CPU_TEMP(cpuP, _F)
#define VR(_R)        CPU_REG(vr[_R])
#define VR_BASE(_I)   CPU_TEMP(vrBase[_I])
#define VR_OFFSET     CPU_TEMP(vrOffset)
#define VR_INDEX      CPU_TEMP(vrIndex)

static void emitVectorOp(
    vmiBinop op,
    Uns32    bits,
    Uns32    rd,
    Uns32    ra,
    Uns32    rb,
    Uns32    offset,
    IntPS    stride
) {
    vmiLabelP repeat     = vmimtNewLabel();
    Uns32     vecSize    = sizeof(vector);
    Uns32     elemSize   = bits/8;
    Uns32     elemNum    = vecSize/elemSize;
    vmiReg    baseS      = VR_BASE(0);
    vmiReg    baseD      = VR_BASE(1);
    vmiReg    index      = VR_INDEX;
    vmiReg    strideR    = VR_OFFSET;
    vmiReg    rdE        = VR(rd);
    vmiReg    raE        = VR(ra);
    vmiReg    rbE        = VR(rb);
    Uns32     strideBits = IMPERAS_POINTER_BITS;

    // use scale instead of stride
    Int32 scale = (stride>0) ? stride : -stride;
    stride = stride/scale;

    // prepare indexed registers
    vmimtGetIndexedRegister(&rdE, &baseD, vecSize);
    vmimtGetIndexedRegister(&raE, &baseS, vecSize);
    vmimtGetIndexedRegister(&rbE, &baseS, vecSize);

    // initialize pseudo-variable stride
    vmimtMoveRC(strideBits, strideR, elemSize*stride);

    // arbitrary base offset
    vmimtAddBaseC(baseS, offset, 0);

    // initialize repeat count
    vmimtMoveRC(32, index, elemNum);

    // loop to here
    vmimtInsertLabel(repeat);

    // do operation
    vmimtBinopRRR(bits, op, rdE, raE, rbE, 0);

    // prepare for next iteration
    vmimtAddBaseC(baseD, elemSize, 0);
    vmimtAddBaseR(baseS, strideR, scale, vecSize, stride<0, stride>0);

    vmimtDJNZLabel(32, index, repeat);
}
```

Notes and Restrictions

1. The base register must be declared using type `UnsPS`, and must have been previously initialized using a call to `vmimtGetIndexedRegister`.
2. See section 9.21 for information on the use of indexed registers in combination with compound floating point operations.

6.5 vmimtGetBaseOffset

Prototype

```
void vmimtGetBaseOffset(Uns32 bits, vmiReg offset, vmiReg base);
```

Description

Given a `base` register previously configured using `vmimtGetIndexedRegister`, this function fills register `offset` of size `bits` with the current displacement of the base register, in bytes. In other words, `offset` is filled with the cumulative total of all the offsets added to the base by functions `vmimtAddBaseC` and `vmimtAddBaseR`, allowing for any wrapping.

Example

The OVP ARM model uses this function to implement SVE comparison operations. For these operations, a bit is set in a result register corresponding to the offset of the elements being compared from the vector base.

```
static ARM_SVE_PER_ELEM_FN(emitSVE_CMP_IV) {

    vmiLabelP    zero  = vmimtNewLabel();
    vmiCondition cond  = state->attrs->cond ^ 1;
    Uns32        pBits = getPBits(state->arm);
    vmiReg       t1    = getTemp(state, 32);

    // skip bit update if condition is False
    if(VMI_ISNOREG(elem[1])) {

        // immediate variant
        Int32 imm = state->info.c;
        vmimtCompareRCJumpLabel(elemBits, cond, elem[0], imm, zero);

    } else {

        // vectors variant
        vmimtCompareRRJumpLabel(elemBits, cond, elem[0], elem[1], zero);
    }

    // get iteration index
    vmimtGetBaseOffset(32, t1, info->base[0]);

    // set bit in result
    vmimtBitopVR(pBits, 32, vmi_BTS, ARM_Z_TMP, t1, VMI_NOREG);

    // here if condition False
    emitLabel(zero);
}
```

Notes and Restrictions

1. The base register must be declared using type `UnsPS`, and must have been previously initialized using a call to `vmimtGetIndexedRegister`.
2. `bits` must be 8, 16, 32 or 64.

6.6 vmimtZeroRV

Prototype

```
void vmimtZeroRV(
    Uns32         destMaxBits,
    vmiReg        rd,
    Uns32         countBits,
    vmiReg        count,
    Int32         countInc,
    Uns32         countScale,
    vmiCheckCount checkCount
);
```

Description

Emit code to write zeros to part of a variable-sized register `rd`. The *maximum* size of `rd` is given by `destMaxBits`. The *effective* size of `rd`, in bytes, is given by:

```
(count + countInc) * countScale
```

Where the contents of register `count` (of size `countBits`) can vary at run time. As an example, if the current value of `count` is 3, `countInc` is -1 and `countScale` is 16 then the operation will fill the first `(3-1)*16` = `32` bytes of `rd` with zeros, leaving remaining bytes of `rd` unchanged.

Argument `checkCount` indicates constraints on the initial value of the effective size (the result of the expression above):

```
typedef enum vmiCheckCountE {
    vmi_CC_NONE,        // no count check required
    vmi_CC_EQ_ZERO,     // count check for zero required
    vmi_CC_LE_ZERO,     // count check for zero or negative required
} vmiCheckCount;
```

Value `vmi_CC_NONE` indicates that the initial value will always be non-zero and positive; this option generates the most efficient JIT-compiled code. Value `vmi_CC_EQ_ZERO` indicates that the initial value will always be either positive or zero, but the operation should only have effect for non-zero values. Value `vmi_CC_LE_ZERO` indicates that the initial value could have any value, including zero or negative, but the operation should only have effect for positive non-zero values.

This function is typically used when implementing vector instructions. Often, these will force to zero parts of registers in a way that depends on the current effective vector size, which can vary dynamically.

Example

The OVP ARM model uses this function to zero-extend SVE Z registers after an SIMD instruction:

```
static void extendVToZ(armMorphStateP state, vmiReg rd) {

    armP arm = state->arm;
```

```
    if(arm->checkSVEVL) {

        // start extension after V register
        Uns32 zBits  = getZBits(arm);
        Uns32 vBits  = 128;
        Uns32 vBytes = vBits/8;

        // start extension after V register
        rd = VMI_REG_DELTA(rd,vBytes);

        vmimtZeroRV(zBits-vBits, rd, 32, ARM_VLM1, 0, vBytes, vmi_CC_EQ_ZERO);
    }
}
```

Notes and Restrictions

1. `destMaxBits` must be a multiple of 8.
2. `countBits` must be 8, 16 or 32.

6.7 vmimtMoveRRV

Prototype

```
void vmimtMoveRRV(
    Uns32         destMaxBits,
    vmiReg        rd,
    vmiReg        ra,
    Uns32         countBits,
    vmiReg        count,
    Int32         countInc,
    Uns32         countScale,
    vmiCheckCount checkCount
);
```

Description

Emit code to move the first part of variable-sized register `ra` to the first part of variable-sized register `rd`. The *maximum* size of `rd` and `ra` is given by `destMaxBits`. The *effective* size of `rd` and `ra`, in bytes, is given by:

```
(count + countInc) * countScale
```

Where the contents of register `count` (of size `countBits`) can vary at run time. As an example, if the current value of `count` is 3, `countInc` is -1 and `countScale` is 16 then the operation will copy the first `(3-1)*16` = `32` bytes of `ra` to the first 32 bytes of `rd`, leaving remaining bytes of `rd` unchanged.

Argument `checkCount` indicates constraints on the initial value of the effective size (the result of the expression above):

```
typedef enum vmiCheckCountE {
    vmi_CC_NONE,        // no count check required
    vmi_CC_EQ_ZERO,     // count check for zero required
    vmi_CC_LE_ZERO,     // count check for zero or negative required
} vmiCheckCount;
```

Value `vmi_CC_NONE` indicates that the initial value will always be non-zero and positive; this option generates the most efficient JIT-compiled code. Value `vmi_CC_EQ_ZERO` indicates that the initial value will always be either positive or zero, but the operation should only have effect for non-zero values. Value `vmi_CC_LE_ZERO` indicates that the initial value could have any value, including zero or negative, but the operation should only have effect for positive non-zero values.

This function is typically used when implementing vector instructions. Often, these will copy parts of registers in a way that depends on the current effective vector size, which can vary dynamically.

Example

The OVP ARM model uses this function to commit results of SVE instructions:

```
static void emitSVE_LDR(armMorphStateP state, vmiReg rd, Uns32 elemBytes) {
```

```
    Uns32       bits     = ARM_GPR_BITS(state);
    Uns32       elemBits = elemBytes*8;
    Uns32       rdBits   = Z_REG_SIZE(state->arm)*elemBits;
    vmiReg      ra       = emitAddressMulXL(state, elemBytes);
    vmiReg      result   = ARM_Z_TMP;
    sveIterInfo info;

    // start SVE iteration (VLx1, size elemBytes)
    startSVEIter1(state, &info, 0,1, elemBytes);

    // prepare indexed registers
    vmiReg rdI = getIndexedP(result, &info, 0);

    // insert repeat label
    vmimtInsertLabel(info.repeat);

    // do element load
    vmimtLoadRRO(elemBits, elemBits, 0, rdI, ra, False, False);

    // prepare for next iteration
    vmimtBinopRC(bits, vmi_ADD, ra, elemBytes, 0);

    // end SVE iteration
    endSVEIter(state, &info);

    // commit result (if no exception)
    vmimtMoveRRV(rdBits, rd, result, 32, ARM_VL, 0, elemBytes, vmi_CC_EQ_ZERO);

    // free temporaries
    freeTemp(state, bits);
}
```

Notes and Restrictions

1. `destMaxBits` must be a multiple of 8.
2. `countBits` must be 8, 16 or 32.
3. Registers `ra` and `rd` must not overlap.

6.8 vmimtBitopVR

Prototype

```
void vmimtBitopVR(
    Uns32    vBits,
    Uns32    iBits,
    vmiBitop op,
    vmiReg   rv,
    vmiReg   ri,
    vmiReg   set
);
```

Description

Given a vector register `rv` of size `vBits`, this function emits code to operate on bit `ri` of that vector. The operation is defined by the `vmiBitop` enumeration in `vmiTypes.h`:

```
typedef enum vmiBitopE {
    vmi_BT         = OCL_BIT_BT,        // bit test
    vmi_BTR        = OCL_BIT_BTR,       // bit test and reset
    vmi_BTS        = OCL_BIT_BTS,       // bit test and set
    vmi_BTC        = OCL_BIT_BTC,       // bit test and complement
    vmi_BITOP_LAST = OCL_BIT_LAST,      // KEEP LAST
} vmiBitop;
```

Operation `vmi_BT` tests bit `ri` in `rv`, and writes the current value of that bit to the byte-sized result register `set`.

Operation `vmi_BTR` resets bit `ri` in `rv`, and writes the previous value of that bit to the byte-sized result register `set`.

Operation `vmi_BTS` sets bit `ri` in `rv`, and writes the previous value of that bit to the byte-sized result register `set`.

Operation `vmi_BTC` toggles the value of bit `ri` in `rv`, and writes the previous value of that bit to the byte-sized result register `set`.

The vector register `rv` can be of any size that is a multiple of 8 bits. Bits selected by `ri` are numbered increasing from the least-significant bit of the first byte of the vector. For example, an index of 100 selects bit 4 of byte 12 of the vector (where the least-significant byte is byte 0).

This function is typically used when implementing vector instructions, to set bits in predicate registers based on the result of a test on vector registers.

Example

The OVP ARM model uses this function to implement SVE comparison instructions:

```
static ARM_SVE_PER_ELEM_FN(emitSVE_CMP_I) {

    vmiLabelP    zero    = vmimtNewLabel();
    vmiCondition cond    = state->attrs->cond ^ 1;
```

```
    Int32        imm     = state->info.c;
    Uns32        pBits   = getPBits(state->arm);
    vmiReg       iterNum = getTemp(state, IMPERAS_POINTER_BITS);

    // skip bit update if condition is False
    vmimtCompareRCJumpLabel(elemBits, cond, elem[0], imm, zero);

    // get iteration index
    getSVEElemOffset(state, info, iterNum);

    // set bit in result
    vmimtBitopVR(pBits, 32, vmi_BTS, ARM_Z_TMP, iterNum, VMI_NOREG);

    // here if condition False
    emitLabel(zero);
}
```

Notes and Restrictions

1. `vBits` must be a multiple of 8.
2. `iBits` must be 8, 16 or 32.

6.9 vmimtTestBitVRJumpLabel

Prototype

```
void vmimtTestBitVRJumpLabel (
    Uns32     vBits,
    Uns32     iBits,
    Bool      jumpIfSet,
    vmiReg    rv,
    vmiReg    ri,
    vmiLabelP toLabel
);
```

Description

Given a vector register `rv` of size `vBits`, this function emits code to test bit `ri` of that vector. If `jumpIfSet` is `True`, it will emit code to jump to label `toLabel` if the bit is `1`. If `jumpIfSet` is `False`, it will emit code to jump to label `toLabel` if the bit is `0`.

The vector register `rv` can be of any size that is a multiple of 8 bits. Bits selected by `ri` are numbered increasing from the least-significant bit of the first byte of the vector. For example, an index of 100 selects bit 4 of byte 12 of the vector (where the least-significant byte is byte 0).

This function is typically used when implementing vector instructions. Often, these have some form of predication, in which operations are only applied to particular members of a data register in a corresponding bit in a predicate register is set or clear.

Example

The OVP ARM model uses this function to enable predicated instructions:

```
static void doPTest(armMorphStateP state, svePSelInfoP info) {

    Uns32 pBits = getPBits(state->arm);

    vmimtTestBitVRJumpLabel(
        pBits, 32, False, GET_P(state, r2), info->pindex, info->pbit0
    );
}
```

Notes and Restrictions

1. `vBits` must be a multiple of 8.
2. `iBits` must be 8, 16 or 32.

7 Embedded Native Call Operations

This section describes emission functions for embedding model function calls within translated native code. This technique is especially useful when there are no suitable `vmimt`-prefixed routines to implement required functionality: it provides a generic extension capability.

To embed a call to a model function:

1. Use the functions with the `vmimtArg` prefix to specify the arguments to the embedded call;
2. Use function `vmimtCallResultAttrs` to specify the function to call (and possibly what should happen to any function result).

7.1 vmimtArgProcessor

Prototype

```
void vmimtArgProcessor(void);
```

Description

`vmimtArgProcessor` specifies that the *current processor handle* should be passed as an argument to an embedded function call. Processor fields can then be accessed directly within the callback.

Example

The OVP ARM model uses this function to implement a stack alignment check in AArch64 mode:

```
void armSA(armP arm) {

    Uns64 thisPC = getPC(arm);

    // fill exception details
    doMisalignedSP(arm, thisPC);
}

void armEmitCheckSA(armMorphStateP state) {

    armP           arm        = state->arm;
    armBlockStateP blockState = state->blockState;

    // emit blockMask check of SA state
    vmimtValidateBlockMask(ARM_BM_SA);

    // determine if alignment check is required
    if(!blockState->alignedSP && (arm->blockMask & ARM_BM_SA)) {

        vmiLabelP ok = vmimtNewLabel();

        // after this instruction, the stack pointer will be aligned (otherwise
        // an exception will have been taken)
        blockState->alignedSP = True;

        // skip mode switch unless stack is misaligned
        vmimtTestRCJumpLabel(64, vmi_COND_Z, ARM_SP64(0), 0xf, ok);

        // emit call to stack alignment exception routine
        vmimtArgProcessor();
        vmimtCallAttrs((vmiCallFn)armSA, VMCA_EXCEPTION);

        // here if no stack alignment exception
        vmimtInsertLabel(ok);
    }
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`armSA`, in the above example).

7.2 vmimtArgUns32

Prototype

```
void vmimtArgUns32(Uns32 arg);
```

Description

`vmimtArgUns32` specifies that a *32-bit unsigned value* (`Uns32`) should be passed as an argument to an embedded function call.

Example

The OVP ARMM model uses this function to implement the `VMSR` instruction:

```
void armWriteFPSCR(armP arm, Uns32 newValue, Uns32 writeMask) {

    Uns32 oldValue = FPSCR_REG(arm);

    // update raw register
    FPSCR_REG(arm) = ((oldValue & ~writeMask) | (newValue & writeMask));

    . . . lines omitted for clarity . . .
}

ARM_MORPH_FN(armEmitVMSR) {

    Uns32 bits = ARM_GPR_BITS;

    if(executeFPCheck(state)) {

        vmimtArgProcessor(state);
        vmimtArgReg(bits, GET_RS(state, r1));
        vmimtArgUns32(FPSCR_MASK);
        vmimtCall((vmiCallFn)armWriteFPSCR);

        // terminate the code block (block masks or floating point
        // mode may have changed)
        vmimtEndBlock();
    }
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`armWriteFPSCR`, in the above example).

7.3 vmimtArgUns64

Prototype

```
void vmimtArgUns64(Uns64 arg);
```

Description

`vmimtArgUns64` specifies that a *64-bit unsigned value* (`Uns64`) should be passed as an argument to an embedded function call.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype.

7.4 vmimtArgFlt64

Prototype

```
void vmimtArgFlt64(Flt64 arg);
```

Description

`vmimtArgFlt64` specifies that an `Flt64` (64-bit floating point) should be passed as an argument to an embedded function call.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`vmic_TestCall`, in the above example).

7.5 vmimtArgReg

Prototype

```
void vmimtArgReg(vmiRegArgType argType, vmiReg r);
```

Description

`vmimtArgReg` specifies that the value of a processor register `r` should be passed as an argument to an embedded function call. The type of the register is given by the `argType` argument, defined in `vmiTypes.h` as follows:

```
typedef enum vmiRegArgTypeE {

    VPRRAT_8     =  8,              // 8-bit register argument
    VPRRAT_16    =  16,             // 16-bit register argument
    VPRRAT_32    =  32,             // 32-bit register argument
    VPRRAT_64    =  64,             // 64-bit register argument

    VPRRAT_FLT   = 0x80000000,      // floating-point argument identifier
    VPRRAT_FLT64 = 64|VPRRAT_FLT    // 64-bit floating point

} vmiRegArgType;
```

The argument type can be an 8-bit, 16-bit, 32-bit or 64-bit integer register or a floating-point register in double-precision format. Other parameter types can be passed *by reference* using function `vmimtArgRegP` if required.

Example

The OVP ARMM model uses this function to implement the `VMSR` instruction:

```
void armWriteFPSCR(armP arm, Uns32 newValue, Uns32 writeMask) {

    Uns32 oldValue = FPSCR_REG(arm);

    // update raw register
    FPSCR_REG(arm) = ((oldValue & ~writeMask) | (newValue & writeMask));

    . . . lines omitted for clarity . . .
}

ARM_MORPH_FN(armEmitVMSR) {

    Uns32 bits = ARM_GPR_BITS;

    if(executeFPCheck(state)) {

        vmimtArgProcessor(state);
        vmimtArgReg(bits, GET_RS(state, r1));
        vmimtArgUns32(FPSCR_MASK);
        vmimtCall((vmiCallFn)armWriteFPSCR);

        // terminate the code block (block masks or floating point
        // mode may have changed)
        vmimtEndBlock();
    }
}
```

Notes and Restrictions

1. In versions of the VMI API prior to 4.2.0, the first argument to this function was a number of bits (8, 16, 32 or 64). The function prototype has been enhanced to allow floating point operands to be explicitly identified, required for 64-bit host support.
2. Argument types other than those listed above can be passed by reference using function `vmimtArgRegP` if required.
3. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`armWriteFPSCR`, in the above example).

7.6 vmimtArgRegP

Prototype

```
void vmimtArgRegP(vmiRegPArgUsage usage, Uns32 bits, vmiReg r);
```

Description

`vmimtArgRegP` specifies that processor register `r` should be passed *by reference* as an argument to an embedded function call – in other words, the embedded call is passed *a pointer* to processor register `r`. This is useful when, for example, the register argument is wider than the maximum width supported by function `vmimtArgReg` (64 bits), or when an embedded call writes *more than one* result register.

The size of the register being accessed is given by the `bits` argument, which can be any multiple of 8. How the register is used in the embedded call is described by the `usage` argument, defined in `vmiTypes.h` as follows:

```
typedef enum vmiRegPArgUsageE {
    VPRAU_R  = 0x1,                     // argument is read
    VPRAU_W  = 0x2,                     // argument is written
    VPRAU_RW = (VPRAU_R|VPRAU_W),       // argument is read and written
} vmiRegPArgUsage;
```

A `usage` of `VPRAU_R` should be used for any argument that is a pure input (*read* by the embedded call, but not *written*).

A `usage` of `VPRAU_W` should be used for any argument that is a pure output (*written* in its entirety by the embedded call, but not *read*).

A `usage` of `VPRAU_RW` should be used for any argument that is not a pure input or pure output – this includes any argument that is both read *and* written, and also any argument that is only *partially* written (some bytes of the register are written, but others are left untouched).

It is important for correct operation that the usage of each argument is correctly described.

Example

Some AES cryptographic functions operate on 128-bit data; for example, the AES `ShiftRows` operation permutes the columns of a 128-bit argument in a fixed pattern. The prototype of this function could be written in one of two ways:

```
typedef struct {
    Uns8 bytes[16];
} AES128;

static void AESShiftRows1(AES128 *rd, AES128 *rs) {
    // assign rd with shifted rows of rs
}

static void AESShiftRows2(AES128 *r) {
```

```
    // modify r in place (r is both read and written)
}
```

For each of these functions, embedded calls could be created like this:

```
static void emitAESShiftRows1(vmiReg rd, vmiReg rs) {
    vmimtArgRegP(VPRAU_W, 128, rd);
    vmimtArgRegP(VPRAU_R, 128, rs);
    vmimtCall((vmiCallFn)AESShiftRows1);
}

static void emitAESShiftRows2(vmiReg r) {
    vmimtArgRegP(VPRAU_RW, 128, r);
    vmimtCall((vmiCallFn)AESShiftRows2);
}
```

Notes and Restrictions

1. See also function `vmimtArgReg` which allows passing of 8, 16, 32 and 64-bit arguments directly.
2. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype.

7.7 vmimtArgRegSimAddress

Prototype

```
void vmimtArgRegSimAddress(Uns32 bits, vmiReg r);
```

Description

`vmimtArgRegSimAddress` specifies that the value of a processor register `r` should be passed as an argument to an embedded function call. The register is of size `bits`, but it should be zero-extended to the size of the `Addr` type when passed as a function argument. This function is useful because many of the run time callbacks (defined in `vmiRt.h`) take generic `Addr` arguments to specify an address. The `Addr` type is 64-bit, but addresses in processor registers may well be less than this (32-bits or less).

The function may also be used to specify a register argument that should be extended to 64 bits even if that argument is not an address – see the example below.

Example

The OVP RISC-V model uses this function to implement the `CSR` write callbacks. All such callbacks take a 64-bit value argument, even if the processor is only operating in 32-bit mode:

```
riscvArchitecture riscvEmitCSRWrite(
    riscvCSRId id,
    riscvP     riscv,
    vmiReg     rs,
    vmiReg     tmp
) {
    riscvArchitecture arch    = riscv->currentArch;
    csrAttrsCP        attrs   = &csrs[id];
    Uns32             bits    = riscvGetXlenMode(riscv);
    riscvCSRWriteFn   writeCB = getCSRWriteCB(id, riscv, bits);
    vmiReg            raw     = getRawArch(attrs, arch);
    Uns64             mask    = getCSRWriteMask(attrs, riscv);

    // indicate that this register has been written
    vmimtRegWriteImpl(attrs->name);

    if(writeCB) {

        // if CSR is implemented externally, mirror the result into any raw
        // register in the model (otherwise discard the result)
        if(!csrImplementExternalWrite(id, riscv)) {
            raw = VMI_NOREG;
        }

        // emit code to call the write function (NOTE: argument is always 64
        // bits, irrespective of the architecture size)
        vmimtArgUns32(id);
        vmimtArgProcessor();
        vmimtArgRegSimAddress(bits, rs);
        vmimtCallResult((vmiCallFn)writeCB, bits, raw);

        // terminate the current block if required
        if(attrs->wEndBlock) {
            vmimtEndBlock();
        }

    } else if(VMI_ISNOREG(raw)) {
```

```
        // emit warning for unimplemented CSR
        emitWarnUnimplementedCSR(id, riscv);

    } else if(mask==-1) {

        // new value is written unmasked
        vmimtMoveRR(bits, raw, rs);

    } else if(mask) {

        // apparent reads of register below are artifacts only
        vmimtRegNotReadR(bits, raw);

        // new value is written masked
        vmimtBinopRC(bits, vmi_ANDN, raw, mask, 0);
        vmimtBinopRRC(bits, vmi_AND, tmp, rs, mask, 0);
        vmimtBinopRR(bits, vmi_OR, raw, tmp, 0);
    }

    // return architectural constraints that apply to this register
    return attrs->arch;
}
```

Notes and Restrictions

1. `bits` may be 8, 16, 32 or 64.
2. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype.

7.8 vmimtArgSimAddress

Prototype

```
void vmimtArgSimAddress(Addr arg);
```

Description

`vmimtArgSimAddress` specifies that the address `arg` should be passed as an argument to an embedded function call. The Addr type is 64 bits wide.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype.

7.9 vmimtArgSimPC

Prototype

```
void vmimtArgSimPC(Uns32 bits);
```

Description

`vmimtArgSimPC` specifies that the current simulated program counter should be passed as an argument to an embedded function call.

If a processor model does not use physically-mapped code dictionaries, then this is equivalent to using `vmimtArgUns32` or `vmimtArgUns64`, specifying the current program counter as the constant argument. However, when processor models do use physically-mapped code dictionaries, `vmimtArgSimPC` **must** be used to obtain the current simulated address, because the same JIT-compiled code block can be mapped at *different* simulated addresses.

See the description of `vmirtAliasMemoryVM` in the *VMI Run Time Function Reference* and also the *Imperas Processor Modeling Guide* for more information about physically-mapped code dictionaries.

Example

The OVP ARM model uses this function when emitting an embedded call implementing the `HVT` instruction:

```
void armEmitHVT(armMorphStateP state, const char *reason, Uns32 syndrome) {
    vmimtArgProcessor();
    vmimtArgSimPC(64);
    vmimtArgUns32(syndrome);
    vmimtArgNatAddress(reason);
    vmimtCallAttrs((vmiCallFn)armHVT, VMCA_EXCEPTION);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.

