OVP VMI Run Time Function Reference

Imperas Software Limited

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

Author:	Imperas Software Limited
Version:	7.70.0
Filename:	OVP_VMI_Run_Time_Function_Reference
Project:	OVP VMI Run Time Function Reference
Last Saved:	Wednesday, 17 July 2024
Keywords:	

Table of Contents

1 Introduction

This is reference documentation for **version 7.70.0** of the VMI *run time* function interface, defined in `ImpPublic/include/host/vmi/vmiRt.h`.

The functions in this interface are generally used within *embedded calls*[1] from translated native code implementing a processor model, generated by the VMI *morph time* interface. See the *VMI Morph Time Function Reference Guide* for more information about the VMI morph time interface and how to use it to create embedded calls.

Functions in the run time interface have the prefix `vmirt`.

[1] An *embedded call* is a function that is called from a JIT-translated code block to implement behavior that cannot easily be described using morpher primitives.

2 QuantumLeap Semantics

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.

When QuantumLeap is active, some functions require the current thread to be suspended until all other concurrent threads have been stopped so that they may safely execute. There are three categories of function with different semantics:

Thread Safe
Thread safe functions never cause the current thread to be suspended.

Synchronizing
Synchronizing functions always cause the current thread to be suspended until all other threads have been safely stopped.

Non-self-synchronizing
Non-self-synchronizing functions are passed a processor as an argument. They cause the current thread to be suspended only if the processor is not the current processor.

When creating a processor model for use with QuantumLeap, be careful to avoid situations where a processor model relies on a cached value of a shared data item that could be updated by another processor in an embedded call, of the form:

```
void vmic_Callback(void) {
    tmp = <shared_value>;
    <vmi synchronizing call>;
    fn(tmp);
}
```

The above pseudo-example is potentially dangerous when QuantumLeap is active if `<shared_value>` can be modified by another processor while the current thread is suspended. To avoid problems like this, try to restructure routines so that the synchronizing call is made before any shared variable access:

```
void vmic_Callback(void) {
    <vmi synchronizing call>;
    tmp = <shared_value>;
    fn(tmp);
}
```

If it is not possible to modify the embedded call in this manner, use function `vmimtAtomic` when emitting the embedded call to ensure that the call is made with all other threads suspended. See the *VMI Morph Time Function Reference Guide* for more information about this function.

3 Model Configuration

A processor model can have optional features that can be configured by the platform during construction. Configuration is controlled by parameters which form part of the model's interface.

Some processor models use a more complex two-stage configuration process; an initial set of parameters are defined by the model. These are configured by the simulator and their values passed to a pre-constructor function in the model. The model uses these values to create the main set of parameters which depend on the values of the initial parameters. For instance, setting a boolean parameter FP_ENABLE which adds a floating point unit could cause another parameter FP_PRECISION to appear that specifies floating point precision - 32-bit or 64-bit. Two-stage parameterization is described in section 3.3.

3.1 Model Parameters

Parameters are specified to the simulator by an iterator function and a size function specified in the model's attributes table. A parameter specification specifies the data type and bounding conditions of the parameter so the model does not need to check for trivial errors. The model must define a structure which contains value fields for each parameter. It should use the provided macros `VMI_<type>_PARAM`, which reserves space for the value itself and also for a boolean which is true if the parameter has been set by the platform, false otherwise.

The supported parameter types are described below:

macro	**data type**
`VMI_BOOL_PARAM`	boolean
`VMI_INT32_PARAM`	32 bit signed
`VMI_UNS32_PARAM`	32 bit unsigned
`VMI_UNS64_PARAM`	64 bit unsigned
`VMI_DBL_PARAM`	floating point
`VMI_STRING_PARAM`	0 terminated string
`VMI_ENUM_PARAM`	0 terminated string
`VMI_ENDIAN_PARAM`	0 terminated string
`VMI_PTR_PARAM`	native host pointer

During initialization, the simulator uses the iterator function to get the list of parameters for the model. Then it allocates the model's parameter structure (using the size function) and fills in the correct values. This structure is then passed as the `params` parameter to the model's constructor, where the model can use the values. Within the constructor the value of parameter `X` can be found using an expression of the form:

```
params->X
```

To determine whether the parameter has been explicitly set, use an expression of the form:

```
params->SETBIT(X)
```

Note that the `params` structure is valid only for use within the constructor and deleted thereafter.

3.2 Parameter Specification

The parameter specification structure is defined in `vmiParameters.h` and should be initialized using these macros:

macro	data type	limits
VMI_BOOL_PARAM_SPEC	boolean	0 or 1
VMI_INT32_PARAM_SPEC	32 bit signed	specified min / max
VMI_UNS32_PARAM_SPEC	32 bit unsigned	specified min / max
VMI_INT64_PARAM_SPEC	64 bit signed	specified min / max
VMI_UNS64_PARAM_SPEC	64 bit unsigned	specified min / max
VMI_DBL_PARAM_SPEC	floating point	specified min / max
VMI_STRING_PARAM_SPEC	0 terminated string	any string (or 0 if not specified)
VMI_ENUM_PARAM_SPEC	0 terminated string	string must be a member of the specified list
VMI_ENDIAN_PARAM_SPEC	0 terminated string	"big" or "little"
VMI_PTR_PARAM_SPEC	native host pointer	none

The iterator function must be supplied by the model and should use the provided macro from `vmiAttrs.h`:

Prototype

```
#define VMI_PROC_PARAM_SPECS_FN(_NAME) vmiParameterP _NAME ( \
    vmiProcessorP processor,    \
    vmiParameterP prev          \
)
```

It should return the first or subsequent parameter specification or 0 if at the end of the list. Note that the iterator is also supplied with the processor pointer, so can include or exclude parameters according to the current configuration.

Example

```
// VMI header files
#include "vmi/vmiAttrs.h"
#include "vmi/vmParameters.h"

//
// Define the parameter structure
//
typedef struct paramValuesS {

    VMI_ENUM_PARAM(variant);           // enumeration specifying the variant
    VMI_ENDIAN_PARAM(endian);          // endian parameter
    VMI_BOOL_PARAM(verbose);           // boolean parameter
    VMI_UNS32_PARAM(nInts);          // 32bit unsigned parameter
    VMI_UNS64_PARAM(rstVec);         // 64bit parameter used as an address
    VMI_STRING_PARAM(dataFile);      // string param used as a file name
    VMI_PTR_PARAM(data);             // ptr to platform provided data struct

} pVals, *pValsP;

//
// Define the list of legal variants (NULL terminated)
//
vmiEnumParameter variantList[] = {
    // name        value   description
    { "variant1",  1,  "first  variant" },
    { "variant2",  2,  "second variant" },
    { "variant3",  3,  "third  variant" },
    { 0 },
};

//
// Define the parameters
//
static vmiParameter formals[] = {
    VMI_ENUM_PARAM_SPEC(  pVals, variant,  variantList, "proc variant"),
    VMI_ENDIAN_PARAM_SPEC(pVals, endian,                "proc endianness"),
    VMI_BOOL_PARAM_SPEC(  pVals, verbose,  0,           "Enable text output"),
    VMI_UNS32_PARAM_SPEC( pVals, nInts,    1, 0, 255,   "Num of int ports"),
    VMI_UNS64_PARAM_SPEC( pVals, rstVec,  0, 0, VMI_MAXU64, "Reset vec addr"),
    VMI_STRING_PARAM_SPEC(pVals, dataFile, "",         "dataFile file name "),
    VMI_PTR_PARAM_SPEC(   pVals, data,     0,          "Addr of data struct"),


    // Add entry with name==NULL to terminate list
    VMI_END_PARAM
};

//
// Function to iterate the parameter specs
//
VMI_PROC_PARAM_SPECS_FN(getParamSpec) {
    if(!prev) {
        return formals;
    } else {
        prev++;
        if (prev->name)
            return prev;
        else
            return 0;
    }
}

//
// Get the size of the parameter values table
//
VMI_PROC_PARAM_TABLE_SIZE_FN(paramValueSize) {
    return sizeof(pVals);
}

//
// model constructor
//
VMI_CONSTRUCTOR_FN(modelConstructor) {
    myProcP myProc = (processorP)processor;
    pValsP params  = (pValsP)parameterValues; // cast to my type

    ...
    // use the parameter values
    if (params->SETBIT(rstVec)) {
        ... params->rstVec ....
    }

    if (params->verbose) {
       vmiPrintf(....);
    }
    ...
}

//
// Add functions to the model attributes table
//
const vmiIASAttr modelAttrs = {
    ...
    .constructorCB    = modelConstructor,
    .paramSpecsCB     = getParamSpec,
    .paramValueSizeCB = paramValueSize,
    ...
```

```
};
```

Restrictions
The parameter structure exists only for the life of the constructor function.

3.3 Parameter groups

A complex model can have more parameters than can be easily managed in a single list. To make parameters easier to find, they can be put into groups. A parameter group is defined by creating a `vmiParameterGroup` structure. To put a parameter in that group the `group` pointer in the `vmiParameter` structure is set to that group.

A `vmiParameterGroup` structure should be defined using the `VMI_PARAM_GROUP_SPEC` macro.

Parameters in a group should be specified using the `VMI_xxx_GROUP_PARAM_SPEC` macros which are similar to the `VMI_xxx_PARAM_SPEC` macros described in section 3.2.

macro	**data type**	**limits**
`VMI_BOOL_GROUP_PARAM_SPEC`	boolean	0 or 1
`VMI_INT32_GROUP_PARAM_SPEC`	32 bit signed	specified min / max
`VMI_UNS32_GROUP_PARAM_SPEC`	32 bit unsigned	specified min / max
`VMI_INT64_GROUP_PARAM_SPEC`	64 bit signed	specified min / max
`VMI_UNS64_GROUP_PARAM_SPEC`	64 bit unsigned	specified min / max
`VMI_DBL_GROUP_PARAM_SPEC`	floating point	specified min / max
`VMI_STRING_GROUP_PARAM_SPEC`	0 terminated string	any string (or 0 if not specified)
`VMI_ENUM_GROUP_PARAM_SPEC`	0 terminated string	string must be a member of the specified list
`VMI_ENDIAN_GROUP_PARAM_SPEC`	0 terminated string	"big" or "little"
`VMI_PTR_GROUP_PARAM_SPEC`	native host pointer	none

Grouped parameters behave exactly as un-grouped parameters but are displayed in their groups by the simulator command `-showoverrides` and in automatically generated model documentation.

Example Prototype

```
#define VMI_STRING_GROUP_PARAM_SPEC (_STR, _NAME, _DEFAULT, _GROUP, _DESC)
```

Example

```
// VMI header files
#include "vmi/vmiAttrs.h"
#include "vmi/vmParameters.h"

//
// Define the parameter structure
//
typedef struct paramValuesS {

    VMI_ENDIAN_PARAM(endian);         // endian parameter
    VMI_BOOL_PARAM(verbose);          // boolean parameter
    VMI_UNS32_PARAM(nInts);           // 32bit unsigned parameter
    VMI_UNS64_PARAM(rstVec);          // 64bit parameter used as an address
    VMI_STRING_PARAM(dataFile);       // string param used as a file name
    VMI_PTR_PARAM(data);              // ptr to platform provided data struct

} pVals, *pValsP;

//
// Define the groups
```

```
//
VMI_PARAM_GROUP_SPEC(text)
VMI_PARAM_GROUP_SPEC(data)

//
// Define the grouped parameters
//
static vmiParameter formals[] = {
    VMI_BOOL_GROUP_PARAM_SPEC  (pVals, debug,                text, "Debug output"),
    VMI_BOOL_GROUP_PARAM_SPEC  (pVals, verbose,  0,          text, "Extra info"),
    VMI_UNS32_GROUP_PARAM_SPEC (pVals, nInts,    1, 0, 255, data, "Num of ports"),
    VMI_STRING_GROUP_PARAM_SPEC(pVals, dataFile, "",         data, "File name "),
    VMI_PTR_GROUP_PARAM_SPEC   (pVals, data,     0,          data, "Addr of data"),
    VMI_END_PARAM
};
```

Restrictions

Each group must have a unique name conforming to C variable syntax.

3.4 Two-stage parameterization

Two-stage parameterization requires two additional entries in the `vmiIASAttrs` structure:

`preParamSpecsCB` specifies the iterator that returns the list of parameters to be evaluated before the main parameter list is decided. The `paramSpecsCB` iterator returns the remainder of the parameters and must not include the pre-parameters.

`preParamValuesCB` specifies a function called to receive the values of the pre-parameters.
This must make decisions based on the pre-parameter values so that the main parameter iterator `paramSpecsCB` can return the modified list of parameters.

The parameter values array holds the values of all parameters, pre- and main.
The iterator function `paramSpecsCB` is also passed the parameter values array. It can use any of the pre-parameter values since they will be valid, though the main parameters will not.