7.10 vmimtArgNatAddress

Prototype

```
void vmimtArgNatAddress(void *arg);
```

Description

`vmimtArgNatAddress` specifies that the void pointer `arg` should be passed as an argument to an embedded function call.

Example

The OVP ARM model uses this function when emitting an embedded call implementing the `HVT` instruction:

```
void armEmitHVT(armMorphStateP state, const char *reason, Uns32 syndrome) {
    vmimtArgProcessor();
    vmimtArgSimPC(64);
    vmimtArgUns32(syndrome);
    vmimtArgNatAddress(reason);
    vmimtCallAttrs((vmiCallFn)armHVT, VMCA_EXCEPTION);
}
```

Notes and Restrictions

1. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype (`armHVT`, in the above example).

7.11 vmimtCall, vmimtCallResult, vmimtCallAttrs, vmimtCallResultAttrs

Prototype

```
void vmimtCallResultAttrs(
    vmiCallFn    arg,
    Uns32        bits,
    vmiReg       rd,
    vmiCallAttrs attrs
);

#define vmimtCall(_ARG) \
    vmimtCallResultAttrs(_ARG, 0, VMI_NOREG, VMCA_NA)

#define vmimtCallResult(_ARG, _BITS, _RD) \
    vmimtCallResultAttrs(_ARG, _BITS, _RD, VMCA_NA)

#define vmimtCallAttrs(_ARG, _ATTRS) \
    vmimtCallResultAttrs(_ARG, 0, VMI_NOREG, _ATTRS)
```

Description

`vmimtCallResultAttrs` emits an *embedded call* to the function specified as an argument. The argument `arg` has type `vmiCallFn`:

```
typedef void (*vmiCallFn)(void);
```

In reality, the argument can be any function type that matches the arguments previously created by calls to functions with the `vmimtArg` prefix, and it will be necessary to cast the function to type `vmiCallFn`.

The return value from the embedded function is stored in processor register `rd` which has size `bits`. If `rd` is `VMI_NOREG`, any function result is discarded and argument `bits` is ignored. If more than one result register is updated, function `vmimtArgRegP` can be used to specify by-reference result registers if required.

The `attrs` argument is used to provide information to the JIT code translation engine about the called function, enabling it to emit better code in some circumstances. The `vmiCallAttrs` type is defined in vmiTypes.h as follows:

```
typedef enum vmiCallAttrsE {
    VMCA_NA            = 0x0,   // no attributes
    VMCA_PURE          = 0x1,   // this call is to a pure function
    VMCA_EXCEPTION     = 0x2,   // this call causes a simulated exception
    VMCA_NO_INVALIDATE = 0x4,   // this call cannot invalidate this block
    VMCA_FP_RESTORE    = 0x8,   // restore floating point state before call
} vmiCallAttrs;
```

Members of the enumeration have the following meaning:

`VMCA_NA`

This value indicates that the code generator can make no assumptions about the called function: it could update any system state. Consequently, the code generator must ensure that all simulated processor state is consistent before making the call (with the exception of floating point control state, which must be explicitly restored if required - see **VMCA_FP_RESTORE** below).

VMCA_PURE
This value indicates that the function is a *pure function*: this means that all inputs are defined by preceding `vmimtArg*` calls and that the function returns a calculated value with no other side effects. The JIT code generator is free to optimize around a pure function call because it knows what state is used and affected by that call.

VMCA_EXCEPTION
This value indicates that the called function causes a simulated exception to be immediately taken so that the function does not return (within the called function, there is a call to a function such as `vmirtSetPCException`). The JIT code generator can safely eliminate any operations following the embedded call because they are unreachable.

VMCA_NO_INVALIDATE
This value indicates that the called function cannot cause the current code block to be invalidated. This typically means that it does not call virtual memory manipulation functions (for example, `vmirtAliasMemory`) or any other functions that could flush processor code dictionaries (for example, `vmirtFlushDict`).

VMCA_FP_RESTORE
This value indicates that native floating point control state should be restored to its default value prior to making the call (usually, all interrupts disabled and rounding mode *nearest*). For performance reasons, the JIT code generator emits code that modifies the default native floating point control state as it runs. If control state is not restored, any floating point operations performed by the called function will use the currently-active floating point state, which may not be what is required. Restoration of floating point state in this way is only necessary for function calls emitted by `vmimtCallResultAttrs` in a processor model: floating point state is automatically restored before function calls emitted by intercept libraries. Note that restoration of floating point state is relatively expensive, so should only be done if it is known that the called function requires the default floating point state in order to run correctly.

Macros `vmimtCall`, `vmimtCallResult` and `vmimtCallAttrs` are wrappers for `vmimtCallResultAttrs` which allow attributes or result to be omitted.

Example

The OVP ARM model uses this function when emitting an embedded call implementing CRC operations, each of which is modeled using an embedded call. These functions are *pure*, because the calls can be eliminated entirely if the operation results are not required (they have no side effects):

```
static void emitCRC32Common(armMorphStateP state, vmiCallFn cb, Uns32 argBits) {

    Uns32  accBits = 32;
    vmiReg rd      = GET_RD(state, r1);
    vmiReg rn      = GET_RS(state, r2);
```

```
    vmiReg rm      = GET_RS(state, r3);

    // emit embedded call to perform operation
    vmimtArgReg(accBits, rn);
    vmimtArgReg(argBits, rm);
    vmimtCallResultAttrs(cb, accBits, rd, VMCA_PURE);
}
```

Notes and Restrictions

1. `bits` may be 8, 16, 32 or 64.
2. There is no automatic verification that the arguments supplied to the function match the prototype of that function: take great care that the sequence of `vmimtArg`-prefixed functions exactly matches the function prototype.

8 Connection Operations

Processor models may have *connections* associated with them. Connections are used to implement direct communication channels between processors. These communication channels allow the processors to communicate without sharing memory. Currently, the only form of connection object supported is a FIFO queue.

Section 4.4 in the *VMI Run Time Function Reference* describes functions that are used to create FIFO connections between processors and set the values of `cpux->inputConn` and `cpux->outputConn`. This section describes routines that are used to send and receive data using connection objects.

8.1 vmimtConnGetRB

Prototype

```
void vmimtConnGetRB(
    Uns32           bits,
    vmiReg          rd,
    vmiReg          connReg,
    Bool            peek,
    vmiConnUpdateFn updateCB
);
```

Description

This function emits code to perform a blocking read from a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to read has width `bits` and should be assigned to register `rd`. If `peek` is `False`, the value will be removed from the container; otherwise, it will be copied from the container.

In the case that the input connection container is empty prior to the attempted read, the processor will stop executing. It will remain stopped until some other processor writes to the container object using `vmimtConnPutRB` (or a related function). When this happens, the callback function `updateCB` is called, which determines how the waiting processor should respond: typically, the response should be to restart the waiting processor using either `vmirtRestartNow` or `vmirtRestartNext`. Upon restart, the current simulated instruction will be restarted, with the effect that the processor will retry the connection read.

Parameter `updateCB` may be `NULL`. In this case, behavior on restart will be as if function `vmirtRestartNow` has been called.

Example

The OVP OR1K training model uses this function. See the *Imperas Processor Modeling Guide* for more details.

```
static void morphConnGetOrBlock(vmiReg rd, Bool peek) {
    vmimtMoveRC(8, OR1K_BLOCK_STATE, OR1K_BS_INPUT);
    vmimtConnGetRB(OR1K_BITS, rd, OR1K_CPU_REG(inputConn), peek, 0);
    vmimtMoveRC(8, OR1K_BLOCK_STATE, OR1K_BS_NONE);
}
```

Notes and Restrictions

1. See also section 14 in the *VMI Run Time Function Reference* which describes a run time read from a connection object.

8.2 vmimtConnGetRNB

Prototype

```
void vmimtConnGetRNB(
    Uns32  bits,
    vmiReg rd,
    vmiReg connReg,
    Bool   peek,
    vmiReg flag
);
```

Description

This function emits code to perform a nonblocking read from a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to read has width `bits` and should be assigned to register `rd`. If `peek` is `False`, the value will be removed from the container; otherwise, it will be copied from the container.

In the case that the input connection container is empty prior to the attempted read, the 8-bit processor register `flag` is assigned the value 0 and `rd` is unchanged; otherwise, `flag` is assigned the value 1.

Example

The OVP OR1K training model uses this function. See the *Imperas Processor Modeling Guide* for more details.

```
static void morphConnGet(vmiReg rd, Bool peek) {
    vmimtConnGetRNB(
        OR1K_BITS, rd, OR1K_CPU_REG(inputConn), peek, OR1K_CARRY
    );
}
```

Notes and Restrictions

1. See also Section 14 in *VMI Run Time Function Reference* which describes a run time read from a connection object.

8.3 vmimtConnPutRB

Prototype

```
void vmimtConnPutRB(
    Uns32            bits,
    vmiReg           connReg,
    vmiReg           ra,
    vmiConnUpdateFn updateCB
);
```

Description

This function emits code to perform a blocking write to a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to write has width `bits` and should obtained from register `ra`.

In the case that the input connection container is full prior to the attempted write, the processor will stop executing. It will remain stopped until some other processor reads from the container object using `vmimtConnGetRB` (or a related function) to make space in the container. When this happens, the callback function `updateCB` is called, which determines how the waiting processor should respond: typically, the response should be to restart the waiting processor using either `vmirtRestartNow` or `vmirtRestartNext`. Upon restart, the current simulated instruction will be restarted, with the effect that the processor will retry the connection write.

Parameter `updateCB` may be `NULL`. In this case, behavior on restart will be as if function `vmirtRestartNow` has been called.

Example

The OVP OR1K training model uses this function. See the *Imperas Processor Modeling Guide* for more details.

```
static void morphConnPutOrBlock(vmiReg rb) {
    vmimtMoveRC(8, OR1K_BLOCK_STATE, OR1K_BS_OUTPUT);
    vmimtConnPutRB(OR1K_BITS, OR1K_CPU_REG(outputConn), rb, 0);
    vmimtMoveRC(8, OR1K_BLOCK_STATE, OR1K_BS_NONE);
}
```

Notes and Restrictions

1. See also section 14 in the *VMI Run Time Function Reference* which describes a run time write to a connection object.

8.4 vmimtConnPutRNB

Prototype

```
void vmimtConnPutRNB(
    Uns32  bits,
    vmiReg connReg,
    vmiReg ra,
    vmiReg flag
);
```

Description

This function emits code to perform a nonblocking write to a connection container object specified by the processor pseudo-register `connReg`, which must previously have been initialized. The data value to write has width `bits` and should obtained from register `ra`.

In the case that the output connection container is full prior to the attempted write, the 8-bit processor register `flag` is assigned the value 0 and the value is not written; otherwise, `flag` is assigned the value 1.

Example

```
static void morphConnPut(vmiReg rb) {
    vmimtConnPutRNB(
        OR1K_BITS, OR1K_CPU_REG(outputConn), rb, OR1K_CARRY
    );
}
```

Notes and Restrictions

1. See also section 14 in the *VMI Run Time Function Reference* which describes a run time write to a connection object.

9 Floating Point Operations

In general, modeling of floating point operations is hard. Although many processors claim to be IEEE Standard 754 compliant, there are usually implementation details that deviate from the Standard in some respects; for example, many processors implement variants of flush-to-zero mode (FZ) or denormals-are-zero mode (DAZ) which are not covered in the Standard and are inconsistently implemented in different hardware.

The VMI API has been designed so that a spectrum of implementation approaches is available for a particular instruction, depending on how closely the VMI primitives match the required behavior. For example:

1. It is possible to use VMI floating point primitives without modification. This provides fastest-possible simulation as floating point operations are efficiently mapped to native floating point instructions.
2. It is possible to use VMI floating point primitives directly with some result adjustment in cases where `NaN` or integer/unsigned indeterminate results are generated. This result adjustment is efficiently done using *handler* functions.
3. It is possible to use VMI floating point primitives directly with result adjustment applied to *every* result, whether a `NaN` or not.
4. It is possible to specify *user-defined* operation primitives, which are callback functions executed within the scope of a (possibly SIMD) floating point instruction.
5. If no other approach is possible, the instruction can be implemented using non-floating-point VMI primitives (usually an embedded call). From VMI version 7.20.0, there are a set of VMI *run-time* floating point primitives available that exactly match the behavior of the morph-time primitives. See the *VMI Run Time Function Reference* manual for more information.

The *VMI Morph Time Function* API implements functions allowing many floating point operations to be implemented natively. Every API function is available in both a simple and a SIMD form. In the SIMD form, a number of operations are performed in parallel, with the results committed only if no operation raises an enabled exception. Unless otherwise stated, floating point operations comply with IEEE Standard 754 - 2008.

The *VMI Run Time Function Reference* manual describes how general characteristics of a floating point unit can be configured, and also describes functions to query and update the simulated floating point control word. It also describes run-time floating point primitives that exactly match the behavior of the morph-time primitives.

9.1 General Floating Point Operation Flow

The floating point operation primitives described later in this section all use a similar flow, outlined below in full SIMD form, as pseudo-code:

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whether stages in the flow are present or absent depends on FPU *configuration* settings, described below.

9.2 `vmiFPConfig` Structure

Every floating point operation is executed with a *configuration* that specifies implementation-specific details of its implementation. There is a *default configuration*, defined using the VMI Run Time function `vmirtConfigureFPU`. In addition, any floating point operation can specify an *operation-specific configuration*, which takes priority over the default configuration. Normally, a default configuration is set up when the model is initialized, and operation-specific configurations used only for those instructions which require different behavior.

Configuration information is given in a static constant structure of type `vmiFPConfig`, defined in `vmiTypes.h` as follows:

```
typedef enum vmiFPFlagForceE {
    vmi_FF_None,    // no force (use value in vmiFPControlWord)
    vmi_FF_0,       // force to 0 (ignore value in vmiFPControlWord)
    vmi_FF_1,       // force to 1 (ignore value in vmiFPControlWord)
} vmiFPFlagForce;

typedef struct vmiFPConfigS {
    Uns16               QNaN16;
    Uns32               QNaN32;
    Uns64               QNaN64;
    Uns8                indeterminateUns8;
    Uns16               indeterminateUns16;
    Uns32               indeterminateUns32;
    Uns64               indeterminateUns64;
    vmiFPQNaN16ResultFn QNaN16ResultCB;
    vmiFPQNaN32ResultFn QNaN32ResultCB;
    vmiFPQNaN64ResultFn QNaN64ResultCB;
    vmiFPInd8ResultFn   indeterminate8ResultCB;
    vmiFPInd16ResultFn  indeterminate16ResultCB;
    vmiFPInd32ResultFn  indeterminate32ResultCB;
```

```
    vmiFPInd64ResultFn  indeterminate64ResultCB;
    vmiFPTinyResultFn   tinyResultCB;
    vmiFPTinyArgumentFn tinyArgumentCB;
    vmiFP8ResultFn      fp8ResultCB;
    vmiFP16ResultFn     fp16ResultCB;
    vmiFP32ResultFn     fp32ResultCB;
    vmiFP64ResultFn     fp64ResultCB;
    vmiFPArithExceptFn  fpArithExceptCB;
    vmiFPFlags          suppressFlags;
    Bool                stickyFlags;
    Bool                fzClearsPF;
    Bool                tininessBeforeRounding;
    Bool                perElementFlags;
    vmiFPFlagForce      forceAHP   : 2;
    vmiFPFlagForce      forceFZ16  : 2;
    vmiFPFlagForce      forceDAZ16 : 2;
} vmiFPConfig;
```

The structure fields are as follows:

1. `QNaN16` specifies the bit pattern produced when a floating point operation generates a 16-bit `QNaN` result. Normally this should be `0x7e00`, but older versions of IEEE Standard 754 permit the most significant bit of the significand to be reversed for `QNaN` and `SNaN`. On a processor where `QNaN` and `SNaN` values are indeed reversed, a different value should be specified (for example, `0x7dff`).
2. `QNaN32` specifies the bit pattern produced when a floating point operation generates a 32-bit `QNaN` result. Normally this should be `0x7fc00000`, but older versions of IEEE Standard 754 permit the most significant bit of the significand to be reversed for `QNaN` and `SNaN`. On a processor such as the MIPS where `QNaN` and `SNaN` values are indeed reversed, a different value should be specified (for example, `0x7fbfffff` for MIPS).
3. `QNaN64` specifies the bit pattern produced when a floating point operation generates a 64-bit `QNaN` result. Normally this should be `0x7ff8000000000000ULL`, but a processor such as the MIPS where `QNaN` and `SNaN` values are reversed, a different value should be specified (for example, `0x7ff7ffffffffffffULL` for MIPS).
4. `QNaN16ResultCB` is a callback function which, if given, is called whenever a 16-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value.
5. `QNaN32ResultCB` is a callback function which, if given, is called whenever a 32-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value.
6. `QNaN64ResultCB` is a callback function which, if given, is called whenever a 64-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value.
7. `indeterminateUns8` specifies the bit pattern produced when a floating point operation generates an 8-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x80`.
8. `indeterminateUns16` specifies the bit pattern produced when a floating point operation generates a 16-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x8000`.

9. `indeterminateUns32` specifies the bit pattern produced when a floating point operation generates a 32-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x80000000`.
10. `indeterminateUns64` specifies the bit pattern produced when a floating point operation generates a 64-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x8000000000000000ULL`.
11. `indeterminate8ResultCB` is a callback function which, if given, is called whenever an 8-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value.
12. `indeterminate16ResultCB` is a callback function which, if given, is called whenever a 16-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value.
13. `indeterminate32ResultCB` is a callback function which, if given, is called whenever a 32-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value.
14. `indeterminate64ResultCB` is a callback function which, if given, is called whenever a 64-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value.
15. `tinyResultCB` is a callback function which, if given, is called whenever a tiny (denormalized) result is generated to give the processor model the opportunity to modify the resulting tiny value (or take any other action).
16. `tinyArgumentCB` is a callback function which, if given, is called whenever a denormalized argument is detected to give the processor model the opportunity to modify the argument value (or take any other action).
17. `fp8ResultCB` is a callback function which, if given, is called whenever an 8-bit result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`.
18. `fp16ResultCB` is a callback function which, if given, is called whenever a 16-bit result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`.
19. `fp32ResultCB` is a callback function which, if given, is called whenever a 32-bit result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`.
20. `fp64ResultCB` is a callback function which, if given, is called whenever a 64-bit result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`.
21. `fpArithExceptCB` is an exception handler callback function which is called whenever a floating point operation generates unmasked exceptions. The

exception handler callback will typically update processor state and cause a jump to a vector address.

22. `suppressFlags` is a field if type `vmiFPFlags` which enables flags generated by a floating point operation to be suppressed: any flag set to 1 in the bitmask will be masked out of the operation result flags.
23. `stickyFlags` is a boolean field which specifies whether the operation result flags should replace any current value of the output flags (if `False`) or whether operation flags should be combined with existing flags using bitwise-or (if `True`).
24. `fzClearsPF` is a boolean field that should be `True` if the processor implements *flush-to-zero* mode and when denormal results are flushed to zero *the precision flag in the floating point status word is not set*. If the processor does *not* implement flush-to-zero mode, or if the precision flag should be *set* when results are flushed to zero, then the argument should be `False`. Most floating point implementations set the precision flag when a denormal result is flushed to zero (e.g. x86, MIPS) but some do not (e.g. ARM).
25. `tininessBeforeRounding`[3] is a boolean field that indicates whether tininess should be detected before rounding a result or afterwards. This affects behavior for intermediate results that round to a minimum normal value of greater absolute magnitude. The boolean affects all floating point operations using IEEE types.
26. `perElementFlags` is a boolean field that indicates whether for a SIMD operation the exception flags for each operation should be reported separately or aggregated. If `perElementFlags` is `False`, then exception flags for all parallel operations will be aggregated (using bitwise-or) and the result stored in the `flags` register specified for a floating point operation (see `vmimtFUnopRR` for an example of how the flags register is specified). If `perElementFlags` is `True`, then flags for each operation will instead be stored in an array of flag bytes *immediately following* the flags specified for a floating point operation. For example, flags for operation 0 will be stored at the flags register location+1, flags for operation 1 will be stored at the flags register location+2 and so on. These flag bytes will typically be used in the floating point exception handler (specified using the `fpArithExceptCB` field) to determine the final flags that should be reported using simulated floating point control registers.
27. `forceAHP` is a field of type `vmiFPFlagForce` which causes the apparent value of the `AHP` field in the current `vmiFPControlWord` to be forced to a particular value irrespective of the actual value of the field.
28. `forceFZ16` is a field of type `vmiFPFlagForce` which causes the apparent value of the `FZ16` field in the current `vmiFPControlWord` to be forced to a particular value irrespective of the actual value of the field.
29. `forceDAZ16` is a field of type `vmiFPFlagForce` which causes the apparent value of the `DAZ16` field in the current `vmiFPControlWord` to be forced to a particular value irrespective of the actual value of the field.

[3] Note that prior to VMI version 7.20.0, this field was called `tininessAfterRounding` and had the opposite sense. The field name and sense have been changed to match the host architecture.

9.3 *vmiFPControlWord* Structure

The dynamic behavior of a processor simulated FPU is specified by a `vmiFPControlWord` structure, defined in `vmiTypes.h`:

```
typedef struct vmiFPControlWordS {

                        // INTERRUPT MASKS
    Uns32   IM    : 1;  // invalid operation mask
    Uns32   DM    : 1;  // denormal mask
    Uns32   ZM    : 1;  // divide-by-zero mask
    Uns32   OM    : 1;  // overflow mask
    Uns32   UM    : 1;  // underflow mask
    Uns32   PM    : 1;  // precision mask
    Uns32   UD1M  : 1;  // user-defined flag 1 mask
    Uns32   UD2M  : 1;  // user-defined flag 2 mask

                        // ROUNDING AND PRECISION
    Uns32   RC    : 3;  // rounding control
    Uns32   FZ    : 1;  // flush to zero
    Uns32   DAZ   : 1;  // denormals are zeros flag

                        // HALF-PRECISION
    Uns32   AHP   : 1;  // use ARM AHP format
    Uns32   FZ16  : 1;  // flush to zero
    Uns32   DAZ16 : 1;  // denormals-are-zeros flag

} vmiFPControlWord;
```

The first eight fields are *interrupt masks* that specify whether a floating-point arithmetic exception of the indicated type should by masked. If the exception is masked (the bit is 1), the exception will be ignored. If the exception is unmasked (the bit is 0), any exception of the indicated type will be signaled by calling the processor arithmetic exception handler (defined with the `VMI_ARITH_EXCEPT_FN` macro in `vmiAttrs.h` and passed as the `arithExceptCB` field of the processor `vmiIASAttr` structure). Masks other than `DM`, `UD1M` and `UD2M` are the IEEE Standard 754 exception masks. `DM` is a non-standard mask indicating *denormal operands*. Mask bits `UD1M` and `UD2M` are available for user-defined purposes. Each mask corresponds to a flag described in the next section.