Example

```
// VMI header files
#include "vmi/vmiAttrs.h"
#include "vmi/vmiParameters.h"

//
// Define the parameter structure
//
typedef struct paramValuesS {

    VMI_ENUM_PARAM(variant);       // enumeration specifying the variant
    VMI_BOOL_PARAM(extend);        // if true, more parameters exist
    VMI_UNS64_PARAM(ext1);         // 64bit conditional parameter
    VMI_STRING_PARAM(ext2);        // conditional string param

} pVals, *pValsP;

//
// Define the list of legal variants (NULL terminated)
//
vmiEnumParameter variantList[] = {
    // name        value   description
    { "variant1",  1,  "first  variant" },
    { "variant2",  2,  "second variant" },
    { 0 },
};

//
// Define the parameters
//
static vmiParameter preFormals[] = {
    VMI_ENUM_PARAM_SPEC  (pVals, variant,  variantList, "processor variant"),
    VMI_BOOL_PARAM_SPEC  (pVals, extend,   0,             "is extended"),
    VMI_END_PARAM
};

static vmiParameter formals[] = {
    VMI_UNS64_PARAM_SPEC (pVals, ext1,  0,0,1000,       "extension 1"),
    VMI_STRING_PARAM_SPEC(pVals, ext2, 0,               "extension 2"),
    VMI_END_PARAM
};
```

```
//
// Function to iterate the pre-parameter specs
//
VMI_PROC_PARAM_SPECS_FN(getPreParamSpec) {
    if(!prev) {
        return preFormals;
    } else {
        prev++;
        if (prev->name)
            return prev;
        else
            return 0;
    }
}

//
// Function to iterate the main parameter specs
//
VMI_PROC_PARAM_SPECS_FN(getParamSpec) {
    pValsP params = (pValsP)parameterValues;

    // read the early parameter
    if(params->extend) {
        if(!prev) {
            return formals;
        } else {
            prev++;
            if (prev->name)
                return prev;
            else
                return 0;
        }
    } else {
        return 0;
    }
}

//
// Get the size of the parameter values table
//
VMI_PROC_PARAM_TABLE_SIZE_FN(paramValueSize) {
    return sizeof(pVals);
}

VMI_SET_PARAM_VALUES_FN(setParamValues) {
}

//
// model constructor
//
VMI_CONSTRUCTOR_FN(modelConstructor) {
    pValsP params = (pValsP)parameterValues; // cast to my type

    ...
    // use the parameter values
    if (params->extend) {

        if (params->ext1) {
            ...
        }
        ...
    }
    ...
}

// the model attributes table

const vmiIASAttr modelAttrs = {
    ...
    .constructorCB    = modelConstructor,
    .preParamSpecsCB  = getPreParamSpec,
    .paramSpecsCB     = getParamSpec,
    .paramValueSizeCB = paramValueSize,
    .preParamValuesCB = setParamValues,
};
```

Diagnostics

When the model is instantiated in a platform, use `-showmodeloverrides` to show which parameters exist in the current configuration. In this example, setting the parameter `extend` will cause the creation of parameters `ext1` and `ext2`.

4 Model Hardware Interface

An OVP model must provide to the simulator a specification of its hardware interface, so that the simulator can check the connections to the platform as they are made, and so that other tools can use the OP or legacy ICM API to interrogate a model and discover its interface without referring to the model's source code. The hardware interface specification separately covers Bus Ports, Net Ports and Connection Ports (also known as FIFO Ports).

4.1 Bus Port Specification

A bus port connects to a bus. A bus transmits read or write requests from a bus master to a bus slave (OVP does not represent details of bus timing, contention, or individual signals that comprise a real bus implementation). The model must provide an iterator function which returns the first or subsequent bus port specifications, or 0 at the end of the list. The function must be registered in the model attributes table and should be defined using this macro from `vmiPorts.h`:

Prototype

```
#define VMI_BUS_PORT_SPECS_FN(_NAME) vmiBusPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiBusPortP   prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can include or exclude bus ports according to the current configuration.

The bus port specification is defined in `vmiPorts.h`.

Definition

```
typedef enum vmiBusPortTypeE {
    vmi_BP_MASTER,
    vmi_BP_SLAVE
} vmiBusPortType;

typedef enum vmiDomainTypeE {
    vmi_DOM_CODE,     // code domain port
    vmi_DOM_DATA,     // data domain port
    vmi_DOM_OTHER     // other domain port
} vmiDomainType;

typedef struct vmiBusPortS {

    const char    *name;
    vmiBusPortType type;
    vmiDomainType  domainType;

    // The number of bits in the address bus if this is a master port.
    // min and max are the minimum and maximum number of bits that
    // the model will support.
    // unset is the number of bits the port will have if not set
    // by another source.
    struct         {Uns8 min; Uns8 max; Uns8 unset;} addrBits;

    // if true, leaving this port unconnected will raise an error
    Bool           mustBeConnected;

    // port description
    const char    *description;

    // domain is non-NULL if port is connected (set by simulator when platform
    // is created)
    memDomainP     domain;

    // address range of connected slave port (set by simulator when platform
    // is created)
    Addr           addrLo;
    Addr           addrHi;

    // maximum decode address (slave port only)
    Addr           maxAddress;
} vmiBusPort;
```

Ports are either of *master* or *slave* type, as indicated by the `type` field. For processor models, ports are almost always masters (i.e. `type` is `vmi_BP_MASTER`). A simple processor model will typically have only two bus master ports, of domain type `vmi_DOM_CODE` and `vmi_DOM_DATA`, named `INSTRUCTION` and `DATA`. Other ports may be present for more complex processor models.

Every port requires these fields to be set:

1. `type` (`vmi_BP_MASTER` or `vmi_BP_SLAVE`);
2. `domainType` (`vmi_DOM_CODE`, `vmi_DOM_DATA` or `vmi_DOM_OTHER`);
3. `mustBeConnected` (whether the port must be connected or can be left unconnected);
4. `description` (port description, or `NULL` for no description).

Master ports require the port *address bits* to be specified, so that the simulator can validate that the port is connected to a suitable bus. This is specified in the `addrBits` field, using subfields `min` and `max` (to specify minimum and maximum address bits). The `unset` field specifies the number of bits to use if the address with is not set as part of platform construction (for example, if the port is not connected).

Slave ports require the `maxAddress` field to be set to specify the width of the address range that is exposed by the slave to external masters. The port width is defined to be `maxAddress+1` bytes.

When the simulator connects to a port, it writes to other fields in this structure to complete the connection. For all port types, the `domain` field is updated with the `memDomainP` pointer implementing the external bus. For a *master* port, the processor can read and write to this domain using functions such as `vmirtReadNByteDomain` and `vmirtWriteNByteDomain`, described later in this manual. For *slave* ports, the `addrLo` and `addrHi` fields are also filled by the simulator, to specify the memory range on the connected domain that should be implemented by the slave; typically, the processor model will alias that region of the external domain to a locally-defined domain object that satisfies externally-generated bus requests.

Because the `domain`, `addrLo` and `addrHi` fields are written by the simulator when the port is connected, processor models must implement a separate copy of the bus port list for *each instance* of a processor. This typically means that the list must be allocated and a pointer to it held on the processor structure. The following example shows how this is done for the standard OR1K processor model.

Example

```
#include "vmi/vmiPorts.h"

//
// Return number of members of an array
//
#define NUM_MEMBERS(_A) (sizeof(_A)/sizeof((_A)[0]))
```

```
//
// Template bus port list
//
const static vmiBusPort busPorts[] = {
    {"INSTRUCTION", vmi_BP_MASTER, vmi_DOM_CODE, {32,32}, 1},
    {"DATA"       , vmi_BP_MASTER, vmi_DOM_DATA, {32,32}, 0},
};

//
// Allocate bus port specifications
//
static void newBusPorts(or1kP or1k) {

    Uns32 i;

    or1k->busPorts = STYPE_CALLOC_N(vmiBusPort, NUM_MEMBERS(busPorts));

    for(i=0; i<NUM_MEMBERS(busPorts); i++) {
        or1k->busPorts[i] = busPorts[i];
    }
}

//
// Free bus port specifications
//
static void freeBusPorts(or1kP or1k) {

    if(or1k->busPorts) {
        STYPE_FREE(or1k->busPorts);
        or1k->busPorts = 0;
    }
}

//
// Get the next bus port
//
VMI_BUS_PORT_SPECS_FN(or1kGetBusPortSpec) {

    or1kP or1k = (or1kP)processor;

    if(!prev) {

        // first port
        return or1k->busPorts;

    } else {

        // port other than the first
        Uns32 prevIndex = (prev-or1k->busPorts);
        Uns32 thisIndex = prevIndex+1;

        return (thisIndex<NUM_MEMBERS(busPorts)) ? &or1k->busPorts[thisIndex] : 0;
    }
}

//
// OR1K processor constructor
//
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;

    . . .

    // create bus port specifications
    newBusPorts(or1k);
}

//
// OR1K processor destructor
```

```
//
VMI_DESTRUCTOR_FN(or1kDestructor) {

    or1kP or1k = (or1kP)processor;

    . . .

    // free bus port specifications
    freeBusPorts(or1k);
}

//
// Register the function in the model attribute table
//
const vmiIASAttr modelAttrs = {
    ...
    .busPortSpecsCB = or1kGetBusPortSpec,
    ...
};
```

4.2 vmirtGetBusPortDomain

Prototype

```
memDomainP vmirtGetBusPortDomain(
    vmiProcessorP processor,
    vmiBusPortP   port
);
```

Description

This function is deprecated. Use `port->domain` to obtain the memory domain object connected to a port.

QuantumLeap Semantics

Thread Safe

4.3 Net Port Specification

A net port connects to a net. A net usually carries a Boolean value, but can be used to carry a 64-bit value. The model must provide an iterator function which returns the first or subsequent net port specifications, or 0 at the end of the list. The function must be registered in the model attributes table and should use this macro from `vmiPorts.h`:

Prototype

```
#define VMI_NET_PORT_SPECS_FN(_NAME) vmiNetPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiNetPortP   prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can use data associated with the instance.

Each specification includes:

- Net port name.
- Net port type (*input*, *output* or *in-out*).
- A callback function and user-data field (for an input).
- Pointer to a handle (used for output).
- Optional description.
- A Boolean indicating whether the port must be connected or may be left unconnected.

The net port specification object is defined in `vmiPorts.h`.

Definition

```
typedef struct vmiNetPortS {

    char          *name;
    vmiNetPortType type;
    void          *userData;

    // callback for input or I/O net change
    vmiNetChangeFn netChangeCB;
    Uns32         *handle;

    // space for documentation
    const char    *description;
    void          *descriptionDom;

    Bool           mustBeConnected;

} vmiNetPort;
```

A processor model will typically use net ports for reset, external interrupt inputs and for integration with an external interrupt controller. Note that if the net port is an output, the output handle must be allocated per port, per processor instance and that the handle pointer in the `vmiNetPort` structure must be set to point to this handle. The simulator will use this pointer to set the handle before simulation begins. The model then uses the handle to update the output net port when required.

Example

This example is a simplified extract from the OVP ARC model.

```
#include "vmi/vmiPorts.h"

//
// Define (part of) the model instance data structure
//
typedef struct arcS {
    . . .
    Uns32          watchdogHandle;          // watchdog timer output handle
    . . .
} arc;

//
// Called by the simulator when an external reset is raised
//
static VMI_NET_CHANGE_FN(externalReset) {

    arcP arc       = (arcP)processor;
    Bool resetHigh = newValue&1;

    if(resetHigh==arc->resetHigh) {

        // no change in reset signal

    } else if(newValue) {

        // halt the processor while reset goes high
        arcHaltProcessor(arc, AD_RESET);

    } else {

        // indicate reset exception is pending
        arc->resetPending = True;

        // restart the processor when reset goes low
        arcRestartProcessor(arc, AD_RESET|AD_LOCKUP);

        // reset the processor
        arcReset(arc);

        // handle pending reset exception
        doInterrupt(arc);
    }

    // save new value of reset net
    arc->resetHigh = resetHigh;
}

//
// Called when watchdog timer expires
//
static void writeWatchdog(arcP arc, Uns32 newValue) {

    Uns32 watchdogHandle = arc->watchdogHandle;

    if(watchdogHandle) {
        vmirtWriteNetPort((vmiProcessorP)arc, watchdogHandle, newValue);
    }
}

//
// Create port specifications (called from the constructor)
//
void arcNewPortSpecs(arcP arc) {

    // declare template port structure for fixed ports
    vmiNetPort template[] = {
        {
            name        : "reset",
            type        : vmi_NP_INPUT,
            netChangeCB : externalReset,
            description : "Processor reset"
        },
        {
            name        : "watchdog",
            type        : vmi_NP_OUTPUT,
            handle      : &arc->watchdogHandle,
            description : "Watchdog timer"
        },
    };

    // allocate permanent port structure (including null terminator)
    Uns32       numPorts = NUM_MEMBERS(template);
    vmiNetPortP result   = STYPE_CALLOC_N(vmiNetPort, numPorts+1);

    // fill members
    for(i=0; i<numPorts; i++) {
        result[i] = template[i];
    }

    // save allocated ports on processor
    arc->ports = result;
}

//
// Get the next net port
//
VMI_NET_PORT_SPECS_FN(arcGetNetPortSpec) {

    arcP        arc = (arcP)processor;
    vmiNetPortP this;

    if(!prev) {
        this = arc->netPorts;
    } else {
        this = prev + 1;
    }

    return this->name ? this : 0;
}

//
// Register the function in the model attribute table
//
const vmiIASAttr modelAttrs = {
    ...
    .netPortSpecsCB     = arcGetNetPortSpec,
    ...
};
```

4.4 Connection Port Specification

If a model uses connection ports, it must define an iterator function that gives access to a list of connection port specifications. The iterator function is specified in the model attributes table. It returns the first or subsequent connection port specification structures, or 0 at the end of the list. The function must be registered in the model attributes table and should use this macro from `vmiPorts.h`:

Prototype

```
#define VMI_FIFO_PORT_SPECS_FN(_NAME) vmiFifoPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiFifoPortP  prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can include or exclude bus ports according to the current configuration.

Each specification includes:

- the port name;
- the port type;
- the data width in bits;
- a pointer to the port handle in the model's main data structure (the model writer is responsible for allocating a handle for each connection); and
- an optional description.

The specification structure is defined in `vmiPorts.h`.

Definition

```
typedef struct vmiFifoPortS {

    const char      *name;
    vmiFifoPortType  type;
    Uns32            bits;
    void           **handle;
    void            *userData;

    // space for documentation
    const char      *description;
    void            *descriptionDom;

} vmiFifoPort;
```

During initialization, the simulator uses the iterator function to get the list of ports and set the port handles. The model can then use the following functions to send data through the connections.

This shows how the function might be defined and installed, and how the handles are created and referenced.

Example

This example is a simplified extract from the OVP OR1K model.