The `RC` field specifies the *rounding control* to use when arithmetic results cannot be exactly represented and precision exceptions are masked. The field value should be one of the first six members of the `vmiFPRC` enumeration:

```
typedef enum vmiFPRCE {

    // these values are valid in both conversion functions and in the rounding
    // control field of vmiFPControlWord, below
    vmi_FPR_NEAREST = 0,    // round towards nearest (even)
    vmi_FPR_NEG_INF = 1,    // round towards negative infinity
    vmi_FPR_POS_INF = 2,    // round towards positive infinity
    vmi_FPR_ZERO    = 3,    // round towards zero
    vmi_FPR_AWAY    = 4,    // round towards nearest, tie away
    vmi_FPR_ODD     = 5,    // round to odd (Von Neumann rounding)

    // these values are valid in conversion functions only
    vmi_FPR_CURRENT = 6,    // use currently-active rounding control
    vmi_FPR_USER    = 0x10  // bitmask implying user-defined (implemented with
                            // result handler functions)
} vmiFPRC;
```

The FZ field specifies that denormal results should be flushed to zero. The DAZ field specifies that denormal arguments should be treated as zero. Neither of these modes are IEEE 754 compliant, but many processors support variants of them.

Fields AHP, FZ16 and DAZ16 control the behavior of operations using 16-bit floating point types. If AHP is 1, then operations using 16-bit floating point numbers will use ARM *AHP* semantics: these specify a modified version of half-precision floating point in which values that would normally encode infinity and NaN values are instead used to extend the range of normalized numbers. Fields FZ16 and DAZ16 are equivalent to FZ and DAZ but apply to 16-bit floating point numbers.

The floating point control word in use for a processor can be set and fetched using two functions from the *VMI Run Time Function* API:

```
//
// Get the processor floating point control word
//
vmiFPControlWord vmirtGetFPControlWord(vmiProcessorP processor);

//
// Set the processor floating point control word
//
void vmirtSetFPControlWord(vmiProcessorP processor, vmiFPControlWord fpcw);
```

9.4 vmiFPFlags Structure

The exception flags generated by a floating point instruction are specified by a vmiFPFlags union, defined in vmiTypes.h:

```
typedef union vmiFPFlagsU {
    Uns8 bits;      // all flags
    struct {
        Uns32 I   : 1;  // invalid operation flag
        Uns32 D   : 1;  // denormal flag
        Uns32 Z   : 1;  // divide-by-zero flag
        Uns32 O   : 1;  // overflow flag
        Uns32 U   : 1;  // underflow flag
        Uns32 P   : 1;  // precision flag
        Uns32 UD1 : 1;  // user-defined flag 1
        Uns32 UD2 : 1;  // user-defined flag 2
    } f;
} vmiFPFlags;
```

The bits field allows all flags to be accessed together as an Uns8 type; alternatively, flags may be accessed individually using the structure members. A brief description of each flag follows: refer to the IEEE 754 Standard for more information on all flags except the non-standard D, UD1 and UD2 flags.

I: *invalid operation flag*
This flag is set whenever an operation is considered invalid by the FPU. Examples are 0 divided by 0, subtracting infinity from infinity, NaN inputs to some instructions, or attempting to find the square root of a negative number. If the exception is masked by the IM bit in the control word, the result of the floating point operation is a NaN. The floating

point configuration may specify a *NaN handler* to configure the exact `NaN` that is returned in these circumstances.

`Z`: *divide-by-zero flag*
This flag is set whenever division of a finite non-zero is attempted. If the exception is masked by the `ZM` bit in the control word, a properly-signed infinity is generated.

`O`: *overflow flag*
This flag is set whenever a value is too large to be represented. For example, multiplication of two very large numbers can generate an overflow. If the exception is masked by the `OM` bit in the control word, a properly-signed infinity is generated.

`U`: *underflow flag*
The behavior of this flag depends on the corresponding mask bit (`UM`) in the control word. If the underflow exception is *masked*, the flag is set only if a result is *both tiny and inexact*; if the underflow exception is unmasked, the flag is set for *any tiny result*. As an example, dividing a very small number by a large number can generate an underflow. If the exception is masked by the `UM` bit in the control word, a denormal or zero result is produced, as appropriate.

`P`: *precision flag*
This flag is set whenever some precision is lost by a floating point operation. For example, dividing 1.0 by 10.0 does not generate an exact result and causes the precision flag to be set. If the exception is masked by the `PM` bit in the control word, the result is rounded according to the rounding control specified by the `RC` field of the control word or the rounding control specified for the instruction (see above).

`D`: *denormal operand flag*
This flag is set whenever the input to a floating point operation is denormalized (also known as *subnormal*). If exceptions are enabled, this causes an exception *before* the floating point operation starts. If the exception is masked by the `DM` bit in the control word, the current floating point operation continues normally. If the `DAZ` bit is set in the control word, then denormal operands are rounded to zero. The floating point configuration may specify a *tiny operand handler* to configure the exact model behavior under which this happens.

`UD1` and **`UD2`**: *user-defined flags*
These flags are not used by the simulator but are available for signaling user-defined exceptions – see following sections for more information.

Note that arithmetic instructions can signal more than one exception: for example, it is possible to get both an underflow and precision exception signalled by a single floating point instruction.

All floating point arithmetic instructions have a `vmiReg` target register that is assigned the instruction exception flags, provided that the floating point exception handler is not called. If the floating point exception handler *is* called (at least one of the exceptions raised by the instruction is not masked) the target flags register is not updated and the generated flags are instead passed as the `flags` argument of the `vmiFPArithExceptFn` handler function (see below).

9.5 vmiFType Enumeration

The type of the arguments for floating point instructions is specified by the `vmiFType` enumeration, defined in `vmiTypes.h`:

```
typedef enum {

    // these values specify that evaluation should be performed using IEEE 754
    // semantics (intermediates are the same type)
    vmi_FT_16_IEEE_754 = 16 | VMI_FT_IEEE_754,  // 16-bit floating point
    vmi_FT_32_IEEE_754 = 32 | VMI_FT_IEEE_754,  // 32-bit floating point
    vmi_FT_64_IEEE_754 = 64 | VMI_FT_IEEE_754,  // 64-bit floating point

    // these values specify that evaluation should be performed using Intel x87
    // semantics (intermediates are promoted to 80-bit long double format)
    vmi_FT_32_X87      = 32 | VMI_FT_X87,       // 32-bit floating point
    vmi_FT_64_X87      = 64 | VMI_FT_X87,       // 64-bit floating point
    vmi_FT_80_X87      = 80 | VMI_FT_X87,       // 80-bit floating point

    // these values are valid in conversion operations only
    vmi_FT_8_INT       =  8 | VMI_FT_INT,       // 8-bit signed integer
    vmi_FT_16_INT      = 16 | VMI_FT_INT,       // 16-bit signed integer
    vmi_FT_32_INT      = 32 | VMI_FT_INT,       // 32-bit signed integer
    vmi_FT_64_INT      = 64 | VMI_FT_INT,       // 64-bit signed integer
    vmi_FT_8_UNS       =  8 | VMI_FT_UNS,       // 8-bit unsigned integer
    vmi_FT_16_UNS      = 16 | VMI_FT_UNS,       // 16-bit unsigned integer
    vmi_FT_32_UNS      = 32 | VMI_FT_UNS,       // 32-bit unsigned integer
    vmi_FT_64_UNS      = 64 | VMI_FT_UNS,       // 64-bit unsigned integer

    // this value specifies BFLOAT16 type, which is vmi_FT_32_IEEE_754 with
    // fraction truncated to 7 bits, giving a storage size of 16 bits
    vmi_FT_BFLOAT16    = 16 | VMI_FT_IEEE_754 | VMI_FT_OP1

} vmiFType;
```

Members `vmi_FT_16_IEEE_754`[4], `vmi_FT_32_IEEE_754` and `vmi_FT_64_IEEE_754` specify IEEE-compliant 16, 32 and 64 bit floating point values and semantics, respectively (see below for semantic differences between IEEE and x87 modes).

Members `vmi_FT_32_X87`, `vmi_FT_64_X87`, and `vmi_FT_80_X87` specify x87 32, 64 and 80 bit values and semantics, respectively (see below for semantic differences between IEEE and x87 modes).

Members `vmi_FT_8_INT`, `vmi_FT_16_INT`, `vmi_FT_32_INT` and `vmi_FT_64_INT` specify 8, 16, 32 and 64 bit signed integer values and are valid as the source or target of floating point conversion functions only.

[4] Type `vmi_FT_16_IEEE_754` can be used for all floating point operations from VMI version 6.44.0. Prior to this, it could only be used for *conversion* operations.

Members `vmi_FT_8_UNS`, `vmi_FT_16_UNS`, `vmi_FT_32_UNS` and `vmi_FT_64_UNS` specify 8, 16, 32 and 64 bit unsigned integer values and are valid as the source or target of floating point conversion functions only.

Member `vmi_FT_BFLOAT16` specifies 16 bit brain floating point values and semantics. This format is a truncated version of IEEE-compliant 32-bit floating point format, typically of use in machine learning applications.

9.6 IEEE and x87 Semantic Differences

IEEE and x87 semantics differ in these ways.

Operand and Intermediate Size

When using IEEE semantics, calculations are performed using the operand size (32-bit float or 64-bit double). When using x87 semantics, operands are first converted to 80-bit long doubles and the result is rounded to float or double length on operation completion, if required. Note that x87 semantics can therefore cause *two* rounding events, firstly when an intermediate result is rounded to 80-bit precision, and a secondly when the final result is rounded to 32-bit or 64-bit precision from 80-bit precision.

For IEEE ternary floating point operations which are specified *not* to round intermediates (argument `roundInt` is `False`) the intermediate result of the multiply is represented using infinite precision. This means that such operations correspond to the IEEE definition of *fused-multiply-add* operations. When using x87 semantics with `roundInt` specified as `False`, the intermediate result is rounded to 80-bit precision. When `roundInt` is `True`, intermediate results of a ternary operation are rounded to the operand size in both cases.

`NaN` Operands

When operations have more than one `NaN` operand and a `NaN` result is generated, the results differ when using IEEE and x87 semantics, as follows:

Source Operands	**Result**
`SNaN` and `QNaN`, `QNaN` and `SNaN`	x87: `QNaN` source operand IEEE: First `NaN` operand, converted to `QNaN`
`SNaN` and `SNaN`	x87: `SNaN` operand with largest significand, converted to `QNaN` IEEE: First `SNaN` operand, converted to `QNaN`
`QNaN` and `QNaN`	x87: `QNaN` operand with largest significand IEEE: First `QNaN` operand

9.7 QNaN/SNaN Polarity Switch

This section describes how to control QNaN/SNaN polarity using a floating point configuration. This affects the highlighted stages in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Older versions of IEEE Standard 754 permit the most significant bit of the significand to be of either polarity to represent QNaN and SNaN. For most processors, a QNaN is indicated by the most significant bit being 1, and SNaN is indicated by the most significant bit being 0 (this allows any SNaN to be efficiently converted to a QNaN by setting the significand msb). Some processor architectures (e.g. legacy MIPS) define the values to be the other way round.

The QNaN/SNaN polarity is controlled by three fields in the configuration structure. QNaN32 specifies the bit pattern produced when a floating point operation generates a 32-bit QNaN result, and also implicitly the polarity of the signaling bit. Fields QNaN16 and QNaN64 define analogous values for a 16-bit QNaN and a 64-bit QNaN, respectively

If the default QNaN values imply a reversed signaling bit polarity, NaN values are automatically switched from reversed format as arguments are processed, and QNaN results are switched to reversed format on operation completion. Stages between the two highlighted lines in the above description always operate on NaN values with standard polarity.

9.8 Denormalized Argument Handler

This section describes how to control denormals-are-zeros mode (`DAZ` mode) using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever a denormalized argument is detected for a floating point operation and the denormals-are-zero (`DAZ` or `DAZ16`) bit is set in the current floating point control word[5], the tiny argument handler is called (specified using field `tinyArgumentCB` in the active configuration). The handler function is defined using the `VMI_FP_TINY_ARGUMENT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_TINY_ARGUMENT_FN(_NAME) vmiFP80Arg _NAME( \
    vmiProcessorP processor,        \
    vmiFP80Arg    value,            \
    vmiFPFlagsP   setFlags          \
)
```

The handler is passed the following arguments:

1. The current processor;
2. The tiny argument value, represented as a `vmiFP80Arg`;
3. An argument of `setFlags` of type `vmiFPFlagsP`, in which bits can be set to 1 to indicate a floating point exception caused by handling the tiny result.

It must return an appropriately-signed zero value as a `vmiFP80Arg` and may perform other updates to processor state.

The `vmiFP80Arg` type is defined as follows, and holds a floating point value in the Intel x87 80-bit format:

```
#define VMI_FP_80_BYTES 10
typedef union vmiFP80ArgU {
    Flt80      f80;
    Flt80Parts f80Parts;
    Uns8       bytes[VMI_FP_80_BYTES];
```

[5] The `DAZ16` bit is used for *16-bit* floating point operands, and the `DAZ` bit for 32, 64 and 80 bit operands.

```
} vmiFP80Arg;
```

Type `Flt80Parts` is one of a set of types used to decode components of floating point numbers:

```
typedef struct Flt16PartsS {
    Uns32 fraction : 10;
    Uns32 exponent :  5;
    Bool  sign     :  1;
} Flt16Parts;

typedef struct Flt32PartsS {
    Uns32 fraction : 23;
    Uns32 exponent :  8;
    Bool  sign     :  1;
} Flt32Parts;

typedef struct Flt64PartsS {
    Uns64 fraction : 52;
    Uns32 exponent : 11;
    Bool  sign     :  1;
} Flt64Parts;

typedef struct Flt80PartsS {
    Uns64 fraction : 64;
    Uns32 exponent : 15;
    Bool  sign     :  1;
} Flt80Parts;
```

Example

This example is derived from the standard MIPS model.

```
static VMI_FP_TINY_ARGUMENT_FN(handleTinyArgument) {

    mipsP tc = (mipsP)processor;

    // when denormal arguments are flushed to zero, set Precision flag unless
    // attribute FA_DONT_SET_I_FLAG is also specified (floating point compare)
    if(!(tc->fopAttrs & FA_DONT_SET_I_FLAG)) {
        setFlags->f.P = 1;
    }

    // return appropriately-signed zero
    value.f80Parts.fraction = 0;
    value.f80Parts.exponent = 0;

    return value;
}
```

This returns an appropriately-signed zero value and updates processor state to force the *precision* (`P`) flag to be set on instruction completion.

If the `DAZ` or `DAZ16` bit is set in the current floating point control word and no tiny argument handler is specified in the configuration, denormal inputs are flushed to an appropriately-signed zero and the precision flag is set in all cases. Default behavior cannot be used for the MIPS model because some operations do not set the precision flag, even though they flush their arguments to zero.

9.9 Tiny Result Handler

This section describes how to control flush-to-zero mode (`FZ` mode) using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever an underflow exception is generated by a floating point operation and the flush-to-zero (`FZ` or `FZ16`) bit is set in the current floating point control word[6], the tiny result handler is called (specified using field `tinyResultCB` in the active configuration). The handler function is defined using the `VMI_FP_TINY_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_TINY_RESULT_FN(_NAME) vmiFP80Arg _NAME( \
    vmiProcessorP processor,        \
    vmiFP80Arg    value,            \
    Uns32         bits,             \
    Bool          isIntermediate,   \
    vmiFPFlagsP   setFlags          \
)
```

The handler function is passed the following arguments:

1. The processor;
2. The result value, represented as a `vmiFP80Arg`;
3. An `Uns32` indicating the result size, in bits;
4. A Boolean indicating whether the result is the final result from a VMI floating point operation (if `False`, it is an intermediate result of a ternary operation);
5. An argument of `setFlags` of type `vmiFPFlagsP`, in which bits can be set to 1 to indicate a floating point exception caused by handling the tiny result.

The handler function should return the desired result, encoded as a `vmiFP80Arg`.

Example

This example is derived from the standard MIPS model.

```
static VMI_FP_TINY_RESULT_FN(handleTinyResult) {
```

[6] The `FZ16` bit is used for *16-bit* floating point results, and the `FZ` bit for 32, 64 and 80 bit results.

```
    mipsP     tc         = (mipsP)processor;
    Bool      isNegative = value.bytes[VMI_FP_80_BYTES-1] & 0x80;
    tinyValue tv;

    GET_VPE;

    // get FPU control bits
    Bool FS = COP1_FIELD(vpe, FENR, FS);
    Bool FN = COP1_FIELD(vpe, FCSR, FN);
    Bool FO = COP1_FIELD(vpe, FCSR, FO);

    // expect either FS or FN to be set if we get here
    VMI_ASSERT(
        FS || FN,
        "expected FS or FN to be set"
    );

    // should not be called for intermediates if FO is set
    VMI_ASSERT(
        !(isIntermediate && FO),
        "unexpected intermediate with FO bit set"
    );

    // when results are flushed to zero, set Underflow and Precision flags
    setFlags->f.U = 1;
    setFlags->f.P = 1;

    // indicate that tiny results are allowed for this function
    tc->fopAttrs |= FA_ALLOW_O_DENORMALS;

    // get current rounding mode
    vmiFPRC rc = getCurrentRoundingMode(tc);

    if(FN && !isIntermediate && (rc==vmi_FPR_NEAREST)) {

        // get minnorm/2 for the result
        vmiFP80Arg minNormDiv2 = (
            isFlt32 ?
            tinyValues32[TV_PLUS_MINNORM_DIV_2] :
            tinyValues64[TV_PLUS_MINNORM_DIV_2]
        );

        // is value >= minnorm/2?
        Bool geThanMinNormDiv2 = (
            ((value.bytes[9]&0x7f) >= minNormDiv2.bytes[9]) &&
            ( value.bytes[8]       >= minNormDiv2.bytes[8]) &&
            ( value.bytes[7]       >= minNormDiv2.bytes[7])
        );

        // here if FN should be applied
        if(isNegative) {
            tv = geThanMinNormDiv2 ? TV_MINUS_MINNORM : TV_MINUS_0;
        } else {
            tv = geThanMinNormDiv2 ? TV_PLUS_MINNORM : TV_PLUS_0;
        }

    } else {

        // here if FS should be applied
        if(isNegative) {
            tv = (rc==vmi_FPR_NEG_INF) ? TV_MINUS_MINNORM : TV_MINUS_0;
        } else {
            tv = (rc==vmi_FPR_POS_INF) ? TV_PLUS_MINNORM : TV_PLUS_0;
        }
    }

    return isFlt32 ? tinyValues32[tv] : tinyValues64[tv];
}
```

The MIPS processor does not generate denormalized results in the normal case – usually, operations producing such results generate Unimplemented Operation exceptions instead. However, it has three special mode bits (FS, FO and FN) that cause denormalized results to be flushed either to zero or to the smallest normalized value, depending on the rounding mode (among other things).

The example tiny result handler examines the offending tiny result value and returns either zero or the smallest normalized value (appropriately signed) depending on the processor state. It also updates processor state to force the *precision* and *underflow* flags to be set on instruction completion.

9.10 General Result Handlers

This section describes how to control handling of general operation results using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever an operation result is produced, a result handler can be supplied which can modify the resulting value if required (specified using `fp8ResultCB`, `fp16ResultCB`, `fp32ResultCB` and `fp64ResultCB` fields in the active configuration). Unlike `QNaN` result handlers (described next), general result handlers are called for *every* generated result, not just `QNaN` results. The 32-bit general handler function is defined using the `VMI_FP_32_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_32_RESULT_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    Uns32         result32,     \
    Uns32         argNum,       \
    vmiFPArgP     args,         \
    vmiFPFlagsP   setFlags      \
)
typedef VMI_FP_32_RESULT_FN((*vmiFP32ResultFn));
```

The handler function is passed the following:

1. The processor generating the result;
2. The result value, represented as an `Uns32`;
3. A count of the number arguments to the operation;
4. An array of `argNum` arguments to the operation;
5. An argument of `setFlags` of type `vmiFPFlagsP`, in which bits can be set to 1 to indicate a floating point exception caused by handling the result.

The handler function should return the desired 32-bit result, encoded as an `Uns32`. It is passed an ordered list of all arguments to the operation in the `args` array, which holds `argNum` values. Each value is a `vmiFPArg` structure, defined as follows:

```
#define VMI_FP_80_BYTES 10
typedef struct vmiFPArgS {
    vmiFType type;
```

```
    union {
        // use these for 8-bit types
        Uns8       u8;
        Int8       i8;
        // use these for 16-bit types
        Uns16      u16;
        Int16      i16;
        Flt16Parts f16Parts;
        // use these for 32-bit types
        Uns32      u32;
        Int32      i32;
        Flt32      f32;
        Flt32Parts f32Parts;
        // use these for 64-bit types
        Uns64      u64;
        Int64      i64;
        Flt64      f64;
        Flt64Parts f64Parts;
        // use these for 80-bit types
        Flt80      f80;
        Flt80Parts f80Parts;
        Uns8       bytes[VMI_FP_80_BYTES];
    };
} vmiFPArg;
```

Field `type` specifies the argument type – note that floating point conversion operations may take argument values of different size and class to the required result, so the handler must cope with such cases.

There will be 1, 2 or 3 values in the `args` list, depending on whether the floating point operation is a conversion, unary, binary or ternary.

There is a similar handler for 64-bit general results, defined using the `VMI_FP_64_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_64_RESULT_FN(_NAME) Uns64 _NAME( \
    vmiProcessorP processor,    \
    Uns64         result64,     \
    Uns32         argNum,       \
    vmiFPArgP     args,         \
    vmiFPFlagsP   setFlags      \
)
typedef VMI_FP_64_RESULT_FN((*vmiFP64ResultFn));
```

The handler works in identical fashion to the 32-bit general result handler, the only difference being that it takes and returns values represented as an `Uns64`.

A similar handler for 16-bit general results is defined using the `VMI_FP_16_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_16_RESULT_FN(_NAME) Uns16 _NAME( \
    vmiProcessorP processor,    \
    Uns16         result16,     \
    Uns32         argNum,       \
    vmiFPArgP     args,         \
    vmiFPFlagsP   setFlags      \
)
typedef VMI_FP_16_RESULT_FN((*vmiFP16ResultFn));
```

The handler works in identical fashion to the 32-bit general result handler, the only difference being that it takes and returns values represented as an `Uns16`.

From VMI version 6.28.0, there is also an 8-bit general result handler, defined using the `VMI_FP_8_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_8_RESULT_FN(_NAME) Uns8 _NAME( \
    vmiProcessorP processor,    \
    Uns8          result8,      \
    Uns32         argNum,       \
    vmiFPArgP     args,         \
    vmiFPFlagsP   setFlags      \
)
typedef VMI_FP_8_RESULT_FN((*vmiFP8ResultFn));
```

Again, the only difference between this and the 32-bit general result handler is the type of the second argument and result. It is used for conversions to `vmi_FT_8_INT` and `vmi_FT_8_INT` types only.

For function results in brain float format (specified as type `vmi_FT_BFLOAT16`) *the 32-bit result handler is called*. The argument `result32` is composed of the 16-bit brain float result, shifted left by 16 bits (i.e. with least significant fraction bits filled with zeros). The function should return an adjusted 32-bit floating point result also with the least-significant 16 bits filled with zeros.

Example

This example is derived from the OVP MIPS model. In the MIPS processor, floating point `RECIP` and `RSQRT` instructions always set the precision (inexact) flag, unless the result is zero, a `QNaN` or infinity.

```
static VMI_FP_32_RESULT_FN(recipRsqrtResult32) {

    if(!(result32 & ~MIPS_SIGN_32)) {
        // no action if (signed) zero result
    } else if((result32 & MIPS_EXP_32) != MIPS_EXP_32) {
        // not a QNaN or infinite result - force the inexact flag
        setFlags->f.P = 1;
    }

    // return unmodified result
    return result32;
}
```

9.11 QNaN Handlers

This section describes how to control handling of QNaN operation results using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever a QNaN value is produced as a result, a QNaN handler can be supplied which can modify the resulting value if required (specified using QNaN16ResultCB, QNaN32ResultCB and QNaN64ResultCB fields in the active configuration). The 32-bit QNaN handler function is defined using the VMI_FP_QNAN32_RESULT_FN macro from vmiTypes.h:

```
#define VMI_FP_QNAN32_RESULT_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    Uns32         QNaN32,       \
    Uns32         NaNArgNum,    \
    vmiFPArgP     NaNArgs,      \
    Uns32         allArgNum,    \
    vmiFPArgP     allArgs       \
)
```

The handler function is passed the following:

1. The processor generating the QNaN result;
2. The QNaN value, represented as an Uns32;
3. A count of the number of NaN arguments to the operation;
4. An array of NaNArgNum NaN arguments to the operation;
5. A count of *all* arguments to the operation;
6. An array of allArgNum arguments to the operation.

The handler function should return the desired 32-bit result, encoded as an Uns32.

If the operation had any NaN inputs (NaNArgNum is non-zero), then the handler can obtain an ordered list of those values from the NaNArgs array, which holds NaNArgNum values. Each value is a vmiFPArg structure, defined as follows:

```
#define VMI_FP_80_BYTES 10
```

```
typedef struct vmiFPArgS {
    vmiFType type;
    union {
        // use these for 8-bit types
        Uns8       u8;
        Int8       i8;
        // use these for 16-bit types
        Uns16      u16;
        Int16      i16;
        Flt16Parts f16Parts;
        // use these for 32-bit types
        Uns32      u32;
        Int32      i32;
        Flt32      f32;
        Flt32Parts f32Parts;
        // use these for 64-bit types
        Uns64      u64;
        Int64      i64;
        Flt64      f64;
        Flt64Parts f64Parts;
        // use these for 80-bit types
        Flt80      f80;
        Flt80Parts f80Parts;
        Uns8       bytes[VMI_FP_80_BYTES];
    };
} vmiFPArg;
```

Field `type` specifies the argument type – note that floating point conversion operations may take argument values of different size and class to the required result, so the handler must cope with such cases.

The handler is also passed an ordered list of all arguments to the operation in the `allArgs` array, which holds `allArgNum` values. Each value is a `vmiFPArg` structure, as described above. There will be 1, 2 or 3 values in this list, depending on whether the floating point operation is a unary, binary or ternary.

The `QNaN` handler usually returns a result which is a `NaN`. However, it is also legal to return a *non-`NaN` result*. In this case, *any Invalid Operation exception signalled for the current floating point operation is cleared*. This is typically useful when modeling instructions with special behavior when multiplying infinity and zero: the default behavior is to produce a `QNaN` in these cases, but some processors instead produce a non-`QNaN` result.

There is a similar handler for 64-bit `QNaN` results, defined using the `VMI_FP_QNAN64_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_QNAN64_RESULT_FN(_NAME) Uns64 _NAME( \
    vmiProcessorP processor,    \
    Uns64         QNaN64,       \
    Uns32         NaNArgNum,    \
    vmiFPArgP     NaNArgs,      \
    Uns32         allArgNum,    \
    vmiFPArgP     allArgs       \
)
```

The handler works in identical fashion to the 32-bit `QNaN` handler, the only difference being that it takes and returns `QNaN` values represented as an `Uns64`.

There is also a 16-bit `QNaN` handler, defined using the `VMI_FP_QNAN16_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_QNAN16_RESULT_FN(_NAME) Uns16 _NAME( \
    vmiProcessorP processor,    \
    Uns16         QNaN16,       \
    Uns32         NaNArgNum,    \
    vmiFPArgP     NaNArgs,      \
    Uns32         allArgNum,    \
    vmiFPArgP     allArgs       \
)
```

The handler works in identical fashion to the 32-bit `QNaN` handler, the only difference being that it takes and returns `QNaN` values represented as an `Uns16`.

For function results in brain float format (specified as type `vmi_FT_BFLOAT16`) *the 32-bit QNaN handler is called*. The argument `QNaN32` is composed of the 16-bit brain float result, shifted left by 16 bits (i.e. with least significant fraction bits filled with zeros). The function should return an adjusted 32-bit floating point result also with the least-significant 16 bits filled with zeros.

Example

This example is derived from the OVP MIPS model.

```
inline static Bool is32BitSNaN(vmiFPArgP arg, Bool standardNaN) {
    return (
        (arg->type==vmi_FT_32_IEEE_754) &&
        !(arg->u32 & MIPS_SBIT_32) == standardNaN
    );
}

inline static Bool is64BitSNaN(vmiFPArgP arg, Bool standardNaN) {
    return (
        (arg->type==vmi_FT_64_IEEE_754) &&
        !(arg->u64 & MIPS_SBIT_64) == standardNaN
    );
}

inline static Bool isSNaN(vmiFPArgP arg, Bool standardNaN) {
    return is32BitSNaN(arg, standardNaN) || is64BitSNaN(arg, standardNaN);
}

static VMI_FP_QNAN32_RESULT_FN(handleQNaN32) {

    mipsP tc          = (mipsP)processor;
    Bool  standardNaN = cfgStandardNaN(tc);
    Uns32 i;

    // PASS 1: if any argument is a 32-bit SNaN, return that (as a QNaN) in
    // standard NaN mode, or if it is any SNaN, return the canonical QNaN (in
    // legacy mode)
    for(i=0; i<NaNArgNum; i++) {
        if(standardNaN && is32BitSNaN(&NaNArgs[i], standardNaN)) {
            return NaNArgs[i].u32 | MIPS_SBIT_32;
        } else if(!standardNaN && isSNaN(&NaNArgs[i], standardNaN)) {
            return MIPS_QNAN_32;
        }
    }

    // PASS 2: if any argument is a 32-bit QNaN, return that
    for(i=0; i<NaNArgNum; i++) {
        if(is32BitQNaN(&NaNArgs[i], standardNaN)) {
```

```
            return NaNArgs[i].u32;
        }
    }

    // otherwise, return positive canonical QNaN or the calculated result
    return standardNaN ? IEEE_QNAN_32 : QNaN32;
}
```

The MIPS floating point unit differs from IEEE 754 semantics because, if a `QNaN` result is generated, the pattern for this is based upon `QNaN` operands *only* and not `SNaN` operands. The `QNaN` handler above selects and returns the first `QNaN` from the argument list, or, if no `QNaN` is found, it returns the default `QNaN` pattern.

9.12 8-bit, 16-Bit, 32-Bit and 64-Bit Indeterminate Handlers

This section describes how to control handling of integer/unsigned indeterminate operation results using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Whenever an indeterminate value is produced as a result of a conversion, an indeterminate handler can be supplied which can provide the required result (specified using `indeterminate8ResultCB`, `indeterminate16ResultCB`, `indeterminate32ResultCB` or `indeterminate64ResultCB` fields in the active configuration). The 32-bit handler function is defined using the `VMI_FP_IND32_RESULT_FN` macro from `vmiTypes.h`:

```
#define VMI_FP_IND32_RESULT_FN(_NAME) Uns32 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value,        \
    Bool          isSigned      \
)
```

The handler function is passed the following:

1. The processor generating the indeterminate result;
2. The argument prior to conversion, represented as a `vmiFPArg`;
3. A Boolean indicating whether a signed conversion was performed.

The handler function should return the desired result as an `Uns32`.

There are similar handlers for 8-bit, 16-bit and 64-bit indeterminate value handling, defined using the `VMI_FP_IND8_RESULT_FN`, `VMI_FP_IND16_RESULT_FN` and `VMI_FP_IND64_RESULT_FN` macros from `vmiTypes.h`:

```
#define VMI_FP_IND8_RESULT_FN(_NAME) Uns8 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value,        \
    Bool          isSigned      \
)
#define VMI_FP_IND16_RESULT_FN(_NAME) Uns16 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value,        \
    Bool          isSigned      \
```

```
)
#define VMI_FP_IND64_RESULT_FN(_NAME) Uns64 _NAME( \
    vmiProcessorP processor,    \
    vmiFPArg      value,        \
    Bool          isSigned      \
)
```

The handlers work in identical fashion to the 32-bit indeterminate handler, the only difference being that they return an `Uns8`, `Uns16` and `Uns64` result, respectively.

Example

This example is extracted from the standard ARM model.

```
static Bool isNegative(vmiFPArg value) {

    if(value.type==vmi_FT_32_IEEE_754) {
        return value.f32Parts.sign;
    } else if(value.type==vmi_FT_64_IEEE_754) {
        return value.f64Parts.sign;
    } else if(value.type==vmi_FT_80_X87) {
        return value.f80Parts.sign;
    } else {
        return False;
    }
}

static VMI_FP_IND32_RESULT_FN(handleIndeterminate32) {

    Uns32 result;

    if(isNaN(value)) {
        result = 0;
    } else if(isNegative(value)) {
        result = isSigned ? ARM_MIN_INT32 : ARM_MIN_UNS32;
    } else {
        result = isSigned ? ARM_MAX_INT32 : ARM_MAX_UNS32;
    }

    return result;
}
```

In the ARM processor, out-of-range values are clamped to the minimum and maximum bounds, and `NaN` inputs are clamped to zero. The ARM processor supports both signed and unsigned conversion.

9.13 Floating Point Exceptions

This section describes how to control floating point exceptions using a floating point configuration. This affects the highlighted stage in the pseudo-code description below.

```
for each SIMD operation do
    for each operand do
        switch QNaN/SNaN polarity
        handle denormal inputs (DAZ)
    done
    do operation
    handle flush-to-zero (FZ)
    adjust intermediate result
    switch QNaN/SNaN polarity
    adjust QNaN/indeterminate result
    save intermediate result
done

take enabled exceptions

for each SIMD operation do
    commit intermediate result to result
done
```

Any unmasked floating point exceptions cause the configured *floating point arithmetic exception handler* to be called. The floating point arithmetic exception handler is of type `vmiFPArithExceptFn` and is specified as the `fpArithExceptCB` field of the active configuration:

```
#define VMI_FP_ARITH_EXCEPT_FN(_NAME) vmiFloatExceptionResult _NAME( \
    vmiProcessorP processor,    \
    vmiFPFlagsP   flags         \
)
typedef VMI_FP_ARITH_EXCEPT_FN((*vmiFPArithExceptFn));
```

The `flags` argument to the arithmetic exception handler indicates the exception flags set by the faulting instruction. The handler may modify the processor state to reflect the exception conditions (for example, by changing simulated register state, or using `vmirtSetPCException` to jump to a simulated exception vector). It may also modify fields in the by-ref `flags` structure to simulate flag behavior that diverges from the IEEE standard. It should return `VMI_FLOAT_CONTINUE` to indicate that simulation should continue or `VMI_FLOAT_UNHANDLED` to indicate that an irrecoverable model error has occurred and simulation should terminate.

Example

This example is derived from the OVP MIPS model. Various implementation-specific adjustments are made to the flag settings, and then function `mipsTakeException` is called to enable execution at the exception vector address if required.

```
static VMI_FP_ARITH_EXCEPT_FN(handleFPException) {

    mipsP        tc       = (mipsP)processor;
    vmiFPFlags   enables  = {getEnabledExceptions(tc)};
    mipsFPOpAttr fopAttrs = tc->fopAttrs;

    // handle denormal arguments
    if(!flags->f.D) {
```

```
        // not a denormal argument
    } else if(fopAttrs & FA_ALLOW_I_DENORMALS) {
        // denormal arguments valid for this instruction
        flags->f.D = 0;
    } else {
        // take Unimplemented Operation exception
        FORCE_UNIMPLEMENTED_OPERATION(flags);
    }

    // handle tiny results
    if(!flags->f.U) {
        // not a tiny result
    } else if(fopAttrs & FA_ALLOW_O_DENORMALS) {
        // tiny results valid for this instruction
    } else {
        // take Unimplemented Operation exception
        FORCE_UNIMPLEMENTED_OPERATION(flags);
    }

    // clear underflow if this instruction requires it
    if(fopAttrs & FA_CLEAR_U_FLAG) {
        flags->f.U = 0;
    }

    // take any pending exception
    if((flags->bits & enables.bits)) {
        GET_FPU;
        fpu->cop1Cause = flags->bits;
        mipsTakeException(tc, excCode_FPE, 0, False);
    }

    return VMI_FLOAT_CONTINUE;
}
```

9.14 vmimtFSetRounding

Prototype

```
void vmimtFSetRounding(vmiFPRC rc);
```

Description

This function should be called immediately before a call to `vmimtFUnopRR`, `vmimtFBinopRRR`, `vmimtFTernopRRRR`, `vmimtFUnopSimdRR`, `vmimtFBinopSimdRRR` or `vmimtFTernopSimdRRRR` to modify the rounding mode that should apply to that operation. The specified mode, `rc`, should be one of the following:

```
    vmi_FPR_NEAREST = 0,    // round towards nearest (even)
    vmi_FPR_NEG_INF = 1,    // round towards negative infinity
    vmi_FPR_POS_INF = 2,    // round towards positive infinity
    vmi_FPR_ZERO    = 3,    // round towards zero
    vmi_FPR_AWAY    = 4,    // round towards nearest, tie away
    vmi_FPR_ODD     = 5,    // round to odd (Von Neumann rounding)
```

The effect is to modify the rounding mode, for that operation only, to the given mode instead of the processor's current rounding mode.

Example

The OVP RISC-V model uses this function to implement instruction-specified rounding modes for unary operations as follows:

```
static Bool emitSetOperationRM(riscvMorphStateP state) {

    riscvP      riscv   = state->riscv;
    riscvRMDesc rm      = state->info.rm;
    Bool        validRM = emitCheckLegalRM(riscv, rm);

    if(validRM) {
        vmimtFSetRounding(mapRMDescToRC(rm));
    }

    return validRM;
}

static RISCV_MORPH_FN(emitFUnop) {

    riscvP        riscv = state->riscv;
    riscvRegDesc  fdA   = getRVReg(state, 0);
    riscvRegDesc  fs1A  = getRVReg(state, 1);
    vmiReg        fd    = getVMIReg(riscv, fdA);
    vmiReg        fs1   = getVMIRegFS(riscv, fs1A, getTmp(1));
    vmiFType      type  = getRegFType(fdA);
    vmiFUnop      op    = state->attrs->fpUnop;
    vmiFPConfigCP ctrl  = getFPControl(state);

    if(emitSetOperationRM(state)) {
        vmimtFUnopRR(type, op, fd, fs1, RISCV_FP_FLAGS, ctrl);
        writeReg(riscv, fdA);
    }
}
```

Notes and Restrictions

None.

9.15 vmimtFConvertRR, vmimtFConvertSimdRR

Prototypes

```
void vmimtFConvertRR(
    vmiFType      destType,
    vmiReg        fd,
    vmiFType      srcType,
    vmiReg        fa,
    vmiFPRC       rc,
    vmiReg        flags,
    vmiFPConfigCP config
);

void vmimtFConvertSimdRR(
    Uns32         num,
    vmiFType      destType,
    vmiReg        fd,
    vmiFType      srcType,
    vmiReg        fa,
    vmiFPRC       rc,
    vmiReg        flags,
    vmiFPConfigCP config
);
```

Description

These functions emit code to convert a value in register `ra` that is in format `srcType`, placing the result in register `fd` in format `destType`. `srcType` and `destType` can be any members of the `vmiFType` enumeration, so this function allows conversion between any pair of floating point or integral types. For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd` and `fa` indicate the first register in a contiguous vector.

If the result cannot be exactly represented using the target type, rounding to that type is controlled by the `rc` argument of type `vmiFPRC`, defined as follows:

```
typedef enum vmiFPRCE {

    // these values are valid in both conversion functions and in the rounding
    // control field of vmiFPControlWord, below
    vmi_FPR_NEAREST = 0,    // round towards nearest (even)
    vmi_FPR_NEG_INF = 1,    // round towards negative infinity
    vmi_FPR_POS_INF = 2,    // round towards positive infinity
    vmi_FPR_ZERO    = 3,    // round towards zero
    vmi_FPR_AWAY    = 4,    // round towards nearest, tie away
    vmi_FPR_ODD     = 5,    // round to odd (Von Neumann rounding)

    // these values are valid in conversion functions only
    vmi_FPR_CURRENT = 6,    // use currently-active rounding control
    vmi_FPR_USER    = 0x10  // bitmask implying user-defined (implemented with
                            // result handler functions)
} vmiFPRC;
```

Values `vmi_FPR_NEAREST`, `vmi_FPR_NEG_INF`, `vmi_FPR_POS_INF` and `vmi_FPR_ZERO` are standard rounding modes specified in IEEE 754-2008. `vmi_FPR_AWAY` specifies round-to-nearest, ties-away rounding, as defined in IEEE 754-2008. `vmi_FPR_ODD` specifies round-to-odd (or Von Neumann) rounding, where inexact results always have the least significant fraction bit set.

A user-defined conversion operation may be implemented by combining value `vmi_FPR_USER` with one of the other rounding control values, and using a configuration with `fp16ResultCB` and/or `fp32ResultCB` and/or `fp64ResultCB` callbacks. In this case, the value to convert will be passed as the first argument of the result handler function and the rounding mode as the second argument (see example 2 below).

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

It is possible for the conversion to generate exceptions: for example, converting from a non-integral floating point source to an integer result will always generate a precision exception. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the conversion result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual operation.

Example

The OVP ARMM model uses `vmimtFConvertRR` for single precision to half precision conversion:

```
ARM_MORPH_FN(armEmitVCVT_HS_VFP) {

    if(executeFPCheck(state)) {

        vmiReg rd    = GET_VFP_SREG(state, r1);
        vmiReg rm    = GET_VFP_SREG(state, r2);
        vmiReg flags = GET_FLAGS(state);

        // point to top half of register when VCVTT
        if(state->attrs->highhalf) {
            rm = VMI_REG_DELTA(rm, 2);
        }

        vmimtFConvertRR(
            vmi_FT_16_IEEE_754, rd, vmi_FT_32_IEEE_754, rm, vmi_FPR_CURRENT,
            flags, 0
        );
    }
}
```

Notes and Restrictions

1. See the descriptions of functions `vmirtGetFConvertRRDesc` and `vmirtFConvertSimdRR` in the *VMI Run Time Function Reference* which allow equivalent functionality to be implemented in an embedded call.

9.16 vmimtFUnopRR, vmimtFUnopSimdRR

Prototypes

```
void vmimtFUnopRR(
    vmiFType      type,
    vmiFUnop      op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        flags,
    vmiFPConfigCP config
);

void vmimtFUnopSimdRR(
    vmiFType      type,
    Uns32         num,
    vmiFUnop      op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        flags,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point unary operation on an argument in register `fa`, writing the result in register `fd`, both of type `type`. For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd` and `fa` indicate the first register in a contiguous vector.

It is possible for the operation to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the operation result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual operation.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

Argument `op` is the unary operation to perform. The available unary floating point operations are specified using the `vmiFUnop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // BASIC ARITHMETIC OPERATIONS
    vmi_FMOV,       // d <- a
    vmi_FABS,       // d <- abs(a), signalling
    vmi_FQABS,      // d <- abs(a), IEEE 754-2008 (clear sign bit only)
    vmi_FNEG,       // d <- -a,     signalling
    vmi_FQNEG,      // d <- -a,     IEEE 754-2008 (toggle sign bit only)
    vmi_FRECIP,     // d <- 1/a
    vmi_FSQRT,      // d <- sqrt(a)
    vmi_FRSQRT,     // d <- 1/sqrt(a)

                    // ROUNDING OPERATIONS
    vmi_FCEIL,      // d <- roundTowardsPositiveInfinity(a)
    vmi_FFLOOR,     // d <- roundTowardsNegativeInfinity(a)
    vmi_FNEAREST,   // d <- roundToNearest(a)
    vmi_FTRUNC,     // d <- roundTowardsZero(a)
    vmi_FROUNDC,    // d <- roundUsingCurrentRoundingMode(a)
```

```
    vmi_FAWAY,      // d <- roundToNearestTiesToAway(a)

                    // TRIGONOMETRIC OPERATIONS
    vmi_FSIN,       // d <- sin(a)
    vmi_FCOS,       // d <- cos(a)

                    // LOGARITHMIC OPERATIONS
    vmi_FLOG2,      // d <- log2(a)

                    // USER-DEFINED OPERATIONS
    vmi_FUNUD,      // (implemented with result handler functions)

    vmi_FUNOP_LAST  // KEEP LAST

 } vmiFUnop;
```

A user-defined operation may be implemented by specifying an operation of `vmi_FUNUD` and a configuration with `fp32ResultCB` and/or `fp64ResultCB` callbacks (see example 2 below).

By default, the operation uses the currently active processor rounding mode. For operations other than `vmi_FCEIL`, `vmi_FFLOOR`, `vmi_FNEAREST`, `vmi_FTRUNC` and `vmi_FAWAY` (which have fixed rounding modes) a different fixed rounding mode may be specified by prefixing the operation with a call to function `vmimtFSetRounding`; for example, this sequence specifies a square root operation with rounding towards zero:

```
        vmimtFSetRounding(vmi_FPR_ZERO);
        vmimtFUnopRR(vmi_FT_32_IEEE_754, vmi_FSQRT, r1, r2, flags, 0);
```

Example 1

The OVP ARM model uses `vmimtFUnopRR` for vector unary floating point operations:

```
ARM_MORPH_FN(armEmitVUnop_F) {

    if(checkStateSDFPEnabled(state)) {

        Uns32     ebytes1 = SIMD_ELBYTES(state->info.r1);
        Uns32     ebytes2 = getVFPResultSize(state, ebytes1);
        Uns32     nels    = SIMD_NUM_ELS(state->info.r1);
        vmiFBinop op      = state->attrs->funop;

        vmiReg r1    = GET_SIMD_RD(state, r1, 0, 0, ebytes2);
        vmiReg r2    = GET_SIMD_RS(state, r2, 0, 0, ebytes2);
        vmiReg flags = GET_FLAGS(state);

        vmimtFUnopSimdRR(bytesToFType(ebytes1), nels, op, r1, r2, flags, 0);

        // extend result if required
        emitVFPExtend(state, r1, ebytes1, ebytes2);
    }
}
```

Example 2

The OVP RISC-V model implements user-defined `vmimtFUnopRR` operations for some operations using RMM rounding mode. For example, single-precision square root operations use a result handler shown below, which implement an interface to the SoftFloat IEEE Floating-Point Arithmetic Package. This is necessary because the RISC-V model allows the RMM rounding mode to be used for general operations (it is not restricted to conversions to integral values).

```
static void beforeFPInt(riscvP riscv, riscvRMDesc rm) {

    // map from RISC-V rounding mode to riscvRMDesc
    static const riscvRMDesc mapRM[] = {
        [0] = RV_RM_RTE,
        [1] = RV_RM_RTZ,
        [2] = RV_RM_RDN,
        [3] = RV_RM_RUP,
        [4] = RV_RM_RMM,
    };

    // if rounding mode is RV_RM_CURRENT, get the current value
    if(rm==RV_RM_CURRENT) {
        rm = mapRM[RD_CSR_FIELD(riscv, fcsr, frm)];
    }

    // set SoftFloat controls
    softfloat_roundingMode   = rm;
    softfloat_exceptionFlags = 0;
}

inline static void beforeFP(vmiProcessorP processor) {

    riscvP riscv = (riscvP)processor;

    beforeFPInt(riscv, riscv->fpActiveRM);
}

inline static void afterFP(vmiFPFlagsP setFlags) {
    setFlags->bits |= softfloat_exceptionFlags;
}

VMI_FP_32_RESULT_FN(riscvFSQRT32) {

    float32_t a = {args[0].u32};

    beforeFP(processor);
    float32_t result = f32_sqrt(a);
    afterFP(setFlags);

    return result.v;
}
```

Notes and Restrictions

1. See the descriptions of functions `vmirtGetFUnopRRDesc` and `vmirtFUnopSimdRR` in the *VMI Run Time Function Reference* which allow equivalent functionality to be implemented in an embedded call.