```
//
// Define (part of) the model instance data structure
//
typedef struct or1kS {
    . . .
    vmiFifoPortP   fifoPorts;       // fifo port descriptions
    vmiConnInputP  inputConn;       // input FIFO connection
    vmiConnOutputP outputConn;      // output FIFO connection
    . . .
} or1k, *or1kP;

//
// Template FIFO port list
//
static vmiFifoPort fifoPorts[] = {
    {"fifoIn" , vmi_FIFO_INPUT,  32, (void *)VMI_CPU_OFFSET(or1kP, inputConn) },
    {"fifoOut", vmi_FIFO_OUTPUT, 32, (void *)VMI_CPU_OFFSET(or1kP, outputConn)}
};

//
// Allocate FIFO port structure
//
static void newFifoPorts(or1kP or1k) {

    Uns32 i;

    or1k->fifoPorts = STYPE_CALLOC_N(vmiFifoPort, NUM_MEMBERS(busPorts));

    for(i=0; i<NUM_MEMBERS(fifoPorts); i++) {

        or1k->fifoPorts[i] = fifoPorts[i];

        // correct FIFO port handles
        Uns8 *raw = (Uns8*)(or1k->fifoPorts[i].handle);
        or1k->fifoPorts[i].handle = (void **)(raw + (UnsPS)or1k);
    }
}

//
// Free FIFO port specifications
//
static void freeFifoPorts(or1kP or1k) {

    if(or1k->fifoPorts) {
        STYPE_FREE(or1k->fifoPorts);
        or1k->fifoPorts = 0;
    }
}

//
// Get the next fifo port, if enabled
//
VMI_FIFO_PORT_SPECS_FN(or1kGetFifoPortSpec) {

    or1kP or1k = (or1kP)processor;

    if(!prev) {

        // first port
        return or1k->fifoPorts;

    } else {

        // port other than the first
        Uns32 prevIndex = (prev-or1k->fifoPorts);
        Uns32 thisIndex = prevIndex+1;

        return (thisIndex<NUM_MEMBERS(fifoPorts)) ? &or1k->fifoPorts[thisIndex] : 0;
    }
}
```

```
VMI_CONSTRUCTOR_FN(or1kConstructor) {
    ...
    // create FIFO port descriptions if enabled
    if(OR1K_ENABLE_FIFOS(processor) || params->fifos) {
        newFifoPorts(or1k);
    }
    ...
}

VMI_DESTRUCTOR_FN(or1kDestructor) {
    ...
    // free FIFO port specifications
    freeFifoPorts(or1k);
    ...
}

//
// Make the function available to the simulator, by adding an entry to the
// modelAttrs structure.
//
const vmiIASAttr modelAttrs = {
    ...
    .fifoPortSpecsCB = or1kGetFifoPortSpec,
    ...
};
```

5 Simulator Control

This section describes simulator control functions for simulation exit, restart of processors in a halted state, processor mode change, and simulated interrupt handling. It also discusses some special support for instruction counting.

5.1 *vmirtYield*

Prototype

```
void vmirtYield(vmiProcessorP processor);
```

Description

In a multiprocessor simulation, this function causes the simulator to cease simulation of the passed processor and resume with the next runnable processor.

When using an OP harness, the function will cause a return from `opProcessorSimulate` with a stop reason of `OP_SR_SCHED`.

When using a legacy ICM harness, the function will cause a return from `icmSimulate` with a stop reason of `ICM_SR_SCHED`.

Example

```
#include “vmi/vmiRt.h”

// embedded call made on move to control status register
static void vmic_MoveToCSR(cpuxP cpux, Uns32 csrId, Uns32 value) {

    if((csrId==CPUX_YIELD_CSR) && (value==1)) {
        vmirtYield((vmiProcessorP)cpux);
    } else {
        // handle other CSR writes
        // … etc …
    }
}
```

Notes and Restrictions

1. This function has no effect for processors other than the currently-executing processor.

QuantumLeap Semantics

Non-self-synchronizing

5.2 *vmirtHalt*

Prototype

```
void vmirtHalt(vmiProcessorP processor);
```

Description

This function halts the passed processor. No more instructions will be executed by that processor unless it is restarted by a call to `vmirtRestartNext` or `vmirtRestartNow`. This could be done by another processor (in a multiprocessor simulation) or by an exception (for example, a countdown timer expiring or an external event signal being activated).

If the processor is the currently executing processor, it will halt on completion of the current instruction.

When using an OP harness, the function will cause a return from `opProcessorSimulate` with a stop reason of `OP_SR_HALT`.

When using a legacy ICM harness, the function will cause a return from `icmSimulate` with a stop reason of `ICM_SR_HALT`.

Example

This example is taken from the OVP ARM model. It halts the processor for one of the possible reasons specified by the `reason` parameter, unless the processor is already halted.

```
static void haltProcessor(armP arm, armDisableReason reason) {

    if(!arm->disable) {
        vmirtHalt((vmiProcessorP)arm);
    }

    arm->disable |= reason;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

5.3 vmirtYieldControl

Prototype

```
void vmirtYieldControl(vmiProcessorP processor);
```

Description

This function causes simulation of the passed processor 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_YIELD`.

When using the legacy ICM interface, the function will cause a return from `icmSimulate` or `icmSimulatePlatform` with a stop reason of `ICM_SR_YIELD`.

Example

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

Notes and Restrictions

1. See also function `vmirtInterrupt`, which can be used to cause the processor to stop as if interrupted by a control-C.

QuantumLeap Semantics

Non-self-synchronizing

5.4 *vmirtInterrupt*

Prototype

```
void vmirtInterrupt(vmiProcessorP processor);
```

Description

This function causes simulation of the passed processor to stop on completion of the current simulated instruction and return from the calling context as if it had been interrupted by a control-C in a debug environment.

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

1. See also function `vmirtYieldControl`, which can be used to cause the processor to yield control to the platform.

QuantumLeap Semantics

Non-self-synchronizing

5.5 vmirtExit

Prototype

```
void vmirtExit(vmiProcessorP processor);
```

Description

This function ends execution for the passed processor. In a multiprocessor simulation with distinct top-level instances, other processors will continue execution until they also exit or simulation is finished (see `vmirtFinish`).

If the processor is a member of a cluster or SMP group (see section 17), 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 calls `vmirtExit` will cause *all* members of the group to exit. If `exitmode` is all, then a call to `vmirtExit` will cause only the current processor to exit; other members of the cluster or SMP group will continue to run.

If the processor is the currently executing processor, it will exit on completion of the current instruction.

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.

When using an OP harness, the function will cause a return from `opProcessorSimulate` with a stop reason of `OP_SR_EXIT`.

When using a legacy ICM harness, the function will cause a return from `icmSimulate` with a stop reason of `ICM_SR_EXIT`.

Example

This example is taken from the OVP ARC model. It implements artifact behavior (controlled by a parameter) that causes the processor to exit if it performs a halt.

```
void arcSetHBit(arcP arc, Bool H) {

    if(AUX_FIELD(arc, status32, H) != H) {

        // update the H field in Status32
        AUX_FIELD(arc, status32, H) = H;

        if(!H) {

            // processor restarting
            arcRestartProcessor(arc, AD_HALT);

        } else if(arc->endOnHalt) {

            // test mode: terminate simulation
            vmirtExit((vmiProcessorP)arc);

        } else {
            // normal mode: halt the processor
            arcHaltProcessor(arc, AD_HALT);
        }

        // handle timer state transitions
        arcUpdateTimer(arc, &arc->timers[0]);
        arcUpdateTimer(arc, &arc->timers[1]);

        // handle model artifact timer state transitions
        arcUpdateInstructionCounters(arc);
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

5.6 vmirtFinish

Prototype

```
void vmirtFinish(Int32 status);
```

Description

This function ends execution for all processors in a simulation.

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. It is usually used by intercept libraries implementing a test harness.

When using the OP interface, the function will cause a return from `opProcessorSimulate` or `opRootModuleSimulate` with a stop reason of `OP_SR_FINISH`.

When using the legacy ICM interface, the function will cause a return from `icmSimulate` or `icmSimulatePlatform` with a stop reason of `ICM_SR_FINISH`.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.7 vmirtStop

Prototype

```
void vmirtStop(void);
```

Description

This function causes simulation to be interrupted. If the Imperas MP Debugger is attached, the debug shell will be entered. If there is no Imperas MP Debugger attached, the function behaves in the same way as a call to `vmirtInterrupt` of the current processor.

An example use might be within a store exception handler to detect a write to a magic address that should cause simulation to stop or terminate. It is usually used by intercept libraries implementing a test harness.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.8 vmirtAtomic

Prototype

```
void vmirtAtomic(void);
```

Description

In a parallel simulation, this function causes all other parallel threads to be suspended before it returns. The parallel threads will remain suspended throughout the execution of the simulated instruction which has called `vmirtAtomic` (typically, an embedded call or a function called from an embedded call). If the simulation is not parallel the function has no effect.

This function is typically used before access is made to some data that is shared between parallel threads.

Example

This example is taken from the OVP ARM processor model. In this model, there is some shared data accessible via the `root` pointer on the processor instance. To ensure consistency, all accesses to the root pointer are preceded by a call to `vmirtAtomic`.

```
#include “vmi/vmiRt.h”

inline static armP getClusterRoot(armP arm) {
    vmirtAtomic();
    return arm->root;
}

void armDoSEV(armP arm) {

    // get cluster root
    armP          root      = getClusterRoot(arm);
    vmiProcessorP processor = (vmiProcessorP)root;

    // send event to all PEs in a cluster
    doSEVHier(root);

    // trigger output event if required
    if(root->EVENTO) {
        vmirtWriteNetPort(processor, root->EVENTO, 1);
        vmirtWriteNetPort(processor, root->EVENTO, 0);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.9 vmirtBlock

Prototype

```
void vmirtBlock(vmiProcessorP processor);
```

Description

This function halts the passed processor. No more instructions will be executed by that processor unless it is restarted by a call to `vmirtRestartNext` or `vmirtRestartNow`. This could be done by another processor (in a multiprocessor simulation) or by an exception (for example, a countdown timer expiring or an external event signal being activated).

If the processor is the currently executing processor, the current instruction will be terminated; when the processor restarts, the current instruction will be re-executed.

Example

This example is taken from the OVP MIPS processor model. In this model, some forms of halt that access gating storage should cause instruction re-execution of restart:

```
void mipsForceHaltTC(mipsP tc, Bool reexecute) {

    // when the TC becomes halted, save future values for TCRestart and
    // TCStatus.TDS
    if(GET_VPE(tc)!=tc) {
        mipsSetTmpHaltState(tc);
    }

    // either block or halt the processor, depending on whether the current
    // instruction should be re-executed on restart (gating storage)
    if(reexecute) {
        vmirtBlock((vmiProcessorP)tc);
    } else {
        vmirtHalt((vmiProcessorP)tc);
    }

    // CMP WatchDog timer may be stopped if this was the last running TC on a
    // VPE
    if(GET_CMP(tc)) {
        mipsCMPRefreshWatchDog(tc);
    }
}
```

Notes and Restrictions

1. This function implicitly combines the effect of `vmirtAbortRepeat` and `vmirtHalt`.

QuantumLeap Semantics

Non-self-synchronizing

5.10 *vmirtAbortRepeat*

Prototype

```
void vmirtAbortRepeat(vmiProcessorP processor);
```

Description

This function aborts any active processor instruction and causes it to be restarted. It will usually be used in combination with another run time function call that causes exit from the simulation loop.

Example

This example is taken from the RISCV processor model. In this model when in Debug mode any exception should cause instruction execution to be abandoned and control to be returned to the simulation harness. To implement this, a call to `vmirtAbortRepeat` is followed by a call to `vmirtInterrupt`:

```
static void enterDM(riscvP riscv, dmCause cause) {

    . . . lines delete for clarity . . .

    // interrupt the processor
    vmirtInterrupt((vmiProcessorP)riscv);
}

void riscvTakeException(
    riscvP         riscv,
    riscvException exception,
    Uns64          tval
) {
    if(inDebugMode(riscv)) {

        // terminate execution of program buffer
        vmirtAbortRepeat((vmiProcessorP)riscv);
        enterDM(riscv, DMC_NONE);

    } else {

        . . . normal exception actions when not in Debug mode
    }
}
```

Notes and Restrictions

1. See also function `vmirtBlock`, which implicitly combines the effect of `vmirtAbortRepeat` with `vmirtHalt`.

QuantumLeap Semantics

Non-self-synchronizing

5.11 vmirtIsHalted

Prototype

```
Bool vmirtIsHalted(vmiProcessorP processor);
```

Description

Processor execution may be halted wither by a call to `vmirtHalt` or `vmirtBlock` (described elsewhere in this section) or by the VMI morph time function `vmimtHalt` (defined in `vmiMt.h`). While in the halted state, a processor will execute no instructions.

This function enables a model to determine whether a processor is currently in a halted state. This is typically used in conjunction with `vmirtRestartNow` or `vmirtRestartNext` to restart a halted processor on some event.