9.17 vmimtFBinopRRR, vmimtFBinopSimdRRR

Prototypes

```
void vmimtFBinopRRR(
    vmiFType      type,
    vmiFBinop     op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    vmiFPConfigCP config
);

void vmimtFBinopSimdRRR(
    vmiFType      type,
    Uns32         num,
    vmiFBinop     op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point binary operation on arguments in registers `fa` and `fb`, writing the result to register `fd`, all of type `type` (except when `op` is `vmi_FSCALEI`, in which case the second argument is an integer). For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd`, `fa` and `fb` indicate the first register in a contiguous vector.

It is possible for the operation to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the operation result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual operation.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

Argument `op` is the binary operation to perform. The available binary floating point operations are specified using the `vmiFBinop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // BASIC ARITHMETIC OPERATIONS
    vmi_FADD,       // d <- a + b
    vmi_FSUB,       // d <- a - b
    vmi_FMUL,       // d <- a * b
    vmi_FDIV,       // d <- a / b

                    // MIN/MAX OPERATIONS
    vmi_FMIN,       // d <- min(a, b)
    vmi_FMAX,       // d <- max(a, b)

                    // SCALE OPERATIONS
    vmi_FSCALEF,    // d <- a * 2^b (floating point b)
```

```
    vmi_FSCALEI,     // d <- a * 2^b (integer b)

                     // QUIET COMPARISON OPERATIONS
    vmi_FQCMPEQ,     // d <-  (a == b)     ? all_ones : all_zeros
    vmi_FQCMPNEQ,    // d <- !(a == b)     ? all_ones : all_zeros
    vmi_FQCMPLT,     // d <-  (a <  b)     ? all_ones : all_zeros
    vmi_FQCMPNLT,    // d <- !(a <  b)     ? all_ones : all_zeros
    vmi_FQCMPLE,     // d <-  (a <= b)     ? all_ones : all_zeros
    vmi_FQCMPNLE,    // d <- !(a <= b)     ? all_ones : all_zeros
    vmi_FQCMPGT,     // d <-  (a >  b)     ? all_ones : all_zeros
    vmi_FQCMPNGT,    // d <- !(a >  b)     ? all_ones : all_zeros
    vmi_FQCMPGE,     // d <-  (a >= b)     ? all_ones : all_zeros
    vmi_FQCMPNGE,    // d <- !(a >= b)     ? all_ones : all_zeros
    vmi_FQCMPOR,     // d <-  ordered(a,b) ? all_ones : all_zeros
    vmi_FQCMPNOR,    // d <- !ordered(a,b) ? all_ones : all_zeros

                     // SIGNALLING COMPARISON OPERATIONS
    vmi_FSCMPEQ,     // d <-  (a == b)     ? all_ones : all_zeros
    vmi_FSCMPNEQ,    // d <- !(a == b)     ? all_ones : all_zeros
    vmi_FSCMPLT,     // d <-  (a <  b)     ? all_ones : all_zeros
    vmi_FSCMPNLT,    // d <- !(a <  b)     ? all_ones : all_zeros
    vmi_FSCMPLE,     // d <-  (a <= b)     ? all_ones : all_zeros
    vmi_FSCMPNLE,    // d <- !(a <= b)     ? all_ones : all_zeros
    vmi_FSCMPGT,     // d <-  (a >  b)     ? all_ones : all_zeros
    vmi_FSCMPNGT,    // d <- !(a >  b)     ? all_ones : all_zeros
    vmi_FSCMPGE,     // d <-  (a >= b)     ? all_ones : all_zeros
    vmi_FSCMPNGE,    // d <- !(a >= b)     ? all_ones : all_zeros
    vmi_FSCMPOR,     // d <-  ordered(a,b) ? all_ones : all_zeros
    vmi_FSCMPNOR,    // d <- !ordered(a,b) ? all_ones : all_zeros

                     // USER-DEFINED OPERATIONS
    vmi_FBINUD,      // (implemented with result handler functions)

    vmi_FBINOP_LAST // KEEP LAST

} vmiFBinop;
```

Operations `vmi_FMIN` and `vmi_FMAX` differ from the IEEE definitions of `minNum` and `maxNum` in that if either argument is a `NaN` then the result is a `QNaN`. This difference is intentional: it allows an operation-specific `QNaN` handler to configure the precise result in this case (the IEEE 754 Specification allows alternative implementations in this case).

Unlike other binary operations, the second argument for operation `vmi_FSCALEI` is a signed integer, 2 bytes in size for 16-bit floating point types, 4 bytes in size for 32-bit floating point types and 8 bytes in size for 64-bit and 80-bit floating point types. If this instruction is being used in a SIMD context, then the integer vector alignment must match the floating point vector; in other words, the integers need to have 2-byte alignment for 16-bit floating point operands, 4-byte alignment for 32-bit floating point operands, 8-byte alignment for 64-bit floating point operands, and 10-byte alignment for 80-bit floating point operands.

Operations `vmi_FQCMPEQ` . . . `vmi_FQCMPNOR` implement the IEEE 754 `compareQuiet` operations. Operations `vmi_FSCMPEQ` . . . `vmi_FSCMPNOR` implement the IEEE 754 `compareSignalling` operations.

A user-defined operation may be implemented by specifying an operation of `vmi_FBINUD` and a configuration with `fp16ResultCB` and/or `fp32ResultCB` and/or `fp64ResultCB` callbacks (see example 2 below).

By default, the operation uses the currently active processor rounding mode. A different fixed rounding mode may be specified by prefixing the operation with a call to function `vmimtFSetRounding`; for example, this sequence specifies an add operation with rounding towards zero:

```
        vmimtFSetRounding(vmi_FPR_ZERO);
        vmimtFBinopRRR(vmi_FT_32_IEEE_754, vmi_FADD, r1, r2, r3, flags, 0);
```

Example 1

The OVP ARM model uses `vmimtFBinopSimdRRR` for vector binary floating point operations:

```
ARM_MORPH_FN(armEmitVBinop_F) {

    if(checkStateSDFPEnabled(state)) {

        Uns32      ebytes1 = SIMD_ELBYTES(state->info.r1);
        Uns32      ebytes2 = getVFPResultSize(state, ebytes1);
        Uns32      nels    = SIMD_NUM_ELS(state->info.r1);
        vmiFBinop op       = state->attrs->fbinop;
        Bool       negate  = state->attrs->negate;

        vmiReg r1    = GET_SIMD_RD(state, r1, 0, 0, ebytes2);
        vmiReg r2    = GET_SIMD_RS(state, r2, 0, 0, ebytes2);
        vmiReg r3    = GET_SIMD_RS(state, r3, 0, 0, ebytes2);
        vmiReg flags = GET_FLAGS(state);

        vmimtFBinopSimdRRR(bytesToFType(ebytes1), nels, op, r1, r2, r3, flags, 0);

        // If negate attribute is selected negate the result
        if(negate) {
            vmimtFUnopSimdRR(
                bytesToFType(ebytes1), nels, vmi_FQNEG, r1, r1, flags, 0
            );
        }

        // extend result if required
        emitVFPExtend(state, r1, ebytes1, ebytes2);
    }
}
```

Example 2

The OVP RISC-V model implements user-defined `vmimtFBinopRRR` operations for some operations using RMM rounding mode. For example, single-precision add operations use a result handler shown below, which implement an interface to the SoftFloat IEEE Floating-Point Arithmetic Package. This is necessary because the RISC-V model allows the RMM rounding mode to be used for general operations (it is not restricted to conversions to integral values).

```
static void beforeFPInt(riscvP riscv, riscvRMDesc rm) {

    // map from RISC-V rounding mode to riscvRMDesc
    static const riscvRMDesc mapRM[] = {
        [0] = RV_RM_RTE,
        [1] = RV_RM_RTZ,
        [2] = RV_RM_RDN,
        [3] = RV_RM_RUP,
        [4] = RV_RM_RMM,
    };

    // if rounding mode is RV_RM_CURRENT, get the current value
```

```
    if(rm==RV_RM_CURRENT) {
        rm = mapRM[RD_CSR_FIELD(riscv, fcsr, frm)];
    }

    // set SoftFloat controls
    softfloat_roundingMode   = rm;
    softfloat_exceptionFlags = 0;
}

inline static void beforeFP(vmiProcessorP processor) {

    riscvP riscv = (riscvP)processor;

    beforeFPInt(riscv, riscv->fpActiveRM);
}

inline static void afterFP(vmiFPFlagsP setFlags) {
    setFlags->bits |= softfloat_exceptionFlags;
}

VMI_FP_32_RESULT_FN(riscvFADD32) {

    float32_t a = {args[0].u32};
    float32_t b = {args[1].u32};

    beforeFP(processor);
    float32_t result = f32_add(a, b);
    afterFP(setFlags);

    return result.v;
}
```

Notes and Restrictions

1. See the descriptions of functions `vmirtGetFBinopRRRDesc` and `vmirtFBinopSimdRRR` in the *VMI Run Time Function Reference* which allow equivalent functionality to be implemented in an embedded call.

9.18 vmimtFTernopRRRR, vmimtFTernopSimdRRRR

Prototypes

```
void vmimtFTernopRRRR(
    vmiFType      type,
    vmiFTernop    op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        fc,
    vmiReg        flags,
    Bool          roundInt,
    vmiFPConfigCP config
);

void vmimtFTernopSimdRRRR(
    vmiFType      type,
    Uns32         num,
    vmiFTernop    op,
    vmiReg        fd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        fc,
    vmiReg        flags,
    Bool          roundInt,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point ternary operation on arguments in registers `fa`, `fb` and `fc`, writing the result in register `fd`, all of type `type`. For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fd`, `fa`, `fb` and `fc` indicate the first register in a contiguous vector.

It is possible for the operation to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `fd` and `flags` will be updated with the operation result and flags. The flags are a bitwise-or of flags resulting from each individual operation.

Argument `op` is the ternary operation to perform. The available ternary floating point operations are specified using the `vmiFTernop` enumeration in `vmiTypes.h`:

```
typedef enum {
                    // UNFUSED OPERATION              FUSED OPERATION
    vmi_FMADD,      // d <- (a * b) + c               d <- ( a * b) + c
    vmi_FMSUB,      // d <- (a * b) - c               d <- ( a * b) - c
    vmi_FNMADD,     // d <- -((a * b) + c)            d <- (-a * b) + c
    vmi_FNMSUB,     // d <- -((a * b) - c)            d <- (-a * b) - c
    vmi_FMSUBR,     // d <- c - (a * b)               d <- (-a * b) + c
    vmi_FMADDH,     // d <- ((a * b) + c) / 2.0       d <- (( a * b) + c) / 2.0
    vmi_FMSUBH,     // d <- ((a * b) - c) / 2.0       d <- (( a * b) - c) / 2.0
    vmi_FMSUBRH,    // d <- (c - (a * b)) / 2.0       d <- ((-a * b) + c) / 2.0

                    // USER-DEFINED OPERATIONS
    vmi_FTERNUD,    // (implemented with result handler functions)
```

```
    vmi_FTERNOP_LAST// KEEP LAST

} vmiFTernop;
```

If `roundInt` is `True`, then each intermediate result will be rounded to the result type before being used as an operand to the next operation (in other words, the operation is *unfused*). The operations are carried out strictly in the precedence order implied by the `UNFUSED OPERATION` column in the table above: for example, `vmi_FMADDH` will first multiply `a` and `b`, then round the result, then add `c`, then round the result, then divide by 2. Unfused operations can therefore generate multiple rounding events.

If `roundInt` is `False`, then for x87 argument types intermediates are represented in the Intel x87 80-bit format, and for IEEE argument types intermediates are represented using infinite precision. This means that, for IEEE types, such operations correspond to the IEEE *fused-multiply-add* definition. In each case, the operations are carried out as indicated in the `FUSED OPERATION` column in the table above.

If non-`NULL`, argument `config` specifies an operation-specific configuration that overrides the default FPU configuration for this operation.

A user-defined operation may be implemented by specifying an operation of `vmi_FTERNUD` and a configuration with `fp16ResultCB` and/or `fp32ResultCB` and/or `fp64ResultCB` callbacks (see example 2 below).

By default, the operation uses the currently active processor rounding mode. A different fixed rounding mode may be specified by prefixing the operation with a call to function `vmimtFSetRounding`; for example, this sequence specifies an fused-multiply-add operation with rounding towards zero:

```
        vmimtFSetRounding(vmi_FPR_ZERO);
        vmimtFTernopRRRR(
            vmi_FT_32_IEEE_754, vmi_FMADD, r1, r2, r3, r4, flags, round, 0
        );
```

Example 1

The OVP ARM model uses `vmimtFTernopSimdRRRR` for vector ternary floating point operations:

```
ARM_MORPH_FN(armEmitVTernop_F) {

    if(checkStateSDFPEnabled(state)) {

        Uns32      ebytes1 = SIMD_ELBYTES(state->info.r1);
        Uns32      ebytes2 = getVFPResultSize(state, ebytes1);
        Uns32      nels    = SIMD_NUM_ELS(state->info.r1);
        Bool       round   = state->attrs->round;
        Bool       r4Used  = state->attrs->r4Used;
        vmiFTernop op      = state->attrs->fternop;

        vmiReg r1    = GET_SIMD_RD(state, r1, 0, 0, ebytes2);
        vmiReg r2    = GET_SIMD_RS(state, r2, 0, 0, ebytes2);
        vmiReg r3    = GET_SIMD_RS(state, r3, 0, 0, ebytes2);
        vmiReg r4    = r4Used ? GET_SIMD_RS(state, r4, 0, 0, ebytes2) : r1;
        vmiReg flags = GET_FLAGS(state);
```

```
        vmimtFTernopSimdRRRR(
            bytesToFType(ebytes1), nels, op, r1, r2, r3, r4, flags, round, 0
        );

        // extend result if required
        emitVFPExtend(state, r1, ebytes1, ebytes2);
    }
}
```

Example 2

The OVP RISC-V model implements user-defined `vmimtFTernopRRRR` operations for some operations using RMM rounding mode. For example, single-precision fused-multiply-add operations use a result handler shown below, which implement an interface to the SoftFloat IEEE Floating-Point Arithmetic Package. This is necessary because the RISC-V model allows the RMM rounding mode to be used for general operations (it is not restricted to conversions to integral values).

```
static void beforeFPInt(riscvP riscv, riscvRMDesc rm) {

    // map from RISC-V rounding mode to riscvRMDesc
    static const riscvRMDesc mapRM[] = {
        [0] = RV_RM_RTE,
        [1] = RV_RM_RTZ,
        [2] = RV_RM_RDN,
        [3] = RV_RM_RUP,
        [4] = RV_RM_RMM,
    };

    // if rounding mode is RV_RM_CURRENT, get the current value
    if(rm==RV_RM_CURRENT) {
        rm = mapRM[RD_CSR_FIELD(riscv, fcsr, frm)];
    }

    // set SoftFloat controls
    softfloat_roundingMode   = rm;
    softfloat_exceptionFlags = 0;
}

inline static void beforeFP(vmiProcessorP processor) {

    riscvP riscv = (riscvP)processor;

    beforeFPInt(riscv, riscv->fpActiveRM);
}

inline static void afterFP(vmiFPFlagsP setFlags) {
    setFlags->bits |= softfloat_exceptionFlags;
}

VMI_FP_32_RESULT_FN(riscvFMADD32) {

    float32_t a = {args[0].u32};
    float32_t b = {args[1].u32};
    float32_t c = {args[2].u32};

    beforeFP(processor);
    float32_t result = f32_mulAdd(a, b, c);
    afterFP(setFlags);

    return result.v;
}
```

Notes and Restrictions

1. See the descriptions of functions `vmirtGetFTernopRRRRDesc` and `vmirtFTernopSimdRRRR` in the *VMI Run Time Function Reference* which allow equivalent functionality to be implemented in an embedded call.

9.19 vmimtFCompareRR, vmimtFCompareSimdRR

Prototypes

```
void vmimtFCompareRR(
    vmiFType      type,
    vmiReg        relation,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    vmiFPConfigCP config
);

void vmimtFCompareSimdRR(
    vmiFType      type,
    Uns32         num,
    vmiReg        relation,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point comparison operation on arguments in register `fa` and `fb`, writing the result in register `relation`. The argument registers are of type `type`; output register `relation` is an 8-bit value of type `vmiFPRelation`, which enumerates the four exclusive relations in IEEE Standard 754:

```
typedef enum {
    vmi_FPRL_UNORDERED = 0x1,   // unordered
    vmi_FPRL_EQUAL     = 0x2,   // equal
    vmi_FPRL_LESS      = 0x4,   // less than
    vmi_FPRL_GREATER   = 0x8    // greater than
} vmiFPRelation;
```

For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fa`, `fb` and `relation` indicate the first register in a contiguous vector (of byte size, in the case of `relation`).

It is possible for the comparison to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `relation` and `flags` will be updated with the comparison result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual comparison.

Argument `allowQNaN` specifies the behavior for IEEE `QNaN` inputs. If `allowQNaN` is `True`, `QNaN` inputs will not cause exceptions and the `vmi_FPRL_UNORDERED` relation will result from comparisons containing these operands. Otherwise, if `allowQNaN` is `False`, `QNaN` inputs will be treated as an error and the invalid operation exception will be raised.

Example

The OVP ARM model uses `vmimtFCompareRR` for per-element vector compare operations:

```
static SIMD_EL_OP_FN(simdVCmpSelBool_F) {

    vmiFPRelation cond      = state->attrs->fpRelation;
    Bool          allowQNaN = state->attrs->allowQNaN;
    Uns64         ones      = allOnes(resultSize);
    vmiReg        relation  = getTemp(state, 32);
    vmiFlags      flags     = getZFFlags(VMI_REG_DELTA(relation, 1));

    // Compare the floating point operands
    vmimtFCompareRR(bytesToFType(opSize/8), relation, r2, r3, flags, allowQNaN, 0);

    // Move zeros/ones to result depending on whether any bits are set in relation
    vmimtBinopRRC(8, vmi_AND, VMI_NOREG, relation, cond, &flags);
    vmimtCondMoveRCC(resultSize, flags.f[vmi_ZF], False, result, ones, 0);
}
```

Notes and Restrictions

1. See also functions `vmimtFCompareRRC` and `vmimtFCompareSimdRRC` which allow result values to be specified in a more general form.
2. See the descriptions of functions `vmirtGetFCompareRRDesc` and `vmirtFCompareSimdRR` in the *VMI Run Time Function Reference* which allow equivalent functionality to be implemented in an embedded call.

9.20 vmimtFCompareRRC, vmimtFCompareSimdRRC

Prototypes

```
void vmimtFCompareRRC(
    Uns8          rdBits,
    vmiFType      type,
    vmiReg        rd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    Uns32         valueUN,
    Uns32         valueEQ,
    Uns32         valueLT,
    Uns32         valueGT,
    vmiFPConfigCP config
);

void vmimtFCompareSimdRRC(
    Uns8          rdBits,
    vmiFType      type,
    Uns32         num,
    vmiReg        rd,
    vmiReg        fa,
    vmiReg        fb,
    vmiReg        flags,
    Bool          allowQNaN,
    Uns32         valueUN,
    Uns32         valueEQ,
    Uns32         valueLT,
    Uns32         valueGT,
    vmiFPConfigCP config
);
```

Description

These functions emit code to perform a floating point comparison operation on arguments in registers `fa` and `fb`, writing the result in register `rd`. The argument registers are of type `type`; output register `rd` is of size `rdBits`. The value assigned to `rd` is selected as follows:

1. If the comparison between `fa` and `fb` produces an *unordered* result, `valueUN` is assigned to `rd`.
2. Otherwise, if `fa` is *equal to* `fb`, `valueEQ` is assigned to `rd`.
3. Otherwise, if `fa` is *less than* `fb`, `valueLT` is assigned to `rd`.
4. Otherwise (`fa` is *greater than* `fb`), `valueGT` is assigned to `rd`.

For the SIMD variant, argument `num` specifies the number of parallel operations (in the range 1 to 16) and arguments `fa`, `fb` and `rd` indicate the first register in a contiguous vector (of size `rdBits`, in the case of `rd`).

It is possible for the comparison to generate exceptions. If generated exceptions are not masked, the configured floating point exception handler will be called; otherwise, `rd` and `flags` will be updated with the comparison result and flags. For the SIMD variant, the flags are a bitwise-or of flags resulting from each individual comparison.

Argument `allowQNaN` specifies the behavior for IEEE `QNaN` inputs. If `allowQNaN` is `True`, `QNaN` inputs will not cause exceptions and an *unordered* result will generated for comparisons containing these operands. Otherwise, if `allowQNaN` is `False`, `QNaN` inputs will be treated as an error and the invalid operation exception will be raised.

These functions generalize the behavior of `vmimtFCompareRR` and `vmimtFCompareSimdRR`. For example,

```
vmimtFCompareRR(
    type, relation, fa, fb, flags, allowQNaN, 0
);
```

is exactly equivalent to:

```
vmimtFCompareRRC(
    8, type, relation, fa, fb, flags, allowQNaN,
    vmi_FPRL_UNORDERED, vmi_FPRL_EQUAL, vmi_FPRL_LESS, vmi_FPRL_GREATER,
    0
);
```

Example

This example is derived from the OVP ARM model. The comparison is designed to directly assign to a value of type `armArithFlags` in the ARM processor structure.

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

// arithmetic flag indices
typedef enum armAFIE {
    AFI_Z,      // zero flag
    AFI_N,      // sign flag
    AFI_C,      // carry flag
    AFI_V,      // overflow flag
    AFI_LAST,   // KEEP LAST: for sizing
} armAFI;

// arithmetic flags
typedef struct armArithFlagsS {
    Uns8 f[AFI_LAST];
} armArithFlags, *armArithFlagsP;

//
// Return a mask bit that sets the given flag in an armArithFlags structure
//
#define FLAG_MASK(_ID)  (1<<(_ID*8))

//
// Compare fa to fb, setting armFlags and flags
//
void armEmitFCompareRRF(
    armMorphStateP state,
    vmiFType       type,
    vmiReg         armFlags,
    vmiReg         fa,
    vmiReg         fb,
    Bool           allowQNaN
) {
    // do prologue actions
```

```
    armStartFPOperation(state);

    // do the compare
    vmimtFCompareRRC(
        32,
        type,
        armFlags,
        fa,
        fb,
        ARM_FP_STICKY,
        allowQNaN,
        FLAG_MASK(AFI_C) | FLAG_MASK(AFI_V),    // unordered
        FLAG_MASK(AFI_Z) | FLAG_MASK(AFI_C),    // equal
        FLAG_MASK(AFI_N),                       // less
        FLAG_MASK(AFI_C),                       // greater
        armGetOpConfig(state)
    );
}

// processor structure
typedef struct armS {
    . . .
    armArithFlags  aflags;              // arithmetic flags
    armArithFlags  sdfpAFlagsAA32;      // FPU comparison flags (AArch32 state)
    . . .
} arm;

//
// morph-time macros to calculate variable offsets to flags in an arm structure
//
#define ARM_AFLAGS              ARM_CPU_REG(aflags)
#define ARM_AFLAGS_AA32         ARM_CPU_REG(sdfpAFlagsAA32)

//
// Common code to execute VFP VCMP instructions
//
static void emitVCmpVFP(armMorphStateP state, vmiReg rd, vmiReg rm, Uns32 ebytes) {

    Bool   allowQNaN = state->attrs->allowQNaN;
    vmiReg armFlags  = IS_AARCH64(state) ? ARM_AFLAGS : ARM_AFLAGS_AA32;

    armEmitFCompareRRF(state, bytesToFType(ebytes), armFlags, rd, rm, allowQNaN);
}
```

Notes and Restrictions

1. See the descriptions of functions `vmirtGetFCompareRRCDesc` and `vmirtFCompareSimdRR` in the *VMI Run Time Function Reference* which allow equivalent functionality to be implemented in an embedded call.

9.21 vmimtFStart, vmimtFEnd

Prototypes

```
void vmimtFStart(void);
void vmimtFEnd(vmiReg flags, vmiFPConfigCP config);
```

Description

These functions enable definition of *compound floating point operations*, which are composed of a number of other floating point or general morph-time primitives. They provide a very powerful mechanism allowing the standard floating point primitives to be extended in a natural way with user-defined emulated instructions.

The general flow is as follows:

1. `vmimtFStart` is called to indicate the start of a compound operation.
2. A series of general `vmimt` primitives is used, which can operate on *true processor registers* or *processor temporaries*.
3. `vmimtFEnd` is called to indicate the end of the compound operation.

Function `vmimtFEnd` takes a flags register and a floating point configuration structure as arguments. Within the floating point compound operation block, any updates to true processor registers in the processor structure described by the morph-time primitives *do not take effect immediately*: instead, updates are recorded in a scratch structure. Then, when the end of the floating point block is encountered, the simulator first checks for any enabled floating point exceptions. If there are such exceptions, the model floating point exception handler is called. Otherwise, changes in the scratch structure are written back to the processor structure. This allows compound operations to be described in a natural way without perturbing the processor state if an enabled exception is taken.

If floating point primitives are used within the compound operation block, then any floating point exception handler associated with the configuration of those instructions is ignored: the exception handler associated with the configuration given as an argument to `vmimtFEnd` is used instead. Configurations for multiple floating point operations used within a compound block will normally indicate that result flags are sticky (`stickyFlags` is `True`).

The only fields used from the floating point configuration passed to `vmimtFEnd` are `fpArithExceptCB` (defining the floating point exception handler), `stickyFlags` (indicating whether the composite operation result flags are sticky or not) and `perElementFlags` (indicating whether operation result flags should be aggregated or reported separately for each SIMD operation). When `perElementFlags` is `True`, the separately-recorded operation flags are *not* written to the scratch structure but are instead written to the true processor structure, which means that they are easily available in the floating point exception handler. This is the only part of the processor structure that is modified before exceptions are taken.

Examples

This example shows how a standard VMI SIMD floating point operation could be recoded as a compound operation. In practice it would always be better to use available SIMD primitives if possible - this example is therefore for API clarification only and not a recommended modeling style.

Original SIMD Code