Example

This example is taken from the OVP V850 processor model. In this model, sensitivity to input signals depends on whether the processor is halted:

```
inline static Bool isHalted(v850P v850) {
    return vmirtIsHalted((vmiProcessorP)v850);
}

Bool v850InterruptPending(v850P v850) {
    if(isHalted(v850)) {
        return ((v850->FLG_INTPEND != 0) || (v850->FLG_NMIPEND != 0));
    } else {
        return nmiPendingAndEnabled(v850) || intpPendingAndEnabled(v850);
    }
}

void v850TestInterrupt(v850P v850) {
    if(v850InterruptPending(v850)) {
        if(isHalted(v850)) {
            vmirtRestartNow((vmiProcessorP)v850);
        }
        v850ProcessException((vmiProcessorP)v850);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

5.12 vmirtRestartNext

Prototype

```
void vmirtRestartNext(vmiProcessorP processor);
```

Description

Processor execution may be halted wither by a call to `vmirtHalt` or `vmirtBlock` (described elsewhere in this section) or by the VMI morph time function `vmimtHalt` (defined in `vmiMt.h`). While in the halted state, a processor will execute no instructions.

This function enables a model to restart a processor that is in a halted state. This is typically used from within an external interrupt callback function.

The simulator runs each processor in a multiprocessor simulation for a time slice. This function will restart the halted processor at the start of the next time slice; use `vmirtRestartNow` to restart the halted processor immediately.

Example

This example is taken from the OVP MIPS processor model:

```
void mipsForceRestartTC(mipsP tc, mipsDisable reason) {

    // restart either immediately or in the next slice, depending on whether
    // the restart is due to some communication event (temporally significant)
    // or implicit state event (not temporally significant)
    if(reason & RESTART_IMMEDIATE) {
        vmirtRestartNow((vmiProcessorP)tc);
    } else {
        vmirtRestartNext((vmiProcessorP)tc);
    }

    // CMP WatchDog timer may be started if this is the first running TC on a
    // VPE
    if(GET_CMP(tc)) {
        mipsCMPRefreshWatchDog(tc);
    }
}
```

Notes and Restrictions

1. This function has no effect if the target processor is not in the halted state.

QuantumLeap Semantics

Non-self-synchronizing

5.13 vmirtRestartNow

Prototype

```
void vmirtRestartNow(vmiProcessorP processor);
```

Description

Processor execution may be halted wither by a call to `vmirtHalt` (described elsewhere in this section) or by the VMI morph time function `vmimtHalt` (defined in `vmiMt.h`). While in the halted state, a processor will execute no instructions.

This function enables a model to restart a processor that is in a halted state. This is typically used from within an external interrupt callback function as shown in the example below.

The simulator runs each processor in a multiprocessor simulation for a time slice. This function will restart the halted processor immediately; use `vmirtRestartNext` to restart the halted processor at the start of the next time slice.

Example

This example is taken from the OVP MIPS processor model:

```
void mipsForceRestartTC(mipsP tc, mipsDisable reason) {

    // restart either immediately or in the next slice, depending on whether
    // the restart is due to some communication event (temporally significant)
    // or implicit state event (not temporally significant)
    if(reason & RESTART_IMMEDIATE) {
        vmirtRestartNow((vmiProcessorP)tc);
    } else {
        vmirtRestartNext((vmiProcessorP)tc);
    }

    // CMP WatchDog timer may be started if this is the first running TC on a
    // VPE
    if(GET_CMP(tc)) {
        mipsCMPRefreshWatchDog(tc);
    }
}
```

Notes and Restrictions

1. This function has no effect if the target processor is not in the halted state.

QuantumLeap Semantics

Non-self-synchronizing

5.14 vmirtTraceOnAfter

Prototype

```
void vmirtTraceOnAfter(vmiProcessorP processor, Uns64 delta);
```

Description

This function turns on instruction tracing after the given number of instructions. After the given number each instruction executed by the processor will disassembled into the simulator output log.

Calling `vmirtTraceOnAfter` a second time before the specified number of instructions has elapsed will cancel the original request and schedule instruction tracing to start after the new instruction count instead. It is usually used by intercept libraries implementing a test harness.

Example

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

Notes and Restrictions

1. This command will override instruction tracing controlled from the simulator command line, and any previously made calls to `opProcessorTraceOnAfter` (or the legacy function `icmTraceOnAfter`).

QuantumLeap Semantics

Non-self-synchronizing

5.15 vmirtTraceOffAfter

Prototype

```
void vmirtTraceOffAfter(vmiProcessorP processor, Uns64 delta);
```

Description

This function turns off instruction tracing after the given number of instructions. After the given number instruction disassembly will be disabled.

Calling `vmirtTraceOffAfter` a second time before the specified number of instructions has elapsed will cancel the original request and schedule instruction tracing to stop after the new instruction count instead. It is usually used by intercept libraries implementing a test harness.

Example

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

Notes and Restrictions

1. This command will override instruction tracing controlled from the simulator command line, and any previously made calls to `opProcessorTraceOffAfter` (or the legacy function `icmTraceOffAfter`).

QuantumLeap Semantics

Non-self-synchronizing

5.16 vmirtAddTraceRange

Prototype

```
void vmirtAddTraceRange(vmiProcessorP processor, Addr low, Addr high);
```

Description

This function adds an address range to the list of ranges that will be traced. Multiple ranges may be specified using multiple calls, and ranges may be deleted using `vmirtRemoveTraceRange()`.

When any address range is defined, then only instructions inside of the defined ranges will be traced. If there are no ranges defined then all instructions will be traced.

Trace ranges may overlap.

Example

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

Notes and Restrictions

Adding a trace range does not enable tracing – that may be controlled separately using any other method, including the command line, control file, OP API calls or `vmirtTraceOnAfter()`.

When register change tracing is enabled, register changes since the last traced instruction (or, for the first traced instruction, since tracing was enabled) are reported. Thus, when a call from a traced region to a function in an untraced region returns, the register changes reported include all the changes since the last traced instruction, i.e. the instruction that made the call.

A call specifying a range that is already defined will be ignored.

QuantumLeap Semantics

Non-self-synchronizing

5.17 vmirtRemoveTraceRange

Prototype

```
void vmirtRemoveTraceRange(vmiProcessorP processor, Addr low, Addr high);
```

Description

This function removes an address range that was previously added to the list of instruction trace ranges that by a call to `vmirtAddTraceRange()`.

When any address range exists then only instructions inside of the defined ranges will be traced. If there are no ranges defined, for example if all ranges added are subsequently removed, then all instructions will be traced again.

Example

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

Notes and Restrictions

Removing the last trace range does not disable tracing – that may be controlled separately using any other method, including the OP API calls or `vmirtTraceOffAfter()`.

A call specifying a range that is not defined will be ignored.

QuantumLeap Semantics

Non-self-synchronizing

5.18 vmirtFlushTarget

Prototype

```
void vmirtFlushTarget(vmiProcessorP processor, Addr flushPC);
```

Description

This routine flushes any code block from the *current* dictionary at the passed address. It is required when the processor or some associated intercept library has undergone some state change that would make any previously-generated code block in the current dictionary at that address potentially invalid.

Within processor models, it is usually much more efficient to implement modal behavior of this kind by having multiple code dictionaries to represent different modes (see the section *Simulated Memory Access* elsewhere in this document for a detailed explanation of code dictionaries and their interaction with processor modes). However, the ability to flush a target address is often useful in *intercept libraries*. As an example, an intercept library may require to install functionality that is executed on *return* from a function. It can do this using `vmirtFlushTarget` as follows:

1. Intercept the first instruction of the function of interest;
2. Within the primary intercept callback, determine the function return address (for example, by examining the ABI stack pointer register);
3. Use `vmirtFlushTarget` to delete any code block that already exists for that return address;
4. In the intercept library morpher callback, emit an embedded call to perform the action required at the function return address.

Example

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

Notes and Restrictions

1. See related function `vmirtFlushTargetMode`, which enables code blocks to be flushed in a different processor mode to the current mode.

QuantumLeap Semantics

Synchronizing

5.19 vmirtFlushTargetMode

Prototype

```
void vmirtFlushTargetMode(
    vmiProcessorP processor,
    Addr          flushPC,
    Uns32         mode
);
```

Description

This routine flushes any code block from the dictionary indicated by the passed mode at the passed address. It is required when the processor or some associated intercept library has undergone some state change that would make any previously-generated code block in that dictionary at that address potentially invalid.

The ability to flush a target address is often useful in *intercept libraries*. As an example, an intercept library may require to install functionality that is executed in some circumstances on *return from an exception*. It can do this using `vmirtFlushTargetMode` as follows:

1. Intercept the first instruction of the exception handler of interest (in the *kernel mode* dictionary);
2. Within the primary intercept callback, determine the system call return address and processor mode (by examining the processor privileged register state);
3. Use `vmirtFlushTargetMode` to delete any code block that already exists for that return address *in the return mode*;
4. In the intercept library morpher callback, emit an embedded call to perform the action required on return the old mode.

Example

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

Notes and Restrictions

1. See related function `vmirtFlushTarget` which allows flushing of code blocks in the current mode.

QuantumLeap Semantics

Synchronizing

5.20 vmirtFlushTargetModeTagged

Prototype

```
void vmirtFlushTargetMode(
    vmiProcessorP processor,
    Addr          flushPC,
    Uns32         mode,
    vmiBlockTag   tagMask,
    vmiBlockTag   tagValue,
    Bool          flushIfEqual
);
```

Description

This routine conditionally flushes code blocks from the dictionary indicated by the passed mode at the passed address. It is required when the processor has undergone some state change that could make a previously-generated code block in that dictionary at that address potentially invalid.

Whether a block is flushed depends on the `tagMask`, `tagValue` and `flushIfEqual` arguments. If `flushIfEqual` is `True`, then a block will be flushed only if:

```
((block->tag & tagMask) == tagValue)
```

If `flushIfEqual` is `False`, then a block will be flushed only if:

```
((block->tag & tagMask) != tagValue)
```

In these expressions, `block->tag` is the tag associated with the block when it was constructed. See function `vmimtTagBlock` in the *VMI Morph Time Function Reference Manual* for more information.

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:

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

void arcEmitZeroOverheadLoop(arcMorphStateP state) {

    if(state->atZOL) {

        // 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);

        . . .
    }
}
```

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.

QuantumLeap Semantics

Synchronizing

5.21 vmirtFlushDict

Prototype

```
void vmirtFlushDict(vmiProcessorP processor);
```

Description

This routine flushes all code blocks from the current dictionary. It is required when the processor has undergone some state change that would make any previously-generated code blocks in the current dictionary potentially invalid.

It is usually much more efficient to implement modal behavior of this kind by having multiple code dictionaries to represent different modes (see the section *Simulated Memory Access* elsewhere in this document for a detailed explanation of code dictionaries and their interaction with processor modes). However, it occasionally makes sense to implement mode changes by flushing a dictionary instead. An example might be simulation of a processor with a hardware mode that enables a trace buffer to record a sequence of recent branch addresses. In almost all normal operation, the trace buffer is disabled. In the very rare case that the trace buffer is enabled and must be simulated, every branch must be preceded in translated native code by an embedded call to simulate an address store in the buffer. In these circumstances, it is often more convenient to simply flush the current dictionary when the trace buffer is enabled or disabled by a processor state change instead of incurring the performance penalty of conditional code to detect whether the buffer is enabled before every branch.

Example

This example is from the OVP ARC processor model. The ARC processor implements a *stack check* construct in which loads and stores at an offset from the stack pointer register are checked for validity (whether they are in allocated, unallocated or reserved regions, based on values in system state registers).

```
void arcUpdateStackCheck(arcP arc) {

    vmiProcessorP processor = (vmiProcessorP)arc;
    Bool          oldSC     = (arc->blockMask & ARC_BM_STACK_CHECK) && True;
    Bool          newSC     = getSC(arc);

    // flush the current dictionary if this is the first time stack checking
    // has been enabled in this processor mode because blocks will need to be
    // generated with blockMask
    if(newSC && !arc->scEnable[arc->mode]) {
        arc->scEnable[arc->mode] = True;
        vmirtFlushDict(processor);
    }

    if(oldSC != newSC) {

        // update blockMask
        if(newSC) {
            arc->blockMask |= ARC_BM_STACK_CHECK;
        } else {
            arc->blockMask &= ~ARC_BM_STACK_CHECK;
        }

        // update blockMask
        vmirtSetBlockMask(processor, arc->blockMask);
    }
}
```

Notes and Restrictions

1. `vmirtFlushDict` is very slow because all code blocks in a dictionary must be discarded and retranslated. Use `vmirtFlushDict` only in very rare circumstances as described above.
2. See also the related routine `vmirtSetPCFlushDict`, which allows the current program counter to be set while the dictionary is flushed.

QuantumLeap Semantics

Synchronizing

5.22 vmirtFlushAllDicts

Prototype

```
void vmirtFlushAllDicts(vmiProcessorP processor);
```

Description

This routine flushes all code blocks from the all dictionaries of the passed processor. It is required when the processor has undergone some state change that would make any previously-generated code blocks in the any dictionary potentially invalid.

Example

This example is from the OVP ARM processor model. The ARM processor in AArch32 state implements two instruction sets: the traditional ARM instruction set and the more compact Thumb instruction set. For efficiency, the processor model assumes initially that the model will use either ARM or Thumb mode instructions, but not both (simplifying the JIT-compiled code that is produced, making it smaller and faster). If this assumption turns out to be untrue (a branch is detected from ARM to Thumb mode, or vice-versa) all dictionaries are flushed so that code blocks can be recreated with mode switch checking enabled.

```
void armEnableInterworkCheck(armP arm) {

    // discard the current code dictionaries (they need to be regenerated with
    // interwork checking enabled in the block mask)
    vmirtFlushAllDicts((vmiProcessorP)arm);

    // indicate that checking is now enabled
    arm->checkInterwork = True;
}
```

Notes and Restrictions

1. `vmirtFlushAllDicts` is very slow because all code blocks in a dictionary must be discarded and retranslated. Use `vmirtFlushAllDicts` only in very rare circumstances as described above.

QuantumLeap Semantics

Synchronizing

5.23 vmirtGetNetPortHandle

Prototype

```
void vmirtGetNetPortHandle(
    vmiProcessorP processor,
    const char   *netPortName
);
```

Description

This function is deprecated. See section 3.3 for more information about the improved methodology for processor net port connection.

QuantumLeap Semantics

Non-self-synchronizing

5.24 vmirtWriteNetPort

Prototype

```
void vmirtWriteNetPort(
    vmiProcessorP  processor,
    Uns32          handle,
    Uns64          value
);
```

Description

This function writes a value to a processor output net port, the handle of which was set up in the net port specification. See section 3.3 for more information about the methodology for processor net port connection.

Example

This example is from the OVP ARM processor model. This processor implements an event signal, that can be triggered be execution of an `SEV` instruction:

```
void armDoSEV(armP arm) {

    // get cluster root
    armP          root      = getClusterRoot(arm);
    vmiProcessorP processor = (vmiProcessorP)root;

    // send event to all PEs in a cluster
    doSEVHier(root);

    // trigger output event if required
    if(root->EVENTO) {
        vmirtWriteNetPort(processor, root->EVENTO, 1);
        vmirtWriteNetPort(processor, root->EVENTO, 0);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.25 vmirtReadNetPort

Prototype

```
Uns64 vmirtReadNetPort(
    vmiProcessorP  processor,
    Uns32          handle
);
```

Description

This function reads a value from a processor input net port, the handle of which was which was set up in the net port specification. See section 3.3 for more information about the methodology for processor net port connection.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.26 vmirtNetPortHasReceivers

Prototype

```
Bool vmirtNetPortHasReceivers(
    vmiProcessorP  processor,
    Uns32          handle
);
```

Description

This function indicates whether the processor output net port with the given handle has any connected *receivers* (input ports that are notified of writes to the net). Although this function is part of the run-time API, it will typically be used in a morph-time callback to optimize away embedded calls that write nets with no receivers

Example

This example is from the OVP RISC-V processor model. This processor implements an artifact net port, `readCSR`, that is written with the CSR index number when that CSR is read. To avoid unnecessary slow performance, the embedded call that writes the net port is only inserted if the net is connected to one or more receivers:

```
static void reportCSRRead(riscvP riscv, Uns32 code) {
    vmirtWriteNetPort((vmiProcessorP)riscv, riscv->readCSR, code);
}

static void emitCSRRCommon(riscvMorphStateP state, vmiReg rs1, Bool write) {

    riscvP          riscv = state->riscv;
    Uns32           csr   = state->info.csr;
    unpackedReg     rd    = unpackRX(state, 0);
    Uns32           bits  = rd.bits;
    Bool            read  = doCSRRead(state, rd.r);
    riscvCSRAttrsCP attrs = riscvValidateCSRAccess(riscv, csr, read, write);

    // action only required for valid access
    if(attrs) {

        . . . lines omitted for clarity . . .

        // report completion of CSR read if required
        if(
            read &&
            (rdIndex=getRIndex(rd.rA)) &&
            vmirtNetPortHasReceivers((vmiProcessorP)riscv, riscv->readCSR)
        ) {
            vmimtArgProcessor();
            vmimtArgUns32((rdIndex<<16) | csr);
            vmimtCall((vmiCallFn)reportCSRRead);
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

5.27 vmirtInstallNetCallback

Prototype

```
Uns32 vmirtInstallNetCallback(
    vmiProcessorP  processor,
    const char    *netPortName,
    vmiNetChangeFn netChangeCB,
    void          *userData
);
```

Description

This function adds a notifier function to a net port, called when the net is written. The notifier function is of type `vmiNetChangeFn`, and should be declared using macro `VMI_NET_CHANGE_FN` (defined in `vmiTypes.h`):

```
#define VMI_NET_CHANGE_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    void         *userData,     \
    Uns64         newValue      \
)
typedef VMI_NET_CHANGE_FN((*vmiNetChangeFn));
```

Example

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

Notes and Restrictions

1. This function is only for use in an intercept library that is monitoring a model; see section 3.3 for information about the methodology for processor net port connection.
2. Notifiers may only be installed on net ports that are connected to nets in the platform.
3. Notifiers may be installed on both output and input net ports.
4. Note that from VMI version 7.29.0, argument `newValue` has changed from `Uns32` to `Uns64` (effectively widening the maximum width of a single net).

QuantumLeap Semantics

Synchronizing

5.28 vmirtDoSynchronousInterrupt

Prototype

```
void vmirtDoSynchronousInterrupt(vmiProcessorP processor);
```

Description

When modeling processor status registers, it is often the case that a change to a status register will cause a processor interrupt to happen immediately after the instruction completes. For example, if a processor has an instruction specifically to enable interrupts, then any previously pending interrupt should be taken immediately after the enable.

This function should be used in an embedded call to indicate that there are exceptions to be processed immediately *after* this simulated instruction completes.

Example

This example is from the OVP ARM processor model:

```
void armDoInterrupt(armP arm) {

    // wake up the processor if required
    restartProcessor(arm, AD_RESTART_INT);

    // take the interrupt on the next instruction
    if(!arm->disable) {
        vmirtDoSynchronousInterrupt((vmiProcessorP)arm);
    }
}
```

Notes and Restrictions

1. The interrupt will be performed only *after* all actions for the current simulated instruction have completed.
2. The interrupt will always be taken before the next instruction. There is no support for an interrupt after more than one instruction.

QuantumLeap Semantics

Non-self-synchronizing

6 Instruction Counting and Cycle Timers

The VMI Run-Time Function API interface defines functions to:

1. Obtain the nominal IPS (instructions-per-second) rate for a processor;
2. Obtain the current *nominal cycle* count for a processor;
3. Obtain the current *instruction* count for a processor;
4. Set or clear timer callbacks to be triggered after a specified number of cycles have been executed by a processor or SMP group (see section 17 for more information on SMP processor modeling).

Prior to VMI version 2.0.17, each processor allowed only a single cycle timer callback to be specified, within the processor model itself.

In VMI versions 2.0.17 and later, multiple cycle timer callbacks can be specified, and it is now possible to specify and use cycle timer callbacks from within intercept libraries.

In VMI versions 2.0.20 and later, it was possible to create a cycle timer at any level in an SMP processor hierarchy. Prior to this version, cycle timers could only be created at leaf level, and were shared by all members of an SMP group.

In VMI versions 6.1.0 and later, it is possible to create *monotonic cycle timers*, which present a consistent view of monotonically-increasing cycle time for all timers in a platform. With this version, creation of cycle timers at processor levels above individually-schedulable processors is *no longer supported* (cycle timers at this level were used to emulate monotonic timers in earlier VMI versions).

In VMI versions 7.61.0 and later, it is possible to create *abstract periodic counters* that can be shared by multiple processors and peripheral models defined in the BHM API. See section 7 for more information.

6.1 *vmirtGetProcessorIPS*

Prototype

```
Flt64 vmirtGetProcessorIPS(vmiProcessorP processor);
```

Description

This function call returns the nominal *instructions-per-second* speed of the given processor. This is the instructions-per-second implied by the `mips` processor attribute given when the processor was created. This information can be useful when modeling timers that expire after a specified *time interval* rather than a cycle delay (it provides a method to convert from a time delay to an equivalent number of instructions).

For simulations under full control of the built-in scheduler (for example, launched by `opRootModuleSimulate` or the legacy `icmSimulatePlatform`) the nominal instructions-per-second is directly related to the current simulation time. For simulations under external scheduler control (for example, a SystemC simulation using `opProcessorSimulate` or the legacy `icmSimulate` to simulate a processor) it is the external scheduler's responsibility to ensure that the processor simulation rate corresponds to the specified MIPS rate.

Example

This example is from the OVP ARM processor model:

```
//
// Get counter frequency for timers
//
Uns32 armTimerFreq(armP arm) {

    Flt64 ips   = vmirtGetProcessorIPS((vmiProcessorP)arm);
    Uns32 scale = arm->configInfo.timerScaleFactor;

    return ips / (scale ? scale : 1);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

6.2 vmirtGetICount

Prototype

```
Uns64 vmirtGetICount(vmiProcessorP processor);
```

Description

This function returns the *nominal cycle count* for a processor, and is designed for use with timer functions (for example, `vmirtSetModelTimer`).

For a non-SMP processor, or for an SMP leaf processor that is not a member of an SMP group, 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`).

For an SMP leaf level processor that is a member of an SMP group, the count returned does not include halted instructions – these are accumulated at SMP group level.

For an SMP group, the count is the sum of leaf-level member counts and halted instructions for the group as a whole.

For SMP containers that are not groups, the count is the sum of the instructions executed by all current children, according to the definitions above.

Example

This example is extracted from the OVP ARM processor model and shows how `vmirtGetICount` is used in combination with `vmirtSetModelTimer` and `vmirtClearModelTimer` to implement the ARMv6 legacy performance monitor counter.

```
inline static Uns64 getICount64(armP arm) {
    return vmirtGetICount((vmiProcessorP)arm);
}

inline static void setTimer(vmiModelTimerP timer, Uns64 delta) {
    if(timer) {
        vmirtSetModelTimer(timer, delta);
    }
}

inline static void clearTimer(vmiModelTimerP timer) {
    if(timer) {
        vmirtClearModelTimer(timer);
    }
}

static void setCCNT(armP arm, Uns64 value) {

    Uns64 period = SYS_WRITE_MASK_CCNT + 1ULL;
    Bool  active = isCCNTActive(arm);
    Bool  enable = arm->SVResetActive || SYS_FIELD(arm, PMNC, ECC);

    // handle clock divider if required
    if(!arm->SVResetActive && SYS_FIELD(arm, PMNC, D)) {
        value  <<= 6;
        period <<= 6;
    }

    // update base depending on counter active state
    if(active) {
        arm->CCNTBase = getICount64(arm) - value;
    } else {
        arm->CCNTBase = value;
    }

    // set or clear timer depending on whether active counter should generate
    // an event on overflow
    if(active && enable && !SYS_FIELD(arm, PMNC, CCR)) {
        setTimer(arm->timerCCNT, period-value);
    } else {
        clearTimer(arm->timerCCNT);
    }
}
```

Notes and Restrictions

1. Related function `vmirtGetExecutedICount` can be used to obtain the number of instructions executed by a processor excluding halted and skipped instructions, if required.

QuantumLeap Semantics

Non-self-synchronizing

6.3 *vmirtGetExecutedICount*

Prototype

```
Uns64 vmirtGetExecutedICount(vmiProcessorP processor);
```

Description

This function returns the exact count of executed instructions for a processor (excluding halted and skipped instructions).

Example

This example is from the OVP RISC-V processor model. This model implements an `instret` register, which shows the count of retired instructions since some time in the past:

```
//
// Return current instruction count
//
inline static Uns64 getInstructions(riscvP riscv) {
    return vmirtGetExecutedICount((vmiProcessorP)riscv);
}

//
// Common routine to read instret counter
//
static Uns64 instretR(riscvP riscv) {
    return getInstructions(riscv) - riscv->baseInstructions;
}

//
// Common routine to write instret counter (NOTE: count is notionally
// incremented *before* the write)
//
static void instretW(riscvP riscv, Uns64 newValue) {
    riscv->baseInstructions = getInstructions(riscv) - newValue + 1;
}
```

Notes and Restrictions

1. Related function `vmirtGetICount` can be used to obtain the number of instructions executed by a processor including halted and skipped instructions, if required.

QuantumLeap Semantics

Non-self-synchronizing

6.4 vmirtSetICountInterrupt

Prototype

```
void vmirtSetICountInterrupt(vmiProcessorP processor, Uns64 iDelta);
```

Description

This function causes the *default instruction count interrupt callback* to be called when the processor cycle count equals the current processor cycle count plus the count specified with the `iDelta` argument.

The cycle count interrupt callback is defined with the `VMI_ICOUNT_FN` macro and installed using the `icountExceptCB` field of the processor `vmiIASAttr` attribute structure (see `vmiAttrs.h`). The `VMI_ICOUNT_FN` macro is defined in `vmiTypes.h` as:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiModelTimerP timer,       \
    Uns64          iCount,      \
    void          *userData     \
)
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The `processor` argument is the processor on which the timer has expired. The `timer` argument is the implicit model timer that is created automatically for every processor model (see function `vmirtCreateModelTimer` in this section for information about how to create explicit timers if more than one timer is required). The `iCount` argument is the current cycle count. The `userData` argument is always `NULL` for the default cycle count callback.

The callback should analyze the processor state and indicate that an interrupt is pending by calling `vmirtDoSynchronousInterrupt` if required. The `processor` argument to `vmirtDoSynchronousInterrupt` may be the current processor or any other processor in the simulation. Every processor subject to a synchronous interrupt call will immediately stop simulating, and the simulator will call the processor’s instruction fetch handler to determine what to do next. The processor instruction fetch exception handler is responsible for arbitrating between the cycle count exception and any other pending exceptions and taking appropriate action (for example, entering kernel mode and setting the program counter to a simulated exception handler).

Example

This example shows timer implementation from the OVP ARC model. In this model, a set of timers are modeled using the implicit model timer, by always selecting the timer that is next to expire:

```
static Bool setTimeout(arcP arc, Uns64 iCount) {

    Bool       pending  = False;
    Uns64      minDelta = -1;
    arcTimerId id;
    // determine delay to first timeout
    for(id=0; id<AT_ID_LAST; id++) {

        if(timerActive(arc, id)) {

            pending = True;

            Uns64 delta = arc->timerCount[id] - iCount;

            if(minDelta > delta) {
                minDelta = delta;
            }
        }
    }

    // set instruction count timeout if required, and return a boolean
    // indicating whether a timer was set
    if(pending) {
        vmirtSetICountInterrupt((vmiProcessorP)arc, minDelta);
        return True;
    } else {
        return False;
    }
}
```

Notes and Restrictions

1. If `vmirtSetICountInterrupt` is called multiple times then each call overrides the count specified with the previous call.

QuantumLeap Semantics

Non-self-synchronizing

6.5 vmirtClearICountInterrupt

Prototype

```
void vmirtClearICountInterrupt(vmiProcessorP processor);
```

Description

Function `vmirtClearICountInterrupt` clears any cycle count interrupt callback that uses the *default cycle count interrupt callback* previously enabled with `vmirtSetICountInterrupt`.

Example

This example shows timer implementation from the OVP ARC model. In this model, a set of timers are modeled using the implicit model timer, by always selecting the timer that is next to expire:

```
static void stopTimer(arcP arc, arcTimerId id, arcTimState state) {

    Uns64 iCount = getICount(arc);
    Uns64 iDelta = arc->timerCount[id] - iCount;

    // put the timer into the stopped state
    arc->timerState[id] = state;

    // if this was the last active timer, clear instruction count interrupt
    if(!setTimeout(arc, iCount)) {
        vmirtClearICountInterrupt((vmiProcessorP)arc);
    }

    // convert the stored value to a DELTA
    arc->timerCount[id] = iDelta;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

6.6 *vmirtCreateModelTimer*

Prototype

```
vmiModelTimerP vmirtCreateModelTimer(
    vmiProcessorP processor,
    vmiICountFn   icountCB,
    Uns32         scale,
    void         *userData
);
```

Description

Function `vmirtCreateModelTimer` creates and returns a new timer for the passed processor. When the timer expires, function `icountCB` is called. This *count interrupt callback* is defined with the `VMI_ICOUNT_FN` macro from `vmiTypes.h` as:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiModelTimerP timer,       \
    Uns64          iCount,      \
    void          *userData     \
)
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The `processor` argument is the processor on which the timer has expired. The `timer` argument is the timer object returned by `vmirtCreateModelTimer`. The `iCount` argument is the current cycle count. The `userData` argument is the value that was originally passed as the `userData` argument to `vmirtCreateModelTimer` when the timer was created.