```
// floating-point configuration with non-sticky flags result
static vmiFPConfig opConfig = {
    . . .
    .stickyFlags     = False,
    .fpArithExceptCB = handleFPExceptions
};

// fd = fa + fb, setting non-sticky flags CPU_FFLAGS
static void emitSIMDFAdd32(vmiReg fd, vmiReg fa, vmiReg fb, Uns32 num) {
    vmimtFBinopSimdRRR(
        vmi_FT_32_IEEE_754,
        num,
        vmi_FADD,
        fd,
        fa,
        fb,
        CPU_FFLAGS,
        &opConfig
     );
}
```

Equivalent Compound Operation

```
// floating-point configuration with non-sticky flags result
static vmiFPConfig opConfig = {
    . . .
    .stickyFlags     = False,
    .fpArithExceptCB = handleFPExceptions
};

// floating-point configuration with sticky flags result used within
// compound floating point operation
static vmiFPConfig innerConfig = {
    . . .
    .stickyFlags = True
};

// fd = fa + fb, setting non-sticky flags CPU_FFLAGS
static void emitSIMDFAdd32(vmiReg fd, vmiReg fa, vmiReg fb, Uns32 num) {

    // start compound operation
    vmimtFStart();

    // perform individual adds accumulating sticky flags
    Uns32 i;
    for(i=0; i<num; i++) {
        vmimtFBinopRRR(
            vmi_FT_32_IEEE_754,
            vmi_FADD,
            VMI_REG_DELTA(fd, i*4),
            VMI_REG_DELTA(fa, i*4),
            VMI_REG_DELTA(fb, i*4),
            CPU_FFLAGS,
            &innerConfig
         );
    }
```

```
    // complete operation, committing results if required
    vmimtFEnd(CPU_FFLAGS, &opConfig);
}
```

Loops in Compound Floating Point Operations

It is possible to use compound operations in combination with `vmimtDJNZLabel` loops and indexed registers. This example shows such a loop, assuming per-element operation flags are not required:

```
static void emitVectorFOp(
    vmiFBinop op,
    Uns32     bits,
    Uns32     rd,
    Uns32     ra,
    Uns32     rb
) {
    vmiLabelP repeat   = vmimtNewLabel();
    Uns32     vecSize  = sizeof(cpuVec);
    Uns32     elemSize = bits/8;
    Uns32     elemNum  = vecSize/elemSize;
    vmiReg    base     = CPU_VR_BASE;
    vmiReg    index    = CPU_VR_INDEX(0);
    vmiReg    fdE      = CPU_VR(rd);
    vmiReg    faE      = CPU_VR(ra);
    vmiReg    fbE      = CPU_VR(rb);
    vmiReg    flags    = CPU_FR(1);

    // prepare indexed registers
    vmimtGetIndexedRegister(&fdE, &base, vecSize);
    vmimtGetIndexedRegister(&faE, &base, vecSize);
    vmimtGetIndexedRegister(&fbE, &base, vecSize);

    // initialize repeat count
    vmimtMoveRC(32, index, elemNum);

    // start compound operation
    vmimtFStart();

    // loop to here
    vmimtInsertLabel(repeat);

    // do operation
    vmimtFBinopRRR(vmi_FT_32_IEEE_754, op, fdE, faE, fbE, flags, &config);

    // prepare for next iteration
    vmimtAddBaseC(base, elemSize, 0);
    vmimtDJNZLabel(32, index, repeat);

    // end compound operation
    vmimtFEnd(elemNum, flags, &config);
}
```

If per-element floating point flags are required, a separate *base register* must be used to iterate across the operation flag members. In addition, separate `vmiReg` structures are required to identify *indexed members* of the operation flags array within the loop and the *composite result flag* outside the loop:

```
static void emitVectorFOp(
    vmiFBinop op,
    Uns32     bits,
    Uns32     rd,
    Uns32     ra,
    Uns32     rb
) {
    vmiLabelP repeat    = vmimtNewLabel();
```

```
    Uns32     vecSize   = sizeof(cpuVec);
    Uns32     elemSize  = bits/8;
    Uns32     elemNum   = vecSize/elemSize;
    Uns32     flagSize  = 1;
    vmiReg    base      = CPU_VR_BASE(0);
    vmiReg    baseF     = CPU_VR_BASE(1);
    vmiReg    index     = CPU_VR_INDEX(0);
    vmiReg    fdE       = CPU_VR(rd);
    vmiReg    faE       = CPU_VR(ra);
    vmiReg    fbE       = CPU_VR(rb);
    vmiReg    loopFlags = CPU_FR(rd);
    vmiReg    cumFlags  = CPU_FR(rd);

    // prepare indexed registers
    vmimtGetIndexedRegister(&fdE,       &base,  vecSize);
    vmimtGetIndexedRegister(&faE,       &base,  vecSize);
    vmimtGetIndexedRegister(&fbE,       &base,  vecSize);
    vmimtGetIndexedRegister(&loopFlags, &baseF, flagSize);

    // initialize repeat count
    vmimtMoveRC(32, index, elemNum);

    // start compound operation
    vmimtFStart();

    // loop to here
    vmimtInsertLabel(repeat);

    // do operation
    vmimtFBinopRRR(vmi_FT_32_IEEE_754, op, fdE, faE, fbE, loopFlags, &config);

    // prepare for next iteration
    vmimtAddBaseC(base,  elemSize, 0);
    vmimtAddBaseC(baseF, flagSize, 0);
    vmimtDJNZLabel(32, index, repeat);

    // end compound operation
    vmimtFEnd(elemNum, cumFlags, &config);
}
```

If any operation within the compound operation block targets *a true processor register* (not a temporary) then the only control flow operations allowed within the block are `vmimtDJNZLabel` loops. In addition, the simulator will always assume that the *entire target register* must be copied back from the scratch structure at the end of the compound operation, so care must be taken to ensure that the vector size is correctly specified in the `vmimtGetIndexedRegister` calls and that all SIMD elements are updated in the loop.

Sometimes, more flexibility than this is required: for example, when defining *vector* operations, it may be the case that the vector size is not static (it is determined by a control register) or that updated vector members are not in a contiguous range, or that different operations should apply to different vector members (requiring control flow operations other than `vmimtDJNZLabel` loops). To implement such operations, use a modified flow, as follows:

1. `vmimtFStart` is called to indicate the start of the compound operation.
2. A series of general `vmimt` primitives within a loop is used, which can operate on true processor registers or processor temporaries *but target only processor temporaries* (with the exception of flags, which may be true registers).
3. `vmimtFEnd` is called to indicate the end of the compound operation, and cause any enabled exceptions to be taken.

4. General `vmimtMoveRR` primitives are called within a second loop to commit results from temporaries to true processor registers.

When operations within a compound block target only temporaries as described above, it is permitted to use labeled branches and jumps within those compound blocks (so members can be treated differently). Because the commit of results is performed explicitly, any subset of vector members can be updated (the only requirement being that the members addressed in the commit phase match those written in the operation phase).

Notes and Restrictions

1. A call to `vmimtFStart` is illegal within a compound floating point operation block (blocks may not be nested).
2. With the exception of `vmimtDJNZLabel` loops, calls to any control-flow modifying primitives are illegal within compound floating point operation blocks, if operations in those blocks target true processor registers (see above). Specifically, inter-instruction jump primitives (e.g. `vmimtCondJump`) and intra-instruction primitives (e.g. `vmimtCondJumpLabel`) may not be used.
3. Calls to memory load/store primitives are are illegal within compound floating point operation blocks.
4. It is possible to call embedded functions within compound blocks, but care must be taken to ensure that the called functions do not directly modify processor state or directly refer to processor state that has already been modified within the compound block. This will result either in the embedded function updating state when it should not (if an exception is taken) or referencing a stale register value (because the correct value is pending in a scratch area).
5. When used in combination with `vmimtDJNZLabel` loops, code is generated assuming that *entire target registers* of the size specified in the `vmimtGetIndexedRegister` call are updated. Be careful to ensure the specified register size is correct and fully written inside the loop.

10 Miscellaneous Operations

This section describes miscellaneous emission functions for simulator control and instruction counting.

10.1 vmimtHalt

Prototype

```
void vmimtHalt(void);
```

Description

This function emits code that halts execution for the current processor. It is used to simulate hardware halt instructions.

A processor that has been halted may be restarted by a call to the run time functions `vmirtRestartNow` or `vmirtRestartNext` (defined in `vmiRt.h`). This call will typically be made within an event handler routine or as a call made by the implementation of a special instruction executed by another processor in the simulated platform.

Example

The OVP ARM model uses this function for halt, `WFI` and `WFE` operations. A field on the processor called `disable` is used to hold the current reason why the processor is not executing:

```
void armEmitHalt(armDisableReason reason) {
    vmimtMoveRC(8, ARM_DISABLE, reason);
    vmimtHalt();
}
```

Notes and Restrictions

None.

10.2 vmimtYield

Prototype

```
void vmimtYield(void);
```

Description

This function emits code that explicitly suspends execution of the current processor on completion of the current simulated instruction to allow other processors in a multiprocessor simulation to run. The processor will run again when all other runnable processors have executed. Unlike related function `vmimtIdle`, the processor will resume execution in the *current* time slice.

This function is useful only for modeling artifact behavior that requires processors to run in a particular order. It should not normally be required or used in processor models, but may be useful in binary intercept libraries.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. See related function `vmimtIdle` which causes the processor to execute in the *next* time slice.

10.3 vmimtIdle

Prototype

```
void vmimtIdle(void);
```

Description

This function emits code that explicitly suspends execution of the current processor on completion of the current simulated instruction, and advances processor time immediately to the end of the scheduled time slot or until the next timer event for that processor, as if it had executed nop instructions in the interim.

This function is useful only for modeling artifact behavior that requires processors to run in a particular order. It should not normally be required or used in processor models, but may be useful in binary intercept libraries.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. See related function `vmimtYield` which causes the processor to execute in the *same* time slice.

10.4 vmimtInterrupt

Prototype

```
void vmimtInterrupt(void);
```

Description

This function causes simulation to stop on completion of the current simulated instruction and return from the calling context. It is intended for use in intercept libraries.

When using the OP interface, the function will cause a return from `opProcessorSimulate` or `opRootModuleSimulate` with a stop reason of `OP_SR_INTERRUPT`.

When using the legacy ICM interface, the function will cause a return from `icmSimulate` or `icmSimulatePlatform` with a stop reason of `ICM_SR_INTERRUPT`.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

None.

10.5 vmimtExit

Prototype

```
void vmimtExit(void);
```

Description

This simulation control function emits code that ends execution for the current processor. In a multiprocessor simulation, remaining processors will continue execution. If the current processor is the only running processor, then the simulation run will end.

If the processor is a member of a cluster or SMP group, then behavior is determined by the leaf-level `exitmode` parameter on the group. If `exitmode` is `first` (the default), then the *first* group member that exits will cause *all* members of the group to exit. If `exitmode` is all, then exit will affect only the current processor; other members of the cluster or SMP group will continue to run.

This function is typically used to model special trap operations intended to cause simulation termination, or to handle situations such as decoded but unimplemented instructions while processor models are under development.

Example

The OVP ARM model uses this function to exit simulation for a decoded but unimplemented instruction:

```
static void emitUnimplemented(armMorphStateP state) {
    vmimtArgProcessor();
    vmimtArgSimPC(ARM_GPR_BITS);
    vmimtCall((vmiCallFn)unimplemented);
    vmimtExit();
}
```

Notes and Restrictions

None.

10.6 vmimtFinish

Prototype

```
void vmimtFinish(void);
```

Description

This simulation control function emits code that ends simulation. Note that this is different to `vmimtExit` because simulation will end even if other processors in a multiprocessor platform are still running.

This function is typically used to model special trap operations intended to cause simulation termination, or to handle situations such as decoded but unimplemented instructions while processor models are under development.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

None.

10.7 vmimtEndBlock

Prototype

```
void vmimtEndBlock(void);
```

Description

This simulation control function forces the current native code block to be terminated after the current simulated instruction – the next simulated instruction is guaranteed to be in a different code block. The circumstances in which this might be useful are described below.

As explained in the section *Interaction with Imperas Simulators*, the simulator maintains native translations of simulated instructions in *code blocks* in a dictionary. The bounds of each code block are determined as follows:

1. Any instruction targeted by a jump instruction starts a code block.
2. Any jump instruction terminates a code block.
3. Large code blocks are also restricted to a maximum of 256 bytes.

The simulator automatically determines the bounds of code blocks as it executes, and usually the rules described above are adequate. There are, however, some circumstances that cause problems:

1. *Mode changes within a code block*
   The simulator can maintain many dictionaries for each simulated processor. This is useful because often instructions have different behavior depending on processor mode: for example, an instruction that writes to a status register may succeed in kernel mode but cause a privileged instruction trap in user mode. By maintaining separate dictionaries for user and kernel modes, the decision about the instruction behavior can be made at *morph time* instead of *run time*, which means the native code is much more efficient.
   If an instruction is executed that causes a mode change, then the next instruction must by definition be in a different code block (because it must be in a different dictionary). In this case, `vmimtEndBlock` should be called to force the current code block to be terminated.

2. *Block mask or polymorphic key changes within a code block*
   The simulator can also maintain alternative translations for particular instruction patterns. In some cases, the choice is made *statically* using a *block mask* idiom (see section 10.11); in other cases, the choice is made *dynamically* using a *polymorphic key* idiom (see section 10.14). In either case, when any operation is performed that changes the active block mask or polymorphic key, `vmimtEndBlock` should be called to force the current code block to be terminated.

3. *Bogus instruction patterns*
   A simulation may set unused memory to a specific pattern, for example `0xdeadbeef`. This pattern usually coincides with a possibly legal but highly unlikely instruction (although the simulator has no way of knowing this).
   If during simulation a branch is made to uninitialized memory (because of an application program error), the simulator will start translation of instructions with the unused memory pattern.
   If the pattern corresponds to an instruction that is not a branch, then the effect will be to create a very large code block consisting of very many repetitions of the translated unused pattern, which can cause an apparent simulator freeze while the code block is processed.
   For performance reasons, it is therefore sensible to terminate the current code block whenever the unlikely-but-legal instruction is encountered.

Example

The OVP RISC-V model uses this function when code is emitted to write a CSR. For example, a write to the `misa` CSR ends the current code block:

```
riscvArchitecture riscvEmitCSRWrite(
    riscvCSRId id,
    riscvP     riscv,
    vmiReg     rs,
    vmiReg     tmp
) {
    riscvArchitecture arch    = riscv->currentArch;
    csrAttrsCP        attrs   = &csrs[id];
    Uns32             bits    = riscvGetXlenMode(riscv);
    riscvCSRWriteFn   writeCB = getCSRWriteCB(id, riscv, bits);
    vmiReg            raw     = getRawArch(attrs, arch);
    Uns64             mask    = getCSRWriteMask(attrs, riscv);

    // indicate that this register has been written
    vmimtRegWriteImpl(attrs->name);

    if(writeCB) {

        // if CSR is implemented externally, mirror the result into any raw
        // register in the model (otherwise discard the result)
        if(!csrImplementExternalWrite(id, riscv)) {
            raw = VMI_NOREG;
        }

        // emit code to call the write function (NOTE: argument is always 64
        // bits, irrespective of the architecture size)
        vmimtArgUns32(id);
        vmimtArgProcessor();
        vmimtArgRegSimAddress(bits, rs);
        vmimtCallResult((vmiCallFn)writeCB, bits, raw);

        // terminate the current block if required
        if(attrs->wEndBlock) {
            vmimtEndBlock();
        }

    } else if(VMI_ISNOREG(raw)) {

        // emit warning for unimplemented CSR
        emitWarnUnimplementedCSR(id, riscv);

    } else if(mask==-1) {

        // new value is written unmasked
```

```
        vmimtMoveRR(bits, raw, rs);

    } else if(mask) {

        // apparent reads of register below are artifacts only
        vmimtRegNotReadR(bits, raw);

        // new value is written masked
        vmimtBinopRC(bits, vmi_ANDN, raw, mask, 0);
        vmimtBinopRRC(bits, vmi_AND, tmp, rs, mask, 0);
        vmimtBinopRR(bits, vmi_OR, raw, tmp, 0);
    }

    // return architectural constraints that apply to this register
    return attrs->arch;
}
```

Notes and Restrictions

None.

10.8 vmimtGetBlockMask

Prototype

```
void vmimtGetBlockMask(vmiReg blockMask);
```

Description

This simulation control function copies the current processor *block mask* to the 32-bit register `blockMask`. The block mask is used to validate that assumptions made when the current block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. The assumptions for the current block are specified using `vmimtValidateBlockMask`.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details and an example of the use of this function.

10.9 vmimtSetBlockMaskC

Prototype

```
void vmimtSetBlockMaskC(Uns32 blockMask);
```

Description

This simulation control function sets the current processor *block mask* to the value `blockMask`. The block mask is used to validate that assumptions made when the current block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. The assumptions for the current block are specified using `vmimtValidateBlockMask`.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details and an example of the use of this function.

10.10 vmimtSetBlockMaskR

Prototype

```
void vmimtSetBlockMaskR(vmiReg blockMask);
```

Description

This simulation control function sets the current processor *block mask* to the value of the 32-bit register `blockMask`. The block mask is used to validate that assumptions made when the current block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. The assumptions for the current block are specified using `vmimtValidateBlockMask`.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details and an example of the use of this function.

10.11 vmimtValidateBlockMask

Prototype

```
void vmimtValidateBlockMask(Uns32 modeMask);
```

Description

This simulation control function is used in combination with the *VMI Run Time API* function `vmirtSetBlockMask` to validate that assumptions made when the block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. Block mask behavior is implemented in two parts:

1. At *morph time*, the current value of bits in the built-in 32-bit block mask selected using a bitwise-and with `modeMask` are recorded with the code block.
2. At *run time*, bits in the built-in 32-bit block mask are again selected using a bitwise-and with `modeMask`. These selected bits are then compared with the morph-time value saved with the block. If these values differ, the block is deleted and retranslated; otherwise, the block is executed.

Block masks are typically used to efficiently implement instructions whose behavior differs based on enable bits that are likely to have a constant value when a block is executed. For example, floating point instructions are typically enabled by a processor configuration bit: if enabled, the instruction executes normally; if disabled, an exception is taken. By encoding the enable bit in a blockmask that is checked using `vmirtValidateBlockMask`, morph time code can create *either* the floating point instruction code *or* code to take an exception, not both. This results in much more compact JIT code with fewer branches.

Example

The OVP ARM model uses this function to implement a stack alignment check in AArch64 mode:

```
void armEmitCheckSA(armMorphStateP state) {

    armP           arm        = state->arm;
    armBlockStateP blockState = state->blockState;

    // emit blockMask check of SA state
    vmimtValidateBlockMask(ARM_BM_SA);

    // determine if alignment check is required
    if(!blockState->alignedSP && (arm->blockMask & ARM_BM_SA)) {

        vmiLabelP ok = vmimtNewLabel();

        // after this instruction, the stack pointer will be aligned (otherwise
        // an exception will have been taken)
        blockState->alignedSP = True;

        // skip mode switch unless stack is misaligned
        vmimtTestRCJumpLabel(64, vmi_COND_Z, ARM_SP64(0), 0xf, ok);

        // emit call to stack alignment exception routine
```

```
        vmimtArgProcessor();
        vmimtCallAttrs((vmiCallFn)armSA, VMCA_EXCEPTION);

        // here if no stack alignment exception
        vmimtInsertLabel(ok);
    }
}
```

Notes and Restrictions

1. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details and an example of the use of this function.
2. See related function `vmimtValidateBlockMaskR`, which allows block masks to be validated using any processor register instead of the built-in 32-bit block mask.

10.12 vmimtValidateBlockMaskR

Prototype

```
void vmimtValidateBlockMaskR(
    Uns32  bits,
    vmiReg r,
    Uns64  modeMask
);
```

Description

This simulation control function is used to validate that assumptions made when a block was translated still apply when it is executed. If the assumptions no longer apply, the code block is automatically deleted and remorphed. Block mask behavior is implemented in two parts:

1. At *morph time*, the current value of bits in register `r` selected using a bitwise-and with `modeMask` are recorded with the code block.
2. At *run time*, bits in register `r` are again selected using a bitwise-and with `modeMask`. These selected bits are then compared with the morph-time value saved with the block. If these values differ, the block is deleted and retranslated; otherwise, the block is executed.

Block masks are typically used to efficiently implement instructions whose behavior differs based on enable bits that are likely to have a constant value when a block is executed. For example, floating point instructions are typically enabled by a processor configuration bit: if enabled, the instruction executes normally; if disabled, an exception is taken. By encoding the enable bit in a blockmask that is checked using `vmirtValidateBlockMaskR`, morph time code can create *either* the floating point instruction code *or* code to take an exception, not both. This results in much more compact JIT code with fewer branches.

This function differs from `vmimtValidateBlockMask` in two ways:

1. The block mask is held in a general processor register, not the built-in processor block mask; and
2. The block mask can be up to 64 bits (not 32 bits, like the built-in block mask).

Example

The OVP ARC model uses this function to implement zero-overhead loop instructions efficiently. These instructions loop based on an end address held in a register, and for efficiency the JIC-compiled code assumes the end address is the same each time the loop is executed. The block mask is used to validate this assumption:

```
void arcEmitStartZOL(arcMorphStateP state) {

    Uns32     bits       = ARC_GPR_BITS;
    vmiLabelP noLPUpdate = vmimtNewLabel();
    Uns32     lpcMask    = state->arc->lpcMask;
```

```
    // when executing this block, validate that lp_end has the same value
    // that it has when code for the block was generated
    vmimtValidateBlockMaskR(ARC_GPR_BITS, ARC_AUX_REG(lp_end), -1);

    // tag this block to avoid unnecessary deletion when lp_end changes
    vmimtTagBlock(VBT_1);

    // ARC700 and ARCv2 implement STATUS32.L flag, disabling zero-overhead loops
    if(isARC700v2(state->arc)) {

        // assume zero-overhead loop is disabled
        vmimtMoveRC(8, ARC_ZOL_BRANCH, 0);

        // go to the label if L bit is set
        vmimtCondJumpLabel(ARC_L, True, noLPUpdate);
    }

    // define flags to detect non-zero condition, when loop count is decremented
    vmiFlags flags = {
        VMI_NOFLAG,
        {
            [vmi_CF] = VMI_NOFLAG,
            [vmi_PF] = VMI_NOFLAG,
            [vmi_ZF] = ARC_ZOL_BRANCH,
            [vmi_SF] = VMI_NOFLAG,
            [vmi_OF] = VMI_NOFLAG
        },
        vmi_FN_ZF
    };

    // decrement loop count and perhaps mask it, generating non-zero flag
    if(state->arc->lpcMask==-1) {
        vmimtBinopRC(bits, vmi_SUB, ARC_LP_COUNT, 1, &flags);
    } else {
        vmimtBinopRC(bits, vmi_SUB, ARC_LP_COUNT, 1, 0);
        vmimtBinopRC(bits, vmi_AND, ARC_LP_COUNT, lpcMask, &flags);
    }

    if(isARC600(state->arc)) {

        // ARC600 loop terminates if pre-decrement value is either 0 or 1, so
        // include detection of *post-decrement* value -1
        vmiReg tf = ARC_TEMP(state->tempIdx+1);
        vmimtCompareRC(bits, vmi_COND_NE, ARC_LP_COUNT, lpcMask, tf);
        vmimtBinopRR(bits, vmi_AND, ARC_ZOL_BRANCH, tf, 0);

    } else if(state->inDelaySlot) {

        // on ARC700 and ARCv2, if in a delay slot, only branch if STATUS32.DE
        // is zero
        vmimtBinopRR(8, vmi_ANDN, ARC_ZOL_BRANCH, ARC_DE, 0);
    }