If the callback is implementing a feature of a processor model, it should analyze the processor state and indicate that an interrupt is pending by calling `vmirtDoSynchronousInterrupt` if required. The `processor` argument to `vmirtDoSynchronousInterrupt` may be the current processor or any other processor in the simulation. Every processor subject to a synchronous interrupt call will immediately stop simulating, and the simulator will call the processor's instruction fetch handler to determine what to do next. The processor instruction fetch exception handler is responsible for arbitrating between the cycle count exception and any other pending exceptions and taking appropriate action (for example, entering kernel mode and setting the program counter to a simulated exception handler).

The timer created by `vmirtCreateModelTimer` expires when the processor has executed *exactly the number of cycles specified*. In a multiprocessor simulation, in which each processor is simulated in turn for a time slice, this can give the appearance of time moving backwards in some circumstances. See related function `vmirtCreateMonotonicModelTimer` for information on monotonic timers, which all present a consistent view of simulation time which does not appear to move backwards.

This function may also be used in intercept libraries to install periodic callbacks for analysis purposes. In this case, the callback should directly manipulate data associated with the intercept library, and not call `vmirtDoSynchronousInterrupt`.

The timer returned by `vmirtCreateModelTimer` is initially disabled. It should be enabled for a specific cycle count by calling `vmirtSetModelTimer`.

The `scale` argument allows the timer rate to be specified as a fraction of the processor nominal MIPS rate. For example, a `scale` of 10 implies that the counter increments once every 10 cycles. A `scale` of 0 is treated as if 1 was specified (the counter will increment at exactly the configured processor MIPS rate).

Example

This example shows the CMP module watchdog timer implementation from the OVP MIPS model. Note that this model implements a microthreaded processor, and the timer is at the *simulation group object*[2] level (the VPE object in MIPS parlance).

```
#include "vmi/vmiRt.h"

static VMI_ICOUNT_FN(iCountPendingCB) {

    . . .
}

static mipsCMPVLocalP allocCMPVPELocals(mipsP tc) {

    mipsP          vpe          = VPE_FOR_TC(tc);
    mipsCMPVLocalP cmpVPELocals = vpe->cmpVPELocals;

    if(!cmpVPELocals) {

        cmpVPELocals = STYPE_CALLOC(mipsCMPVLocal);

        // link cmpVPELocals to VPE
        vpe->cmpVPELocals = cmpVPELocals;

        . . .

        // create timer for CMP count/compare exceptions on this VPE
        cmpVPELocals->pintTimer = vmirtCreateMonotonicModelTimer(
            (vmiProcessorP)vpe, iCountPendingCB, 0, 0
        );

        // create WatchDog timer on this VPE
        cmpVPELocals->wdTimer = vmirtCreateModelTimer(
            (vmiProcessorP)vpe, wdPendingCB, 0, 0
        );
    }

    return cmpVPELocals;
}
```

Notes and Restrictions

1. Explicit model timers (created with this call) can coexist with the implicit model timer, used by the default count interrupt callback (see `vmirtSetICountInterrupt`): models may have either, or both.

[2] See section 17.1 for the definition of *simulation group object*.

2. The `scale` argument exists only from VMI version 5.7.0.
3. Timers created by `vmirtCreateModelTimer` must be on *self-contained objects*, *simulation group objects* or *simulation group members*; they may not be created at other container levels. See section 17.1 for a definition of these terms.

QuantumLeap Semantics

Synchronizing

6.7 vmirtCreateMonotonicModelTimer

Prototype

```
vmiModelTimerP vmirtCreateMonotonicModelTimer(
    vmiProcessorP processor,
    vmiICountFn   icountCB,
    Uns32         scale,
    void         *userData
);
```

Description

Function `vmirtCreateMonotonicModelTimer` is identical to `vmirtCreateModelTimer` except that:

1. The timer created presents a *monotonically-increasing view of time*, when considered together with all other monotonic timers in a simulation; and:
2. Monotonic timers may only be created on *self-contained objects* or *simulation group objects*; they may not be created on *simulation group members*. See section 17.1 for a definition of these terms.

The algorithm for determining the timer value in a multiprocessor simulation is as follows:

1. When the timer is accessed, the *implied current processor time* is determined. This is derived from the *current processor cycle count* combined with the current processor nominal *instructions-per-second* speed.
2. The *implied current processor time* is then compared with the *current monotonic minimum time*.
3. If the implied current processor time is *earlier* than the current monotonic minimum time, the timer count value is recalculated, based on the current monotonic minimum time.
4. If the implied current processor time is *later* than the current monotonic minimum time, the current monotonic minimum time is updated to the implied current processor time.

Example

This example shows the CMP module count/compare timer implementation from the OVP MIPS model. Note that this model implements a microthreaded processor, and the timer is at the *simulation group object*[3] level (the VPE object in MIPS parlance).

```
#include "vmi/vmiRt.h"

static VMI_ICOUNT_FN(iCountPendingCB) {

    . . .
}

static mipsCMPVLocalP allocCMPVPELocals(mipsP tc) {

    mipsP          vpe          = VPE_FOR_TC(tc);
    mipsCMPVLocalP cmpVPELocals = vpe->cmpVPELocals;
```

[3] See section 17.1 for the definition of *simulation group object*.

```
    if(!cmpVPELocals) {

        cmpVPELocals = STYPE_CALLOC(mipsCMPVLocal);

        // link cmpVPELocals to VPE
        vpe->cmpVPELocals = cmpVPELocals;

        . . .

        // create timer for CMP count/compare exceptions on this VPE
        cmpVPELocals->pintTimer = vmirtCreateMonotonicModelTimer(
            (vmiProcessorP)vpe, iCountPendingCB, 0, 0
        );

        // create WatchDog timer on this VPE
        cmpVPELocals->wdTimer = vmirtCreateModelTimer(
            (vmiProcessorP)vpe, wdPendingCB, 0, 0
        );
    }

    return cmpVPELocals;
}
```

Notes and Restrictions

1. Explicit model timers (created with this call) can coexist with the implicit model timer, used by the default count interrupt callback (see `vmirtSetICountInterrupt`): models may have either, or both.
2. The `scale` argument exists only from VMI version 5.7.0.
3. Timers created by `vmirtCreateMonotonicModelTimer` must be on *self-contained objects* or *simulation group objects*; they may not be created on *simulation group members* or at other container levels. See section 17.1 for a definition of these terms.

QuantumLeap Semantics

Synchronizing

6.8 vmirtDeleteModelTimer

Prototype

```
void vmirtDeleteModelTimer(vmiModelTimerP modelTimer);
```

Description

Function `vmirtDeleteModelTimer` deletes any timer previously created with `vmirtCreateModelTimer`. If the timer is active, it is disabled prior to deletion.

Example

This example shows the CMP module `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded processor, and the timer is at the *simulation group object*[4] level (the VPE object in MIPS parlance).

```
#include "vmi/vmiRt.h"

static void freeCMPVPELocals(mipsP mips) {

    mipsCMPVLocalP cmpVPELocals = mips->cmpVPELocals;

    if(
        cmpVPELocals &&
        (IS_VPE(mips) || (IS_TC(mips) && (VPE_FOR_TC(mips)==mips)))
    ) {
        vmirtDeleteModelTimer(cmpVPELocals->pintTimer);
        vmirtDeleteModelTimer(cmpVPELocals->wdTimer);
        STYPE_FREE(cmpVPELocals);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

[4] See section 17.1 for the definition of *simulation group object*.

6.9 vmirtSetModelTimer

Prototype

```
void vmirtSetModelTimer(vmiModelTimerP modelTimer, Uns64 delta);
```

Description

This function causes the *counter interrupt callback* associated with the passed timer to be called when the *timer tick count* equals the *current timer tick count* plus the count specified with the `delta` argument. The timer tick count is the cycle count of the processor on which the timer is installed divided by the scale factor specified for the timer when `vmirtCreateModelTimer` was called. See the description of `vmirtGetICount` for information about how cycles are counted for different kinds of processor.

The counter interrupt callback is defined with the `VMI_ICOUNT_FN` macro and passed as the `icountCB` argument to `vmirtCreateModelTimer`. The `VMI_ICOUNT_FN` macro is defined in `vmiTypes.h` as:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiModelTimerP timer,       \
    Uns64          iCount,      \
    void          *userData     \
)
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The `processor` argument is the processor on which the timer has expired. The `timer` argument is the timer previously returned by `vmirtCreateModelTimer`. The `iCount` argument is the current cycle count (note that this will be an exact multiple of the tick count). The `userData` argument is the value that was originally passed as the `userData` argument to `vmirtCreateModelTimer` when the timer was created.

The callback should analyze the processor state and indicate that an interrupt is pending by calling `vmirtDoSynchronousInterrupt` if required. The `processor` argument to `vmirtDoSynchronousInterrupt` may be the current processor or any other processor in the simulation. Every processor subject to a synchronous interrupt call will immediately stop simulating, and the simulator will call the processor's instruction fetch handler to determine what to do next. The processor instruction fetch exception handler is responsible for arbitrating between the counter exception and any other pending exceptions and taking appropriate action (for example, entering kernel mode and setting the program counter to a simulated exception handler).

Example

This example shows the CMP module `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded processor, and the timer is at the *simulation group object*[5] level (the VPE object in MIPS parlance).

[5] See section 17.1 for the definition of *simulation group object*.

```
#include “vmi/vmiRt.h”

static void scheduleTimerInterrupt(
    mipsP       vpe,
    const char *reason,
    Bool        atExpiry
) {
    vmiModelTimerP modelTimer = vpe->cmpVPELocals->pintTimer;

    if(!CMPG_FIELD(vpe, GIC_SH_CONFIG, COUNTSTOP)) {

        // counter enabled - schedule the next interrupt
        Uns64 delta = getLocalCompareDelta(vpe);
        delta = !delta && atExpiry ? -1ULL : delta;

        // set CMP timer
        vmirtSetModelTimer(modelTimer, delta);

        // emit debug output if required
        if(MIPS32_DEBUGCMP(vpe)) {
            vmiMessage(
                "I", CPU_PREFIX,
                "%s: %s (GIC_VPE_Compare=0x"FMT_64x" GIC_Counter=0x"FMT_64x") - "
                "schedule CMP timer interrupt after "FMT_64u" (0x"FMT_64x")",
                GET_NAME(vpe), reason, getCompare(vpe), getCount(vpe),
                delta, delta
            );
        }

    } else if(!atExpiry) {

        . . .
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.10 vmirtClearModelTimer

Prototype

```
void vmirtClearModelTimer(vmiModelTimerP modelTimer);
```

Description

Function `vmirtClearModelTimer` disables the passed model timer. It will remain disabled until a subsequent call to `vmirtSetModelTimer`.

Example

This example shows the CMP module `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded processor, and the timer is at the *simulation group object*[6] level (the VPE object in MIPS parlance).

```
#include “vmi/vmiRt.h”

static void scheduleTimerInterrupt(
    mipsP       vpe,
    const char *reason,
    Bool        atExpiry
) {
    vmiModelTimerP modelTimer = vpe->cmpVPELocals->pintTimer;

    if(!CMPG_FIELD(vpe, GIC_SH_CONFIG, COUNTSTOP)) {

        . . . .

    } else if(!atExpiry) {

        // counter disabled - no interrupt should be scheduled
        vmirtClearModelTimer(modelTimer);

        // emit debug output if required
        if(MIPS32_DEBUGCMP(vpe)) {
            vmiMessage(
                "I", CPU_PREFIX,
                "%s: %s (GIC_VPE_Compare=0x"FMT_64x" GIC_Counter=0x"FMT_64x") - "
                "clear CMP timer interrupt",
                GET_NAME(vpe), reason, getCompare(vpe), getCount(vpe)
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

[6] See section 17.1 for the definition of *simulation group object*.

6.11 vmirtIsModelTimerEnabled

Prototype

```
Bool vmirtIsModelTimerEnabled(vmiModelTimerP modelTimer);
```

Description

Function `vmirtIsModelTimerEnabled` returns a boolean indicating if the passed timer is currently enabled.

Example

This example shows how the OVP MIPS model uses `vmirtIsModelTimerEnabled` to set a timer to trigger a debug single-step exception if required. The timer should be set only if it is not already activated.

```
#include “vmi/vmiRt.h”

void mipsEnableDebugSingleStepException(mipsP tc) {

    if(
        debugSingleStepExceptionEnabled(tc) &&
        !vmirtIsModelTimerEnabled(tc->sstTimer)
    ) {
        vmirtSetModelTimer(tc->sstTimer, 1);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.12 vmirtGetModelTimerCurrentCount

Prototype

```
Uns64 vmirtGetModelTimerCurrentCount(
    vmiModelTimerP modelTimer,
    Bool           ticks
);
```

Description

Function `vmirtGetModelTimerCurrentCount` returns the current value of the passed timer. If `ticks` is `False`, the returned value is the *cycle count* of the processor with which the timer is associated. If `ticks` is `True`, the returned value is the equivalent tick count, calculated by dividing the cycle count by the scale specified when the timer was created.

Example

This example shows how the OVP ARM model uses `vmirtGetModelTimerCurrentCount` when printing debug messages describing timer updates. The debug message prints both the current cycle count and the count at which an interrupt is scheduled to occur.

```
#include “vmi/vmiRt.h”

inline static Uns64 getTimerCurrentICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerCurrentCount(timer, False);
}

inline static Uns64 getTimerExpiryICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerExpiryCount(timer, False);
}

static void scheduleIntLT(armP arm, armLTimerP timer, const char *reason) {

    Int64 delta = instructionsToWrapLT(timer);

    delta -= timer->SR.scale;

    if(delta<=0) {
        delta += timer->SR.period;
    }

    vmirtSetModelTimer(timer->vmiTimer, delta);

    MP_INFO(arm, "_STI",
        "%s - icount="FMT_64u" schedule %s timer interrupt at "FMT_64u,
        reason,
        getTimerCurrentICount(timer->vmiTimer),
        timer->name,
        getTimerExpiryICount(timer->vmiTimer)
    );
}
```

Notes and Restrictions

1. This function exists only in VMI version 5.7.0 and above.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.13 vmirtGetModelTimerExpiryCount

Prototype

```
Uns64 vmirtGetModelTimerExpiryCount(
    vmiModelTimerP modelTimer,
    Bool           ticks
);
```

Description

Function `vmirtGetModelTimerExpiryCount` returns the count at which the passed timer is scheduled to expire (i.e. call the counter interrupt function associated with the timer). If `ticks` is `False`, the returned value is the *cycle count* at which the timer will expire. If `ticks` is `True`, the returned value is the equivalent tick count, calculated by dividing the cycle count by the scale specified when the timer was created.

Example

This example shows how the OVP ARM model uses `vmirtGetModelTimerExpiryCount` when printing debug messages describing timer updates. The debug message prints both the current cycle count and the count at which an interrupt is scheduled to occur.

```
#include “vmi/vmiRt.h”

inline static Uns64 getTimerCurrentICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerCurrentCount(timer, False);
}

inline static Uns64 getTimerExpiryICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerExpiryCount(timer, False);
}

static void scheduleIntLT(armP arm, armLTimerP timer, const char *reason) {

    Int64 delta = instructionsToWrapLT(timer);

    delta -= timer->SR.scale;

    if(delta<=0) {
        delta += timer->SR.period;
    }

    vmirtSetModelTimer(timer->vmiTimer, delta);

    MP_INFO(arm, "_STI",
        "%s - icount="FMT_64u" schedule %s timer interrupt at "FMT_64u,
        reason,
        getTimerCurrentICount(timer->vmiTimer),
        timer->name,
        getTimerExpiryICount(timer->vmiTimer)
    );
}
```

Notes and Restrictions

1. This function exists only in VMI version 5.7.0 and above.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

7 Abstract Periodic Counters

From VMI version 7.61.0, the VMI Run-Time Function API interface defines functions to create and use *abstract periodic counters*. These provide:

1. Counters that increment at a configured frequency; and:
2. The ability to schedule event callbacks when the counter exceeds a particular value or is explicitly modified.

Abstract periodic counters can be either local to a processor model or shared with other processor models or peripherals defined using the BHM API.

7.1 vmirtCreateModelCounter

Prototype

```
vmiModelCounterP vmirtCreateModelCounter(
    vmiProcessorP processor,
    const char   *name,
    vmiCounterFn  counterCB,
    Flt64         frequency,
    Uns32         bits,
    void         *userData
);
```

Description

Function `vmirtCreateModelCounter` creates a new *abstract periodic counter* associated with the given processor. If `name` is `NULL`, the counter is local to the processor. If `name` is non-`NULL`, the counter is shared with any other processors or BHM peripherals that define a counter of the same name. The `frequency` argument specifies the counter frequency, in megahertz. The `bits` argument specifies the number of bits that the counter implements; the counter value will wrap round when it overflows what can be represented in this number of bits. The `userData` argument is a client-specific data pointer that is passed to the counter callback function when the counter expires or is modified.

The `counterCB` argument is a callback function that is called when the counter expires or is modified by `vmirtSetCounterValue` (or `bhmSetCounterValue`, if the counter is shared with a BHM peripheral). The `vmiCounterFn` type is defined in `vmiTypes.h` as follows:

```
#define VMI_COUNTER_FN(_NAME) void _NAME( \
    vmiProcessorP    processor,     \
    vmiModelCounterP counter,       \
    void            *userData,      \
    Uns64            countValue,    \
    Uns64            overrun,       \
    vmiCounterReason reason         \
)
typedef VMI_COUNTER_FN((*vmiCounterFn));
```

The arguments to this callback are as follows:

1. `processor`: the processor associated with the counter when it was created.
2. `counter`: the counter object.
3. `userData`: the client-specific data pointer passed to `vmirtCreateModelCounter`.
4. `countValue`: expiry count scheduled by `vmirtCounterDelay` (see section 7.4).
5. `overrun`: if non-zero, the amount by which the counter has overrun due to quantum scheduling effects (see section 7.4).
6. `reason`: the reason why the callback was called, specified by the `vmiCounterReason` type (see below).

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

```
typedef enum vmiCounterReasonE {
    VMICR_TRIGGERED,    // triggered by counter expiry
    VMICR_CHANGE        // counter value has been written
} vmiCounterReason;
```

This type is used to determine whether the callback has been invoked because the delay indicated by a call to `vmirtCounterDelay` has expired (if `VMICR_TRIGGERED`) or whether the counter has been modified (if `VMICR_CHANGE`).

Function `vmirtCreateModelCounter` returns a pointer to the counter object, or `NULL` in any of these error cases:

1. The frequency is either zero or negative.
2. The frequency, when converted to Hz, exceeds the value representable in a 64-bit unsigned number.
3. The counter bit size is either zero or greater than 64.

If this is a shared counter and either the frequency or bit size differ from a previous definition, then a warning is issued and the previous definition is used.

Example

This example shows how the OVP RISC-V model uses `vmirtCreateModelCounter` to create a shared counter that supplies the value of the `mtime` memory-mapped counter. The counter name is configurable using a model parameter (`mtime_counter`), another parameter (`mtime_Hz`) gives the counter frequency, and a third parameter (`mtime_bits`) gives the counter bit width:

```
#include “vmi/vmiRt.h”

void riscvNewCounters(riscvP riscv) {
    riscv->mtime = vmirtCreateModelCounter(
        (vmiProcessorP)riscv,
        riscv->configInfo.mtime_counter,
        mtimeCB,
        riscv->configInfo.mtime_Hz/1e6,
        riscv->configInfo.mtime_bits,
        0
    );
}
```

The callback `mtimeCB` is used to schedule counter callbacks to implement interrupts because of `mtimecmp`, `stimecmp` and `vstimecmp` compare registers:

```
//
// Refresh timer interrupt state controlled by mtimecmp, stimecmp and vstimecmp
//
static void updateTimer(riscvP hart, Uns64 mtime) {

    Bool  mtimecmpPresent = hart->clint;
    Bool  stimecmpPresent = Sstc(hart);
    Uns64 timeout         = -1;

    // handle mtimecmp effect
    if(mtimecmpPresent) {

        Uns32 intNum   = exceptionToInt(riscv_E_MTimerInterrupt);
        Bool  oldValue = hart->ip[0] & (1<<intNum);
        Uns64 mtimecmp = riscvReadMTIMECMP(hart);
        Bool  newValue = updateTimerInt(hart, mtime, mtimecmp, &timeout);
        // modify mtip pending state if required
        if(oldValue!=newValue) {
            riscvUpdateInterrupt(hart, intNum, newValue);
        }
    }

    // handle stimecmp effect
    if(stimecmpPresent) {

        Uns64 stimecmp = RD_CSR64(hart, stimecmp);
        Bool  newValue = updateTimerInt(hart, mtime, stimecmp, &timeout);

        // modify stip pending state if required
        if(hart->stimecmp!=newValue) {
            hart->stimecmp = newValue;
            riscvUpdatePending(hart);
        }
    }

    // handle vstimecmp effect
    if(stimecmpPresent && hypervisorPresent(hart)) {

        Uns64 vstimecmp = RD_CSR64(hart, vstimecmp);
        Uns64 vmtime    = mtime + RD_CSR64(hart, htimedelta);
        Bool  newValue  = updateTimerInt(hart, vmtime, vstimecmp, &timeout);

        // modify vstip pending state if required
        if(hart->vstimecmp!=newValue) {
            hart->vstimecmp = newValue;
            riscvUpdatePending(hart);
        }
    }

    // set next timeout if required
    if(mtimecmpPresent || stimecmpPresent) {
        vmirtCounterDelay(hart->mtime, mtime+timeout);
    }
}

//
// Timer timeout callback: refresh timer
//
static VMI_COUNTER_FN(mtimeCB) {
    updateTimer((riscvP)processor, countValue);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.61.0 and above.

QuantumLeap Semantics

Synchronizing

7.2 vmirtGetCounterValue

Prototype

```
Uns64 vmirtGetCounterValue(vmiModelCounterP counter);
```

Description

Function `vmirtGetCounterValue` returns the current value of the given abstract periodic counter. This value is the later of:

1. The counter value corresponding to the effective time of the current processor, based on its configured MIPS rate and the number of cycles it has executed; and:
2. The latest value previously read for the counter by another processor or BHM peripheral.

This definition means that abstract periodic counters always increase *monotonically*: it is not possible for a processor to read a value corresponding to an earlier time than has already been seen by another processor or peripheral. However, the quantized nature of a multiprocessor simulation may mean that the counter value does not increase smoothly within a quantum.

Example

This example shows how the OVP RISC-V model uses `vmirtGetCounterValue` to implement a read of the `mtime` memory-mapped counter:

```
#include “vmi/vmiRt.h”

Uns64 riscvReadMTIME(riscvP hart) {
    return vmirtGetCounterValue(hart->mtime);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.61.0 and above.

QuantumLeap Semantics

Synchronizing

7.3 *vmirtSetCounterValue*

Prototype

```
void vmirtSetCounterValue(vmiModelCounterP counter, Uns64 newValue)
```

Description

Function `vmirtSetCounterValue` updates the current value of the given abstract periodic counter, which will then continue incrementing from that value at its configured rate. The given `newValue` is masked to the bit size specified for the counter.

If the counter value changes because of this call, then:

1. Any peripheral threads waiting for the counter are restarted with reason `BHM_RR_CHANGE` (see the *OVP BHM and PPM API Function Reference* manual for more information).
2. Any callbacks scheduled by `vmirtCounterDelay` for all views of the counter are cancelled.
3. The `counterCB` callbacks for all views of the counter are called with reason `VMICR_CHANGE`.

The `counterCB` callback is called so that each counter view can react to the new counter value, and perhaps schedule a new counter event using `vmirtCounterDelay`.