    // here if zero-overhead loops are disabled
    vmimtInsertLabel(noLPUpdate);
}
```

Notes and Restrictions

1. `bits` must be 8, 16, 32 or 64.
2. If a simulated instruction modifies register `r`, *ensure that the current code block is terminated using `vmimtEndBlock`*. If this is not done, then subsequent simulated instructions in the code block may operate incorrectly if their behavior depends on modified bits in the block mask register.
3. See the description of `vmirtSetBlockMask` in the *VMI Run Time Function Reference* for extensive details on the behavior of block masks.

10.13 vmimtTagBlock

Prototype

```
void vmimtTagBlock(vmiBlockTag tag);
```

Description

This simulation control function is used to tag a block at morph time to enable it to be conditionally preserved or deleted by a later call to VMI run time function `vmirtFlushTargetModeTagged`. The `tag` argument is an enumerated type defined in `vmiTypes.h`:

```
typedef enum vmiBlockTagE {
    VBT_NA = 0,         // no tag
    VBT_1  = (1<<0),    // tag 1
    VBT_2  = (1<<1),    // tag 2
    VBT_3  = (1<<2),    // tag 3
    VBT_4  = (1<<3),    // tag 4
    VBT_5  = (1<<4),    // tag 5
    VBT_6  = (1<<5),    // tag 6
    VBT_7  = (1<<6),    // tag 7
    VBT_8  = (1<<7),    // tag 8
} vmiBlockTag;
```

When a block is tagged, the tag value is combined with any existing tags for that block using bitwise-or. This means that multiple calls to `vmimtTagBlock` add tags cumulatively to the current code block, with up to eight distinct tags supported.

Example

This example is from the OVP ARC processor model. The ARC processor implements a *zero-overhead loop* construct in which a code block can only be reused if a limiting address held in a register holds a certain value. Setting the register to a new address must invalidate any blocks at that address unless they implement zero-overhead loop behavior, in which case they can be preserved.

Unnecessary block flushes are suppressed by using tagged blocks as follows:

```
void arcEmitStartZOL(arcMorphStateP state) {

    Uns32     bits       = ARC_GPR_BITS;
    vmiLabelP noLPUpdate = vmimtNewLabel();
    Uns32     lpcMask    = state->arc->lpcMask;

    // when executing this block, validate that lp_end has the same value
    // that it has when code for the block was generated
    vmimtValidateBlockMaskR(ARC_GPR_BITS, ARC_AUX_REG(lp_end), -1);

    // tag this block to avoid unnecessary deletion when lp_end changes
    vmimtTagBlock(VBT_1);

    // ARC700 and ARCv2 implement STATUS32.L flag, disabling zero-overhead loops
    if(isARC700v2(state->arc)) {

        // assume zero-overhead loop is disabled
        vmimtMoveRC(8, ARC_ZOL_BRANCH, 0);

        // go to the label if L bit is set
```

```
        vmimtCondJumpLabel(ARC_L, True, noLPUpdate);
    }

    // define flags to detect non-zero condition, when loop count is decremented
    vmiFlags flags = {
        VMI_NOFLAG,
        {
            [vmi_CF] = VMI_NOFLAG,
            [vmi_PF] = VMI_NOFLAG,
            [vmi_ZF] = ARC_ZOL_BRANCH,
            [vmi_SF] = VMI_NOFLAG,
            [vmi_OF] = VMI_NOFLAG
        },
        vmi_FN_ZF
    };

    // decrement loop count and perhaps mask it, generating non-zero flag
    if(state->arc->lpcMask==-1) {
        vmimtBinopRC(bits, vmi_SUB, ARC_LP_COUNT, 1, &flags);
    } else {
        vmimtBinopRC(bits, vmi_SUB, ARC_LP_COUNT, 1, 0);
        vmimtBinopRC(bits, vmi_AND, ARC_LP_COUNT, lpcMask, &flags);
    }

    if(isARC600(state->arc)) {

        // ARC600 loop terminates if pre-decrement value is either 0 or 1, so
        // include detection of *post-decrement* value -1
        vmiReg tf = ARC_TEMP(state->tempIdx+1);
        vmimtCompareRC(bits, vmi_COND_NE, ARC_LP_COUNT, lpcMask, tf);
        vmimtBinopRR(bits, vmi_AND, ARC_ZOL_BRANCH, tf, 0);

    } else if(state->inDelaySlot) {

        // on ARC700 and ARCv2, if in a delay slot, only branch if STATUS32.DE
        // is zero
        vmimtBinopRR(8, vmi_ANDN, ARC_ZOL_BRANCH, ARC_DE, 0);
    }

    // here if zero-overhead loops are disabled
    vmimtInsertLabel(noLPUpdate);
}
```

In a callback that is activated when the `lp_end` register changes, code blocks at the end address implied by the new value of the `lp_end` register are flushed using this idiom:

```
static void flushTargetZOL(arcP arc, Uns32 lpEnd) {

    for(mode=0; mode<ARC_MODE_LAST; mode++) {
        vmirtFlushTargetModeTagged(
            (vmiProcessorP)arc, lpEnd, mode, VBT_1, VBT_1, False
        );
    }
}
```

The call to `vmirtFlushTargetModeTagged` in this case will flush any code block at address lpEnd for which the expression

```
((block->tag & VBT_1) == VBT_1)
```

is `False`. This flushes any code block that was not tagged as a zero-overhead loop block when it was constructed.

Notes and Restrictions

None.

10.14 vmimtPolymorphicBlock

Prototype

```
vmimtPolymorphicBlock(Uns32 bits, vmiReg key);
```

Description

This simulation control function is used to indicate that the current code block is *polymorphic*, with variants selected based on the current value of the `key` register. When blocks are polymorphic, the simulator can maintain a number of different JIT translated blocks for the same address and select from them dynamically at run time based on the key. The key register can be either 8 bits (allowing up to 254 alternative block translations) or 16 bits (allowing up to 65534 alternative block translations). Key value 0 is reserved and indicates that a block is not polymorphic, which is the default simulator behavior.

Polymorphic blocks should be used when alternative versions of a block are *likely* to occur at run time and therefore maintaining these alternative versions using block masks is not practical (because switching between versions requires deletion and retranslation of the block, which is slow when done frequently). Polymorphic blocks have a run time penalty compared to non-polymorphic blocks because extra work is required to select the appropriate block at run time, but this is small compared to the cost of frequent retranslations.

Example

This example is from the OVP RISC-V processor model. The RISC-V processor vector extension implements instructions which are polymorphic. Their behavior depends on:

1. The active *standard element width* (8, 16, 32 or 64 bits); and
2. The active *vector length*; and
3. The active *vector length multiplier*.

All three controls can be varied independently. The behavior of a particular vector instruction is dynamically dependent on the current settings, meaning that the same instruction can behave differently as a program runs.

The processor model has a 16-bit field, `pmKey`, which holds the current polymorphic key:

```
//
// Processor model structure
//
typedef struct riscvS {
    . . . lines omitted for clarity . . .
    Uns16 pmKey;    // polymorphic key
    . . . lines omitted for clarity . . .
} riscv;
```

This field is updated when the processor `vl` or `vtype` control registers change. Register `vl` is the *current vector length*, while register `vtype` contains fields `vsew` (the *standard element width*) and `vlmul` (the *vector length multiplier*):

```
void riscvRefreshVectorPMKey(riscvP riscv) {

    Uns32 vl       = RD_CSR(riscv, vl);
    Uns32 SEW      = 8<<RD_CSR_FIELD(riscv, vtype, vsew);
    Uns32 VLMUL    = 1<<RD_CSR_FIELD(riscv, vtype, vlmul);
    Uns32 vlMax    = riscv->configInfo.VLEN*VLMUL/SEW;
    Uns32 vtypeKey = RD_CSR(riscv, vtype)<<2;
    Uns32 villKey  = RD_CSR_FIELD(riscv, vtype, vill)<<2;
    Uns32 pmKey;

    // set current maximum vl for SEW/VLMUL combination
    riscv->vlMax = (vl>vlMax) ? vlMax : vl;

    // compose key
    if(villKey) {
        pmKey = VLCLASSMT_UNKNOWN | villKey;
    } else if(!vl) {
        pmKey = VLCLASSMT_ZERO;
    } else if(riscv->vlMax==vlMax) {
        pmKey = VLCLASSMT_MAX | vtypeKey;
    } else {
        pmKey = VLCLASSMT_NONZERO | vtypeKey;
    }

    // update polymorphic key
    riscv->pmKey = (riscv->pmKey & ~PMK_VECTOR) | pmKey;
}
```

When RISC-V vector instructions are translated, the model checks the current value of the `pmKey` field using `vmimtPolymorphicBlock` and is then able to treat the current values of vector length, standard element width and vector multiplier as constants. All vector instructions are wrapped by a call to function `checkVectorOp`, as follows:

```
static void checkVectorOp(riscvMorphStateP state, iterDescP id) {

    riscvP riscv = state->riscv;
    Uns32  VLEN  = riscv->configInfo.VLEN;

    // fill operation-specific data
    id->VLMUL     = getVLMULMt(riscv);
    id->SEW       = getSEWMt(riscv);
    id->MLEN      = id->SEW/id->VLMUL;
    id->vBytesMax = VLEN * id->VLMUL / 8;

    dispatchVector(state, state->attrs->checkCB, id);
}
```

As an example, the constant vector length multiplier is obtained by a call to function `getVLMULMt`, defined as follows:

```
inline static void emitCheckPolymorphic(void) {
    vmimtPolymorphicBlock(16, RISCV_PM_KEY);
}

static riscvVLMULMt getVLMULMt(riscvP riscv) {

    riscvBlockStateP blockState = riscv->blockState;
    riscvVLMULMt     VLMUL      = blockState->VLMULMt;

    if(VLMUL==VLMULMT_UNKNOWN) {

        emitCheckPolymorphic();

        blockState->VLMULMt = VLMUL =
            vlmulToVLMUL(RD_CSR_FIELD(riscv, vtype, vlmul));
```

```
    }

    return VLMUL;
}
```

A similar idiom is used to obtain the standard element width and vector length. Code is then generated for the current block assuming that these three values are constant. If they change, the simulator will call the RISC-V model code translation function again to create a new block for the new settings. It will then automatically select between the blocks using the current value of the `pmKey` field in the processor structure.

Notes and Restrictions

1. The `bits` parameter must be 8 or 16.
2. If a simulated instruction modifies or could modify register `key`, *ensure that the current code block is terminated using `vmimtEndBlock`*. If this is not done, then subsequent simulated instructions in the code block may operate incorrectly if their behavior depends on the key state.

10.15 vmimtICount

Prototype

```
void vmimtICount(Uns32 bits, vmiReg rd);
```

Description

This function assigns the *nominal cycle count* for a processor to the given register. It is designed for use with timer functions (for example, `vmirtSetModelTimer`) and intercept libraries implementing timing tools.

The nominal cycle count is the sum of *executed instructions*, *halted* cycles and *skipped* cycles. *Halted* cycles are those for which the processor is not executing because it has been stopped by `vmirtHalt`. *Skipped* cycles are those that have been *explicitly* skipped (by a call to `vmirtAddSkipCount`) or *implicitly* skipped because the processor has been derated (see `vmirtSetDerateFactor`).

Example

The Imperas VAP tools include a tool for generation of basic block vector (BBV) files. This tool requires the least-significant 32 bits of the nominal cycle count to be recorded at the start of each block so that the cycles taken in the block can be calculated as simulation progresses. This is implemented in the tool as follows:

```
VMIOS_MORPH_FN(bbvMorph) {

    if(firstInBlock) {

        bbvDataP bbvData = object->bbvData;

        if (bbvData->started) {

            // record start-of-block ICount
            vmimtICount(32, bbvData->thisICountReg);

            // update BBV state
            vmimtArgNatAddress(object);
            vmimtCall((vmiCallFn)bbvStartBlock);

            // update previous block index
            vmimtMoveRC(32, bbvData->prevIndexReg, getBBVIndex(object, thisPC));
        }
    }

    // no callback function is required
    return 0;
}
```

Notes and Restrictions

1. The `bits` argument must be 8, 16, 32 or 64.

11 QuantumLeap Parallel Simulation Support

As of VMI version 6.0.0, Imperas Professional Simulation products implement a parallel simulation algorithm called *QuantumLeap*, which enables multicore platform simulation to be distributed over separate threads on multiple cores of the host machine for improved performance. Refer to the *OVP and CpuManager User Guide* for more information about QuantumLeap usage.

This section describes functions required to make processor models compatible with QuantumLeap.

11.1 vmimtAtomic

Prototype

```
void vmimtAtomic(void);
```

Description

This function indicates that the current instruction requires synchronization and that all other processor threads must be suspended while the instruction executes. `vmimtAtomic` should be used in three cases:

1. In a test-and-set instruction that reads, modifies and writes memory.
2. In the first instruction of a load/store exclusive or speculate/commit instruction pair.
3. When emitting code for any instruction which also emits an embedded call to a function which accesses shared data in an uncontrolled manner.

Example

The OVP ARM model uses this function in atomic loads:

```
static void emitAtomicLoadRR(
    armMorphStateP state,
    Uns32          regBits,
    Uns32          memBits,
    vmiReg         rd,
    vmiReg         ra
) {
    // indicate instruction is atomic
    armEmitAtomic();

    // do load in atomic context
    vmimtMoveRC(8, ARM_SI_TYPE, ASIT_ATOMIC);
    vmimtLoadRRO(regBits, memBits, 0, rd, ra, False, False);
    vmimtMoveRC(8, ARM_SI_TYPE, ASIT_NONE);
}
```

Notes and Restrictions

1. Refer to the *OVP Processor Modeling Guide* for a detailed explanation of when `vmimtAtomic` should be used.

12 Extension Library Support

An *extension library* is a special case of a binary intercept library that is used to add new instructions, registers, ports or other behavior to an existing processor model without requiring access to the source of that model. This provides a powerful method of modeling user-defined processor extensions.

This section gives information about functions that are specifically intended for use in extension libraries.

12.1 vmimtGetR

Prototype

```
void vmimtGetR(
    vmiProcessorP processor,
    Uns32         bits,
    vmiReg        rd,
    vmiRegInfoCP  ra
);
```

Description

Emit code to copy a value from processor source register `ra` to extension library target register `rd`. The source register is described using a `vmiRegInfoCP` structure; the target register is described using a `vmiReg` structure (usually created using function `vmimtGetExtReg` or `vmimtGetExtTemp`).

Argument `bits` specifies the register size; this must match the size specified in the `vmiRegInfoCP` structure.

If register `ra` does not have read access, a zero value is written to `rd`.

Example

This example shows how the function is used in an extension library implementing an exchange instruction for the OR1K processor (see the *OVP Processor Modeling Guide* for more information).

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

typedef struct vmiosObjectS {

    vmiRegInfoCP or1kRegs[OR1K_GPR_NUM];

    // new 32-bit registers implemented by this extension library
    Uns32 exchCount;
    Uns32 exchAddress;
    Uns32 exchRDValue;
    Uns32 exchWRValue;

    // 32-bit temporaries implemented by this extension library
    Uns32 exchTmp

} vmiosObject;

static void emitExchange(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         instruction
) {
    // get processor endianness for loads and stores
    memEndian     endian     = vmirtGetProcessorDataEndian(processor);
    memConstraint constraint = MEM_CONSTRAINT_ALIGNED;

    // extract instruction fields
    Uns32 ra = OPEX_A(instruction);
    Uns32 rb = OPEX_B(instruction);
    Int16 i  = OPEX_I(instruction);
```

```
    // create vmiReg objects addressing extension registers and temporaries
    // from processor context
    vmiReg exchCount   = vmimtGetExtReg (processor, &object->exchCount);
    vmiReg exchAddress = vmimtGetExtReg (processor, &object->exchAddress);
    vmiReg exchRDValue = vmimtGetExtReg (processor, &object->exchRDValue);
    vmiReg exchWRValue = vmimtGetExtReg (processor, &object->exchWRValue);
    vmiReg exchTmp     = vmimtGetExtTemp(processor, &object->exchTmp);

    // increment count of exchange instructions executed
    vmimtBinopRC(32, vmi_ADD, exchCount, 1, 0);

    // copy rb and ra processor GPRs to exchWRValue and exchAddress
    vmimtGetR(processor, 32, exchWRValue, object->or1kRegs[rb]);
    vmimtGetR(processor, 32, exchAddress, object->or1kRegs[ra]);

    // adjust address, including constant offset
    vmimtBinopRC(32, vmi_ADD, exchAddress, i, 0);

    // load exchTmp from exchAddress
    vmimtLoadRRO(32, 32, 0, exchTmp, exchAddress, endian, False, constraint);

    // store exchWRValue to exchAddress
    vmimtStoreRRO(32, 0, exchAddress, exchWRValue, endian, constraint);

    // copy exchTmp to exchRDValue
    vmimtMoveRR(32, exchRDValue, exchTmp);

    // copy exchTmp to processor GPR
    vmimtSetR(processor, 32, object->or1kRegs[rb], exchTmp);
}
```

Notes and Restrictions

None.

12.2 vmimtSetR

Prototype

```
void vmimtSetR(
    vmiProcessorP processor,
    Uns32         bits,
    vmiRegInfoCP  rd,
    vmiReg        ra
);
```

Description

Emit code to copy a value from extension library source register `ra` to processor target register `rd`. The target register is described using a `vmiRegInfoCP` structure; the source register is described using a `vmiReg` structure (usually created using function `vmimtGetExtReg` or `vmimtGetExtTemp`).

Argument `bits` specifies the register size; this must match the size specified in the `vmiRegInfoCP` structure.

If register `rd` does not have write access, it is not updated.

Example

This example shows how the function is used in an extension library implementing an exchange instruction for the OR1K processor (see the *OVP Processor Modeling Guide* for more information).

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

typedef struct vmiosObjectS {

    vmiRegInfoCP or1kRegs[OR1K_GPR_NUM];

    // new 32-bit registers implemented by this extension library
    Uns32 exchCount;
    Uns32 exchAddress;
    Uns32 exchRDValue;
    Uns32 exchWRValue;

    // 32-bit temporaries implemented by this extension library
    Uns32 exchTmp

} vmiosObject;

static void emitExchange(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         instruction
) {
    // get processor endianness for loads and stores
    memEndian     endian     = vmirtGetProcessorDataEndian(processor);
    memConstraint constraint = MEM_CONSTRAINT_ALIGNED;

    // extract instruction fields
    Uns32 ra = OPEX_A(instruction);
    Uns32 rb = OPEX_B(instruction);
    Int16 i  = OPEX_I(instruction);
```

```
    // create vmiReg objects addressing extension registers and temporaries
    // from processor context
    vmiReg exchCount   = vmimtGetExtReg (processor, &object->exchCount);
    vmiReg exchAddress = vmimtGetExtReg (processor, &object->exchAddress);
    vmiReg exchRDValue = vmimtGetExtReg (processor, &object->exchRDValue);
    vmiReg exchWRValue = vmimtGetExtReg (processor, &object->exchWRValue);
    vmiReg exchTmp     = vmimtGetExtTemp(processor, &object->exchTmp);

    // increment count of exchange instructions executed
    vmimtBinopRC(32, vmi_ADD, exchCount, 1, 0);

    // copy rb and ra processor GPRs to exchWRValue and exchAddress
    vmimtGetR(processor, 32, exchWRValue, object->or1kRegs[rb]);
    vmimtGetR(processor, 32, exchAddress, object->or1kRegs[ra]);

    // adjust address, including constant offset
    vmimtBinopRC(32, vmi_ADD, exchAddress, i, 0);

    // load exchTmp from exchAddress
    vmimtLoadRRO(32, 32, 0, exchTmp, exchAddress, endian, False, constraint);

    // store exchWRValue to exchAddress
    vmimtStoreRRO(32, 0, exchAddress, exchWRValue, endian, constraint);

    // copy exchTmp to exchRDValue
    vmimtMoveRR(32, exchRDValue, exchTmp);

    // copy exchTmp to processor GPR
    vmimtSetR(processor, 32, object->or1kRegs[rb], exchTmp);
}
```

Notes and Restrictions

None.

12.3 vmimtGetExtReg

Prototype

```
vmiReg vmimtGetExtReg(vmiProcessorP processor, void *pointer);
```

Description

Return a `vmiReg` descriptor that will access the data at `pointer` in the context of the given processor. The descriptor can be used as an argument to any of the morph-time API calls described in this document, enabling registers in extension libraries to be efficiently read and written.

Example

This example shows how the function is used in an extension library implementing an exchange instruction for the OR1K processor (see the *OVP Processor Modeling Guide* for more information).

```
#include "vmi/vmiMt.h"
#include "vmi/vmiTypes.h"

typedef struct vmiosObjectS {

    vmiRegInfoCP or1kRegs[OR1K_GPR_NUM];

    // new 32-bit registers implemented by this extension library
    Uns32 exchCount;
    Uns32 exchAddress;
    Uns32 exchRDValue;
    Uns32 exchWRValue;

    // 32-bit temporaries implemented by this extension library
    Uns32 exchTmp

} vmiosObject;

static void emitExchange(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         instruction
) {
    // get processor endianness for loads and stores
    memEndian     endian     = vmirtGetProcessorDataEndian(processor);
    memConstraint constraint = MEM_CONSTRAINT_ALIGNED;

    // extract instruction fields
    Uns32 ra = OPEX_A(instruction);
    Uns32 rb = OPEX_B(instruction);
    Int16 i  = OPEX_I(instruction);

    // create vmiReg objects addressing extension registers and temporaries
    // from processor context
    vmiReg exchCount   = vmimtGetExtReg (processor, &object->exchCount);
    vmiReg exchAddress = vmimtGetExtReg (processor, &object->exchAddress);
    vmiReg exchRDValue = vmimtGetExtReg (processor, &object->exchRDValue);
    vmiReg exchWRValue = vmimtGetExtReg (processor, &object->exchWRValue);
    vmiReg exchTmp     = vmimtGetExtTemp(processor, &object->exchTmp);

    // increment count of exchange instructions executed
    vmimtBinopRC(32, vmi_ADD, exchCount, 1, 0);

    // copy rb and ra processor GPRs to exchWRValue and exchAddress
    vmimtGetR(processor, 32, exchWRValue, object->or1kRegs[rb]);
```

```
    vmimtGetR(processor, 32, exchAddress, object->or1kRegs[ra]);

    // adjust address, including constant offset
    vmimtBinopRC(32, vmi_ADD, exchAddress, i, 0);

    // load exchTmp from exchAddress
    vmimtLoadRRO(32, 32, 0, exchTmp, exchAddress, endian, False, constraint);

    // store exchWRValue to exchAddress
    vmimtStoreRRO(32, 0, exchAddress, exchWRValue, endian, constraint);

    // copy exchTmp to exchRDValue
    vmimtMoveRR(32, exchRDValue, exchTmp);

    // copy exchTmp to processor GPR
    vmimtSetR(processor, 32, object->or1kRegs[rb], exchTmp);
}
```

Notes and Restrictions

None.

12.4 vmimtGetExtTemp

Prototype

```
vmiReg vmimtGetExtTemp(vmiProcessorP processor, void *pointer);
```

Description

Return a `vmiReg` descriptor that will access the data at `pointer` in the context of the given processor. The descriptor can be used as an argument to any of the morph-time API calls described in this document, enabling registers in extension libraries to be efficiently read and written. The `vmiReg` descriptor is for a *temporary* (used only for intermediate calculations).

Example

This example shows how the function is used in an extension library implementing an exchange instruction for the OR1K processor (see the *OVP Processor Modeling Guide* for more information).

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

typedef struct vmiosObjectS {

    vmiRegInfoCP or1kRegs[OR1K_GPR_NUM];

    // new 32-bit registers implemented by this extension library
    Uns32 exchCount;
    Uns32 exchAddress;
    Uns32 exchRDValue;
    Uns32 exchWRValue;

    // 32-bit temporaries implemented by this extension library
    Uns32 exchTmp

} vmiosObject;

static void emitExchange(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         instruction
) {
    // get processor endianness for loads and stores
    memEndian     endian     = vmirtGetProcessorDataEndian(processor);
    memConstraint constraint = MEM_CONSTRAINT_ALIGNED;

    // extract instruction fields
    Uns32 ra = OPEX_A(instruction);
    Uns32 rb = OPEX_B(instruction);
    Int16 i  = OPEX_I(instruction);

    // create vmiReg objects addressing extension registers and temporaries
    // from processor context
    vmiReg exchCount   = vmimtGetExtReg (processor, &object->exchCount);
    vmiReg exchAddress = vmimtGetExtReg (processor, &object->exchAddress);
    vmiReg exchRDValue = vmimtGetExtReg (processor, &object->exchRDValue);
    vmiReg exchWRValue = vmimtGetExtReg (processor, &object->exchWRValue);
    vmiReg exchTmp     = vmimtGetExtTemp(processor, &object->exchTmp);

    // increment count of exchange instructions executed
    vmimtBinopRC(32, vmi_ADD, exchCount, 1, 0);

    // copy rb and ra processor GPRs to exchWRValue and exchAddress
```

```
    vmimtGetR(processor, 32, exchWRValue, object->or1kRegs[rb]);
    vmimtGetR(processor, 32, exchAddress, object->or1kRegs[ra]);

    // adjust address, including constant offset
    vmimtBinopRC(32, vmi_ADD, exchAddress, i, 0);

    // load exchTmp from exchAddress
    vmimtLoadRRO(32, 32, 0, exchTmp, exchAddress, endian, False, constraint);

    // store exchWRValue to exchAddress
    vmimtStoreRRO(32, 0, exchAddress, exchWRValue, endian, constraint);

    // copy exchTmp to exchRDValue
    vmimtMoveRR(32, exchRDValue, exchTmp);

    // copy exchTmp to processor GPR
    vmimtSetR(processor, 32, object->or1kRegs[rb], exchTmp);
}
```

Notes and Restrictions

None.

13 Instruction Attributes Support

The Imperas Professional Simulation products implement an *Instruction Attributes API*, allowing introspection of the details of an executing instruction stream. Using this API, it is possible for tools, typically written using intercept library technology, to obtain information about each instruction as it executes, including:

1. The instruction *address*;
2. The instruction *size*, in bytes;
3. The instruction *disassembly*;
4. Any registers *read* or *written* by the instruction;
5. Any memory locations *read* or *written* by the instruction;
6. The instruction *class* (e.g. integer, floating point or branch);
7. Any *condition* associated with the instruction.

Most information visible through this API is automatically generated by the simulator. However, in some cases it is necessary for models to provide extra information or modify the automatically-generated information in some way. This section gives information on simulator functions designed for this purpose.

Note that instruction classes are specified by a bitfield of type `octiaInstructionClass`, defined in file `ocliaTypes.h` as follows:

```
typedef enum octiaInstructionClassE {
    OCL_IC_NONE        = 0x0,          ///< no class information
    OCL_IC_NOP         = 1ULL<<0,      ///< explicit NOP
    OCL_IC_INTEGER     = 1ULL<<1,      ///< instruction uses integer ALU
    OCL_IC_FLOAT       = 1ULL<<2,      ///< instruction uses FPU
    OCL_IC_DSP         = 1ULL<<3,      ///< instruction uses DSP
    OCL_IC_MULTIPLY    = 1ULL<<4,      ///< instruction implements multiply
    OCL_IC_DIVIDE      = 1ULL<<5,      ///< instruction implements divide
    OCL_IC_FMA         = 1ULL<<6,      ///< instruction implements
                                       ///  fused-multiply-add
    OCL_IC_SIMD        = 1ULL<<7,      ///< instruction implements SIMD operation
    OCL_IC_TRIG        = 1ULL<<8,      ///< instruction implements trigonometric
                                       ///  operation
    OCL_IC_LOG         = 1ULL<<9,      ///< instruction implements logarithmic
                                       ///  operation
    OCL_IC_RECIP       = 1ULL<<10,     ///< instruction implements reciprocal
                                       ///  operation
    OCL_IC_SQRT        = 1ULL<<11,     ///< instruction implements square root
                                       ///  operation
    OCL_IC_SYSREG      = 1ULL<<12,     ///< instruction accesses system register
                                       ///  state
    OCL_IC_IBARRIER    = 1ULL<<13,     ///< instruction barrier
    OCL_IC_DBARRIER    = 1ULL<<14,     ///< data barrier
    OCL_IC_ABARRIER    = 1ULL<<15,     ///< artifact barrier
    OCL_IC_ICACHE      = 1ULL<<16,     ///< instruction cache maintenance
    OCL_IC_DCACHE      = 1ULL<<17,     ///< data cache maintenance
    OCL_IC_MMU         = 1ULL<<18,     ///< memory management unit operation
    OCL_IC_ATOMIC      = 1ULL<<19,     ///< instruction implements atomic operation
    OCL_IC_EXCLUSIVE   = 1ULL<<20,     ///< instruction implements exclusive
                                       ///  operation
    OCL_IC_HINT        = 1ULL<<21,     ///< hint instruction
    OCL_IC_SYSTEM      = 1ULL<<22,     ///< system instruction
    OCL_IC_FCONVERT    = 1ULL<<23,     ///< instruction implements floating
                                       ///  point conversion
    OCL_IC_FCOMPARE    = 1ULL<<24,     ///< instruction implements floating
```

```
                                        ///  point comparison
    OCL_IC_BRANCH      = 1ULL<<25,      ///< instruction implements branch operation
    OCL_IC_BRANCH_DS   = 1ULL<<26,      ///< instruction implements branch operation
                                        ///  with delay slot
    OCL_IC_BRANCH_DSA  = 1ULL<<27,      ///< instruction implements branch operation
                                        ///  with annulled delay slot (if not taken)
    OCL_IC_OPAQUE_INT  = 1ULL<<28,      ///< instruction is subject to opaque
                                        ///  intercept
    OCL_IC_RESERVED1   = 1ULL<<29,      ///< start range for future class
                                        ///  extensions
    OCL_IC_RESERVEDN   = 1ULL<<47,      ///< end range for future class extensions
    OCL_IC_CUSTOM1     = 1ULL<<48,      ///< custom class 1
    OCL_IC_CUSTOM2     = 1ULL<<49,      ///< custom class 2
    OCL_IC_CUSTOM3     = 1ULL<<50,      ///< custom class 3
    OCL_IC_CUSTOM4     = 1ULL<<51,      ///< custom class 4
    OCL_IC_CUSTOM5     = 1ULL<<52,      ///< custom class 5
    OCL_IC_CUSTOM6     = 1ULL<<53,      ///< custom class 6
    OCL_IC_CUSTOM7     = 1ULL<<54,      ///< custom class 7
    OCL_IC_CUSTOM8     = 1ULL<<55,      ///< custom class 8
    OCL_IC_CUSTOM9     = 1ULL<<56,      ///< custom class 9
    OCL_IC_CUSTOM10    = 1ULL<<57,      ///< custom class 10
    OCL_IC_CUSTOM11    = 1ULL<<58,      ///< custom class 11
    OCL_IC_CUSTOM12    = 1ULL<<59,      ///< custom class 12
    OCL_IC_CUSTOM13    = 1ULL<<60,      ///< custom class 13
    OCL_IC_CUSTOM14    = 1ULL<<61,      ///< custom class 14
    OCL_IC_CUSTOM15    = 1ULL<<62,      ///< custom class 15
    OCL_IC_CUSTOM16    = 1ULL<<63       ///< custom class 16
} octiaInstructionClass;
```

13.1 vmimtRegNotReadR

Prototype

```
void vmimtRegNotReadR(Uns32 bits, vmiReg r);
```

Description

By default, information about the registers read and written by an instruction is automatically derived by examination of usage of `vmiReg` objects in each instruction, and cross-referencing this with the registers defined in the processor debug interface.

Sometimes, the automatic derivation incorrectly marks a register as read when it is not: a typical case is update of a system register, where only some bits are writable. In JIT-compiled code, this could be implemented by:

1. Reading the old register value;
2. Masking-in writable bits given in the new value;
3. Writing the new register value.

The sequence above suggests that the system register has been read *and* written by the instruction, whereas it was in fact only written (the read was a simulation artifact). Function `vmimtRegNotReadR` can be used to indicate that a register has not been written by any `vmiReg` references after the position of the function in the NMI node list.

Example

The OVP RISC-V model uses this function to indicate that masked writes as described above are not reads, as follows:

```
riscvArchitecture riscvEmitCSRWrite(
    riscvCSRId id,
    riscvP     riscv,
    vmiReg     rs,
    vmiReg     tmp
) {
    riscvArchitecture arch    = riscv->currentArch;
    csrAttrsCP        attrs   = &csrs[id];
    Uns32             bits    = riscvGetXlenMode(riscv);
    riscvCSRWriteFn   writeCB = getCSRWriteCB(id, riscv, bits);
    vmiReg            raw     = getRawArch(attrs, arch);
    Uns64             mask    = getCSRWriteMask(attrs, riscv);

    // indicate that this register has been written
    vmimtRegWriteImpl(attrs->name);

    if(writeCB) {

        // if CSR is implemented externally, mirror the result into any raw
        // register in the model (otherwise discard the result)
        if(!csrImplementExternalWrite(id, riscv)) {
            raw = VMI_NOREG;
        }

        // emit code to call the write function (NOTE: argument is always 64
        // bits, irrespective of the architecture size)
        vmimtArgUns32(id);
        vmimtArgProcessor();
```

```
        vmimtArgRegSimAddress(bits, rs);
        vmimtCallResult((vmiCallFn)writeCB, bits, raw);

        // terminate the current block if required
        if(attrs->wEndBlock) {
            vmimtEndBlock();
        }

    } else if(VMI_ISNOREG(raw)) {

        // emit warning for unimplemented CSR
        emitWarnUnimplementedCSR(id, riscv);

    } else if(mask==-1) {

        // new value is written unmasked
        vmimtMoveRR(bits, raw, rs);

    } else if(mask) {

        // apparent reads of register below are artifacts only
        vmimtRegNotReadR(bits, raw);

        // new value is written masked
        vmimtBinopRC(bits, vmi_ANDN, raw, mask, 0);
        vmimtBinopRRC(bits, vmi_AND, tmp, rs, mask, 0);
        vmimtBinopRR(bits, vmi_OR, raw, tmp, 0);
    }

    // return architectural constraints that apply to this register
    return attrs->arch;
}
```

Notes and Restrictions

None.

13.2 vmimtRegReadImpl

Prototype

```
void vmimtRegReadImpl(const char *name);
```

Description

By default, information about the registers read and written by an instruction is automatically derived by examination of usage of `vmiReg` objects in that instruction, and cross-referencing this with the registers defined in the processor debug interface.

Sometimes, the automatic derivation cannot determine that a register has been read. There are two common reasons for this:

1. The register defined in the debug interface (using a `vmiRegInfo` structure) does not specify a corresponding processor register (the `raw` field has value `VMI_NOREG`) because its value is instead implemented with a callback function.
2. The register is updated by an embedded call in the JIT-compiled code, instead of using VMI morph-time API primitives.

Function `vmimtRegReadImpl` can be used to indicate that a named processor register has been read, even if that is not apparent from the JIT-compiled code because of reasons specified above.

Example

The OVP RISC-V model uses this function to indicate that CSR registers are being read, as follows:

```
void riscvEmitCSRRead(riscvCSRId id, riscvP riscv, vmiReg rd, Bool isWrite) {

    riscvArchitecture arch   = riscv->currentArch;
    csrAttrsCP        attrs  = &csrs[id];
    Uns32             bits   = riscvGetXlenMode(riscv);
    riscvCSRReadFn    readCB = getCSRReadCB(id, riscv, bits, isWrite);
    vmiReg            raw    = getRawArch(attrs, arch);

    // indicate that this register has been read
    vmimtRegReadImpl(attrs->name);

    if(readCB) {

        // if CSR is implemented externally, mirror the result into any raw
        // register in the model (otherwise discard the result)
        if(!csrImplementExternalRead(id, riscv)) {
            raw = VMI_NOREG;
        }

        // emit code to call the write function
        vmimtArgUns32(id);
        vmimtArgProcessor();
        vmimtCallResult((vmiCallFn)readCB, bits, rd);
        vmimtMoveRR(bits, raw, rd);

    } else if(VMI_ISNOREG(raw)) {

        // emit warning for unimplemented CSR
```

```
        emitWarnUnimplementedCSR(id, riscv);
        vmimtMoveRC(bits, rd, 0);

    } else {

        // simple register read
        vmimtMoveRR(bits, rd, raw);
    }
}
```

Notes and Restrictions

None**.**

13.3 vmimtRegWriteImpl

Prototype

```
void vmimtRegWriteImpl(const char *name);
```

Description

By default, information about the registers read and written by an instruction is automatically derived by examination of usage of `vmiReg` objects in that instruction, and cross-referencing this with the registers defined in the processor debug interface.

Sometimes, the automatic derivation cannot determine that a register has been read. There are two common reasons for this:

1. The register defined in the debug interface (using a `vmiRegInfo` structure) does not specify a corresponding processor register (the `raw` field has value `VMI_NOREG`) because its value is instead implemented with a callback function.
2. The register is updated by an embedded call in the JIT-compiled code, instead of using VMI morph-time API primitives.

Function `vmimtRegWriteImpl` can be used to indicate that a named processor register has been written, even if that is not apparent from the JIT-compiled code because of reasons specified above.

Example

The OVP RISC-V model uses this function to indicate that CSR registers are being written, as follows:

```
riscvArchitecture riscvEmitCSRWrite(
    riscvCSRId id,
    riscvP     riscv,
    vmiReg     rs,
    vmiReg     tmp
) {
    riscvArchitecture arch    = riscv->currentArch;
    csrAttrsCP        attrs   = &csrs[id];
    Uns32             bits    = riscvGetXlenMode(riscv);
    riscvCSRWriteFn   writeCB = getCSRWriteCB(id, riscv, bits);
    vmiReg            raw     = getRawArch(attrs, arch);
    Uns64             mask    = getCSRWriteMask(attrs, riscv);

    // indicate that this register has been written
    vmimtRegWriteImpl(attrs->name);

    if(writeCB) {

        // if CSR is implemented externally, mirror the result into any raw
        // register in the model (otherwise discard the result)
        if(!csrImplementExternalWrite(id, riscv)) {
            raw = VMI_NOREG;
        }

        // emit code to call the write function (NOTE: argument is always 64
        // bits, irrespective of the architecture size)
        vmimtArgUns32(id);
        vmimtArgProcessor();
```

```
        vmimtArgRegSimAddress(bits, rs);
        vmimtCallResult((vmiCallFn)writeCB, bits, raw);

        // terminate the current block if required
        if(attrs->wEndBlock) {
            vmimtEndBlock();
        }

    } else if(VMI_ISNOREG(raw)) {

        // emit warning for unimplemented CSR
        emitWarnUnimplementedCSR(id, riscv);

    } else if(mask==-1) {

        // new value is written unmasked
        vmimtMoveRR(bits, raw, rs);

    } else if(mask) {

        // apparent reads of register below are artifacts only
        vmimtRegNotReadR(bits, raw);

        // new value is written masked
        vmimtBinopRC(bits, vmi_ANDN, raw, mask, 0);
        vmimtBinopRRC(bits, vmi_AND, tmp, rs, mask, 0);
        vmimtBinopRR(bits, vmi_OR, raw, tmp, 0);
    }

    // return architectural constraints that apply to this register
    return attrs->arch;
}
```

Notes and Restrictions

None**.**

13.4 vmimtInstructionClassAdd

Prototype

```
void vmimtInstructionClassAdd(octiaInstructionClass value);
```

Description

By default, information about the class of an instruction is automatically derived by examination of usage of `vmiReg` objects in that instruction. Sometimes, the automatic derivation cannot determine the class correctly. There are two common reasons for this:

1. The instruction implementation contains VMI primitives that are used for artifact purposes (for example, manufacturing an address using a multiply operation). In this case, the instruction class may contain unwanted extra information.
2. The instruction might be implemented by an embedded call. In this case, no information can be derived about the class of the instruction from morph-time primitives alone.

Function `vmimtInstructionClassAdd` can be used to add additional class information to the current instruction.

Example

The OVP RISC-V model uses this function in the main JIT callback to add extra information about each instruction, as follows:

```
VMI_MORPH_FN(riscvMorph) {

    riscvP riscv = (riscvP)processor;
    riscvMorphState state;

    // get instruction and instruction type
    riscvDecode(riscv, thisPC, &state.info);

    state.attrs      = &dispatchTable[state.info.type];
    state.riscv      = riscv;
    state.blockState = blockState;

    if(disableMorph(&state)) {

        // no action if in disassembly mode

    } else if(state.info.type==RV_IT_LAST) {

        // take Illegal Instruction exception
        emitIllegalInstruction();

    } else if(!instructionEnabled(riscv, &state)) {

        // instruction not enabled

    } else if(state.attrs->morph) {

        // translate the instruction
        vmimtInstructionClassAdd(state.attrs->iClass);
        state.attrs->morph(&state);

    } else {
```

```
        // here if no morph callback specified
        vmiMessage("F", CPU_PREFIX "_UIMP", // LCOV_EXCL_LINE
            SRCREF_FMT "unimplemented",
            SRCREF_ARGS(riscv, thisPC)
        );
    }
}
```

Notes and Restrictions

None.

13.5 vmimtInstructionClassSub

Prototype

```
void vmimtInstructionClassSub(octiaInstructionClass value);
```

Description

By default, information about the class of an instruction is automatically derived by examination of usage of `vmiReg` objects in that instruction. Sometimes, the automatic derivation cannot determine the class correctly. There are two common reasons for this:

1. The instruction implementation contains VMI primitives that are used for artifact purposes (for example, manufacturing an address using a multiply operation). In this case, the instruction class may contain unwanted extra information.
2. The instruction might be implemented by an embedded call. In this case, no information can be derived about the class of the instruction from morph-time primitives alone.

Function `vmimtInstructionClassAdd` can be used to add additional class information to the current instruction.

Example

The OVP RISC-V model uses this function in the function that generates an exclusive address for `AMO` operations. This function is automatically determined to be of class `OCL_IC_ATOMIC` (because it uses the `vmimtAtomic` primitive) but is in fact more usefully categorized as `OCL_IC_EXCLUSIVE`. `vmimtInstructionClassSub` is therefore used to *remove* `OCL_IC_ATOMIC` from the automatically-derived current instruction class:

```
static void startEA(riscvMorphStateP state, vmiReg ra) {

    // instruction must execute atomically but should not be classed as atomic
    // by instruction attributes (it is OCL_IC_EXCLUSIVE)
    vmimtAtomic();
    vmimtInstructionClassSub(OCL_IC_ATOMIC);

    // generate exclusive access tag for this address
    generateEATag(state, RISCV_EA_TAG, ra);
}
```

Notes and Restrictions

None**.**

13.6 vmimtSetInstructionCondition

Prototype

```
void vmimtSetInstructionCondition(Uns32 condition);
```

Description

Most processors implement *conditional instructions*, which have an effect only if a particular flag condition is satisfied. Usually such instructions are conditional branches, but some processors (for example, ARM variants) allow conditional execution of other instruction types as well.

Function `vmimtSetInstructionCondition` can be used to specify that the instruction that is currently being translated is conditional. An argument of 0 indicates the instruction is unconditional; other values specify a model-specific condition. The specified condition can be found later using function `ocliaGetInstructionCondition` from the Instruction Attributes API, and function `ocliaEvaluateInstructionCondition` can be used to evaluate the condition using the current processor state, returning a Boolean result. Function `vmirtEvaluateCondition` in the VMI Run Time Function API can also be used to evaluate a model-specific condition code. See the *VMI Run Time Function Reference* manual for more information about these functions.

Example

The OVP ARM model uses this function to indicate the condition for conditional instructions, inside a routine that returns a label used to skip the instruction action if the condition is `False`:

```
static vmiLabelP emitStartSkip(armMorphStateP state, armCondition cond) {

    armCond   entry  = armEmitPrepareCondition(state, cond, False);
    vmiLabelP doSkip = 0;

    if(entry.op!=ACO_ALWAYS) {
        doSkip = vmimtNewLabel();
        vmimtCondJumpLabel(entry.flag, entry.op==ACO_FALSE, doSkip);
        vmimtSetInstructionCondition(cond+1);  // convert to non-zero condition
    }

    return doSkip;
}
```

To support conditional evaluation by `ocliaEvaluateInstructionCondition` and `vmirtEvaluateCondition`, the processor must have a *condition evaluation callback* specified. The prototype for this is defined in file `vmiAttrs.h` as follows:

```
#define VMI_EVALUATE_CONDITION_FN(_NAME) Bool _NAME ( \
    vmiProcessorP processor,    \
    Uns32         condition     \
)
typedef VMI_EVALUATE_CONDITION_FN((*vmiEvaluateConditionFn));
```

For the ARM model, the condition evaluation callback is implemented like this:

```
VMI_EVALUATE_CONDITION_FN(armEvaluateConditionCB) {

    armP         arm    = (armP)processor;
    armCondition cond   = condition-1; // convert from non-zero condition
    Bool         Z      = arm->aflags.f[AFI_Z];
    Bool         N      = arm->aflags.f[AFI_N];
    Bool         C      = arm->aflags.f[AFI_C];
    Bool         V      = arm->aflags.f[AFI_V];
    Bool         result = False;

    switch(cond) {
        case ARM_C_EQ: result = Z;                 break;
        case ARM_C_NE: result = !Z;                break;
        case ARM_C_CS: result = C;                 break;
        case ARM_C_CC: result = !C;                break;
        case ARM_C_MI: result = N;                 break;
        case ARM_C_PL: result = !N;                break;
        case ARM_C_VS: result = V;                 break;
        case ARM_C_VC: result = !V;                break;
        case ARM_C_HI: result = (C && !Z);         break;
        case ARM_C_LS: result = !(C && !Z);        break;
        case ARM_C_GE: result = (N == V);          break;
        case ARM_C_LT: result = !(N == V);         break;
        case ARM_C_GT: result = (!Z && (N == V));  break;
        case ARM_C_LE: result = !(!Z && (N == V)); break;
        default:
            VMI_ABORT("unimplemented condition %u", cond);  // LCOV_EXCL_LINE
    }

    return result;
}
```

This function extracts the current value of the processor condition flags and uses these in combination with the `condition` argument to determine whether the condition is currently `True` or `False`. The condition evaluation callback is specified as the `evalConditionCB` argument in the instruction attributes structure:

```
const vmiIASAttr modelAttrs =  {

    . . . lines omitted for clarity . . .

    ////////////////////////////////////////////////////////////////////////
    // INSTRUCTION ATTRIBUTES SUPPORT
    ////////////////////////////////////////////////////////////////////////

    .evalConditionCB = armEvaluateConditionCB,
};
```

Notes and Restrictions

1. A condition value of 0 is special and means *unconditional*. If the condition recorded with an instruction is 0, calls to `ocliaEvaluateInstructionCondition` and `vmirtEvaluateCondition` will return `True` without calling the model-specific condition evaluation callback.

14 Timing Estimation

Functions in this section are designed to allow timing models to feed back delays into a simulation so that application performance can be estimated. They are typically used in intercept libraries.

14.1 vmimtAddSkipCountC

Prototype

```
void vmimtAddSkipCountC(Uns64 skipCount);
```

Description

Emit code to add `skipCount` instructions to the pending skipped instruction count for this processor. The accumulated skipped cycles will typically be committed at the start of the next quantum. See the *VMI Run Time Function Reference* manual for more detailed information.

Example

This example shows how this function could be used in a performance estimation library that adds cycles for memory delays. In this example, one cycle is added for each read access, and two cycles for each write access.

```
static VMIOS_MORPH_FN(morphCallback) {

    // get instruction attributes
    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, OCL_DS_ADDRESS, False);

    if(attrs) {

        Uns32           delay = 0;
        octiaMemAccessP ma;

        // calculate extra delay for this instruction based on loads and stores
        for(
            ma = ocliaGetFirstMemAccess(attrs);
            ma;
            ma = ocliaGetNextMemAccess(ma)
        ) {
            switch(ocliaGetMemAccessType(ma)) {
                case OCL_MAT_LOAD:
                    delay += 1;
                    break;
                case OCL_MAT_STORE:
                    delay += 2;
                    break;
                default:
                    break;
            }
        }

        // annotate extra delay
        if(delay) {
            vmimtAddSkipCountC(delay);
        }

        // free attributes
        ocliaFreeAttrs(attrs);
    }

    // indicate that normal instruction translation should be done
    return 0;
}
```

Notes and Restrictions

None**.**

14.2 vmimtAddSkipCountR

Prototype

```
void vmimtAddSkipCountR(Uns32 bits, vmiReg skipCount);
```

Description

Emit code to add the 64-bit `skipCount` register to the pending skipped instruction count for this processor. The accumulated skipped cycles will typically be committed at the start of the next quantum. See the *VMI Run Time Function Reference* manual for more detailed information.

Example

This function is not currently used in any public OVP models.

Notes and Restrictions

1. The `bits` argument must be 64.