Example

This example shows how the OVP RISC-V model uses `vmirtSetCounterValue` to implement a write of the `mtime` memory-mapped counter by a CLINT model:

```
#include "vmi/vmiRt.h"

void riscvWriteMTIME(riscvP hart, Uns64 value) {
    vmirtSetCounterValue(hart->mtime, value);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.61.0 and above.

QuantumLeap Semantics

Synchronizing

7.4 vmirtCounterDelay

Prototype

```
void vmirtCounterDelay(vmiModelCounterP counter, Uns64 countValue);
```

Description

Function `vmirtCounterDelay` schedules an expiry of the abstract model counter to occur when the counter value equals or passes the given `countValue`, which is masked to the counter bit width. The scheduled expiry replaces any currently scheduled expiry event (each counter only ever has a single scheduled event).

When the scheduled expiry happens, the `counterCB` callback for the counter is called with reason `VMICR_TRIGGERED`. The `counterCB` callback is passed the `countValue` given here, to indicate the requested expiry count. Depending on the size of the quantum and scheduled delay size, this value may be *less* than the current counter value as read by `vmirtGetCounterValue`, meaning that the counter appears to overrun before the callback occurs: if this is the case, the `overrun` parameter to the `counterCB` callback is non-zero and indicates the difference between the current counter value and the requested expiry value. If overruns are undesirable, reduce the simulation quantum size so that counter delays are always scheduled *after* the end of the current quantum, which will ensure that the `counterCB` `countValue` argument always matches the value returned by `vmirtGetCounterValue` and that `overrun` is zero.

Example

See section 7.1 for an example of this function used in the OVP RISC-V model.

Notes and Restrictions

1. This function exists only in VMI version 7.61.0 and above.

QuantumLeap Semantics

Synchronizing

7.5 *vmirtCounterWaitChange*

Prototype

```
void vmirtCounterWaitChange(vmiModelCounterP counter);
```

Description

Function `vmirtCounterWaitChange` cancels any scheduled expiry of the abstract model counter. The `counterCB` callback for the counter will therefore never be called with reason `VMICR_TRIGGERED`, but it will still be called with reason `VMICR_CHANGE` if the counter value is ever set.

Example

No OVP models use this function currently.

Notes and Restrictions

1. This function exists only in VMI version 7.61.0 and above.

QuantumLeap Semantics

Synchronizing

8 Simulated Time Access Functions

Functions in this section are designed to allow simulated time to be accessed, usually from an intercept library, and also to allow a *quantum timer* to be created that will be called at each quantum boundary.

Time is fundamentally represented in *simulator ticks*, which are counted using `Uns64` values. The size of a tick is configured by the *time precision* specified when the platform is created. If no time precision is explicitly specified, the precision defaults to one picosecond (1e-9 seconds), meaning that there are 1 billion simulator ticks per simulated second; use the function `opRootModuleSetSimulationTimePrecision` in the OP interface to change the precision if required. Using the default picosecond time precision, the largest representable time is about 585 simulated years. Specifying higher time precision will proportionately reduce the largest representable time; for example, using a time precision of one femtosecond reduces the largest representable time to about 214 simulated days.

Some functions in this interface allow times to be accessed as 64-bit floating point `vmiTime` values. These are convenient to operate on but cannot exactly represent all possible tick values when time is more than 1/2048th of the maximum representable time. In cases when this is important, an alternative set of functions that return time in terms of fundamental simulator ticks is available. There is also a function to obtain the configured simulator ticks per second for the current simulation (`vmirtGetTicksPerSecond`). The value returned by this function can be used to convert from simulator ticks to time.

8.1 *vmirtGetQuantumStartTime*

Prototype

```
vmiTime vmirtGetQuantumStartTime(vmiProcessorP processor);
```

Description

Function `vmirtGetQuantumStartTime` returns the simulated time of the *start* of the current quantum. This is primarily intended for use within timing estimator intercept libraries. The `vmiTime` result is a floating point value giving time in seconds.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    vmiPrintf("quantum start  : %fus\n", uS(vmirtGetQuantumStartTime(processor)));
    vmiPrintf("quantum end    : %fus\n", uS(vmirtGetQuantumEndTime(processor)));
    vmiPrintf("local time     : %fus\n", uS(vmirtGetLocalTime(processor)));
    vmiPrintf(“monotonic time : %fus\n", uS(vmirtGetMonotonicTime(processor)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.35.0 and above.
2. See also function `vmirtGetQuantumStartTicks`, which returns the equivalent value in simulator ticks.

QuantumLeap Semantics

Thread safe

8.2 vmirtGetQuantumEndTime

Prototype

```
vmiTime vmirtGetQuantumEndTime(vmiProcessorP processor);
```

Description

Function `vmirtGetQuantumEndTime` returns the simulated time of the *end* of the current quantum. This is primarily intended for use within timing estimator intercept libraries. The `vmiTime` result is a floating point value giving time in seconds.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    vmiPrintf("quantum start  : %fus\n", uS(vmirtGetQuantumStartTime(processor)));
    vmiPrintf("quantum end    : %fus\n", uS(vmirtGetQuantumEndTime(processor)));
    vmiPrintf("local time     : %fus\n", uS(vmirtGetLocalTime(processor)));
    vmiPrintf(“monotonic time : %fus\n", uS(vmirtGetMonotonicTime(processor)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.35.0 and above.
2. See also function `vmirtGetQuantumEndTicks`, which returns the equivalent value in simulator ticks.

QuantumLeap Semantics

Thread safe

8.3 vmirtGetLocalTime

Prototype

```
vmiTime vmirtGetLocalTime(vmiProcessorP processor);
```

Description

Function `vmirtGetLocalTime` returns the *local* simulated time implied by the state of the given processor, calculated from its current cycle count and configured MIPS rate. When called at the end of a quantum, the result is the same as returned by `vmirtGetQuantumEndTime`; otherwise, the returned time will lie between the quantum start time (as returned by `vmirtGetQuantumStartTime`) and the quantum end time. The `vmiTime` result is a floating point value giving time in seconds.

The processor local time is derived from the given processor state only. This means that in a multiprocessor simulation, local times for each processor are independent, and times may appear to go backwards within the granularity of a quantum when queried for different processors. Related function `vmirtGetMonotonicTime` instead returns a calculated time that is guaranteed to monotonically increase.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    vmiPrintf("quantum start  : %fus\n", uS(vmirtGetQuantumStartTime(processor)));
    vmiPrintf("quantum end    : %fus\n", uS(vmirtGetQuantumEndTime(processor)));
    vmiPrintf("local time     : %fus\n", uS(vmirtGetLocalTime(processor)));
    vmiPrintf(“monotonic time : %fus\n", uS(vmirtGetMonotonicTime(processor)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.35.0 and above.
2. See also function `vmirtGetLocalTicks`, which returns the equivalent value in simulator ticks.

QuantumLeap Semantics

Non-self Synchronizing

8.4 vmirtGetMonotonicTime

Prototype

```
vmiTime vmirtGetMonotonicTime(vmiProcessorP processor);
```

Description

Function `vmirtGetMonotonicTime` returns the *monotonic* simulated time implied by the state of the given processor, calculated from its current cycle count and configured MIPS rate. When called at the end of a quantum, the result is the same as returned by `vmirtGetQuantumEndTime`; otherwise, the returned time will lie between the quantum start time (as returned by `vmirtGetQuantumStartTime`) and the quantum end time. The `vmiTime` result is a floating point value giving time in seconds.

The monotonic time is guaranteed to increase through the simulation. This means that in a multiprocessor simulation, the time observed when `vmirtGetMonotonicTime` is called for one processor may affect the value returned when the same function is called for another processor – in effect, the first call sets a floor for the time value, and the second call is clamped to this floor value if less.

Monotonic times simply data presentation in some classes of tool at the cost of potential lumpiness in the returned value. In addition, using this function in one processor context can cause visible side effects in unrelated processor contexts (because of the value clamping).

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    vmiPrintf("quantum start  : %fus\n", uS(vmirtGetQuantumStartTime(processor)));
    vmiPrintf("quantum end    : %fus\n", uS(vmirtGetQuantumEndTime(processor)));
    vmiPrintf("local time     : %fus\n", uS(vmirtGetLocalTime(processor)));
    vmiPrintf(“monotonic time : %fus\n", uS(vmirtGetMonotonicTime(processor)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.35.0 and above.
2. See also function `vmirtGetMonotonicTicks`, which returns the equivalent value in simulator ticks.

QuantumLeap Semantics

Synchronizing

8.5 vmirtGetTicksPerSecond

Prototype

```
Uns64 vmirtGetTicksPerSecond(vmiProcessorP processor);
```

Description

Function `vmirtGetTicksPerSecond` returns the number of simulator ticks per second that are implied by the configured time precision (see the introduction to this section for more details about this). The returned value can be used to convert between simulated ticks and simulated time.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    Uns64 tps               = vmirtGetTicksPerSecond(processor);
    Uns64 quantumStartTicks = vmirtGetQuantumStartTicks(processor);
    Uns64 quantumEndTicks   = vmirtGetQuantumEndTicks(processor);
    Uns64 localTicks        = vmirtGetLocalTicks(processor);
    Uns64 monotonicTicks    = vmirtGetMonotonicTicks(processor);
    vmiPrintf("quantum start  : %fus\n", uS(quantumStartTicks / ((Flt64)tps)));
    vmiPrintf("quantum end    : %fus\n", uS(quantumEndTicks   / ((Flt64)tps)));
    vmiPrintf("local time     : %fus\n", uS(localTicks        / ((Flt64)tps)));
    vmiPrintf(“monotonic time : %fus\n", uS(monotonicTicks    / ((Flt64)tps)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.58.0 and above.

QuantumLeap Semantics

Synchronizing

8.6 vmirtGetQuantumStartTicks

Prototype

```
Uns64 vmirtGetQuantumStartTicks(vmiProcessorP processor);
```

Description

Function `vmirtGetQuantumStartTicks` returns the simulator ticks of the *start* of the current quantum. This is primarily intended for use within timing estimator intercept libraries.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    Uns64 tps               = vmirtGetTicksPerSecond(processor);
    Uns64 quantumStartTicks = vmirtGetQuantumStartTicks(processor);
    Uns64 quantumEndTicks   = vmirtGetQuantumEndTicks(processor);
    Uns64 localTicks        = vmirtGetLocalTicks(processor);
    Uns64 monotonicTicks    = vmirtGetMonotonicTicks(processor);
    vmiPrintf("quantum start  : %fus\n", uS(quantumStartTicks / ((Flt64)tps)));
    vmiPrintf("quantum end    : %fus\n", uS(quantumEndTicks   / ((Flt64)tps)));
    vmiPrintf("local time     : %fus\n", uS(localTicks        / ((Flt64)tps)));
    vmiPrintf(“monotonic time : %fus\n", uS(monotonicTicks    / ((Flt64)tps)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.58.0 and above.
2. See also function `vmirtGetQuantumStartTime`, which returns the equivalent value in simulated time.

QuantumLeap Semantics

Thread safe

8.7 *vmirtGetQuantumEndTicks*

Prototype

```
Uns64 vmirtGetQuantumEndTicks(vmiProcessorP processor);
```

Description

Function `vmirtGetQuantumEndTicks` returns the simulator ticks of the *end* of the current quantum. This is primarily intended for use within timing estimator intercept libraries.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    Uns64 tps               = vmirtGetTicksPerSecond(processor);
    Uns64 quantumStartTicks = vmirtGetQuantumStartTicks(processor);
    Uns64 quantumEndTicks   = vmirtGetQuantumEndTicks(processor);
    Uns64 localTicks        = vmirtGetLocalTicks(processor);
    Uns64 monotonicTicks    = vmirtGetMonotonicTicks(processor);
    vmiPrintf("quantum start  : %fus\n", uS(quantumStartTicks / ((Flt64)tps)));
    vmiPrintf("quantum end    : %fus\n", uS(quantumEndTicks   / ((Flt64)tps)));
    vmiPrintf("local time     : %fus\n", uS(localTicks        / ((Flt64)tps)));
    vmiPrintf(“monotonic time : %fus\n", uS(monotonicTicks    / ((Flt64)tps)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.58.0 and above.
2. See also function `vmirtGetQuantumEndTime`, which returns the equivalent value in simulated time.

QuantumLeap Semantics

Thread safe

8.8 vmirtGetLocalTicks

Prototype

```
Uns64 vmirtGetLocalTicks(vmiProcessorP processor);
```

Description

Function `vmirtGetLocalTicks` returns the *local* simulator ticks implied by the state of the given processor, calculated from its current cycle count and configured MIPS rate. When called at the end of a quantum, the result is the same as returned by `vmirtGetQuantumEndTicks`; otherwise, the returned simulator ticks will lie between the quantum start ticks (as returned by `vmirtGetQuantumStartTicks`) and the quantum end ticks.

The processor local simulator tick count is derived from the given processor state only. This means that in a multiprocessor simulation, tick counts for each processor are independent, and ticks may appear to go backwards within the granularity of a quantum when queried for different processors. Related function `vmirtGetMonotonicTicks` instead returns a calculated simulator ticks value that is guaranteed to monotonically increase.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    Uns64 tps               = vmirtGetTicksPerSecond(processor);
    Uns64 quantumStartTicks = vmirtGetQuantumStartTicks(processor);
    Uns64 quantumEndTicks   = vmirtGetQuantumEndTicks(processor);
    Uns64 localTicks        = vmirtGetLocalTicks(processor);
    Uns64 monotonicTicks    = vmirtGetMonotonicTicks(processor);
    vmiPrintf("quantum start  : %fus\n", uS(quantumStartTicks / ((Flt64)tps)));
    vmiPrintf("quantum end    : %fus\n", uS(quantumEndTicks   / ((Flt64)tps)));
    vmiPrintf("local time     : %fus\n", uS(localTicks        / ((Flt64)tps)));
    vmiPrintf(“monotonic time : %fus\n", uS(monotonicTicks    / ((Flt64)tps)));
}

static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.58.0 and above.
2. See also function `vmirtGetLocalTime`, which returns the equivalent value in simulated time.

QuantumLeap Semantics

Non-self Synchronizing

8.9 vmirtGetMonotonicTicks

Prototype

```
Uns64 vmirtGetMonotonicTicks(vmiProcessorP processor);
```

Description

Function `vmirtGetMonotonicTicks` returns the *monotonic* simulator ticks implied by the state of the given processor, calculated from its current cycle count and configured MIPS rate. When called at the end of a quantum, the result is the same as returned by `vmirtGetQuantumEndTicks`; otherwise, the returned time will lie between the quantum start ticks (as returned by `vmirtGetQuantumStartTicks`) and the quantum end ticks.

The monotonic tick count is guaranteed to increase through the simulation. This means that in a multiprocessor simulation, the tick count observed when `vmirtGetMonotonicTicks` is called for one processor may affect the value returned when the same function is called for another processor – in effect, the first call sets a floor for the tick count value, and the second call is clamped to this floor value if less.

Monotonic tick counts simply data presentation in some classes of tool at the cost of potential lumpiness in the returned value. In addition, using this function in one processor context can cause visible side effects in unrelated processor contexts (because of the value clamping).

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries and on read and write accesses to memory.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

#define READ_DELAY       1
#define WRITE_DELAY      2

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static void printStats(vmiProcessorP processor) {
    Uns64 tps               = vmirtGetTicksPerSecond(processor);
    Uns64 quantumStartTicks = vmirtGetQuantumStartTicks(processor);
    Uns64 quantumEndTicks   = vmirtGetQuantumEndTicks(processor);
    Uns64 localTicks        = vmirtGetLocalTicks(processor);
    Uns64 monotonicTicks    = vmirtGetMonotonicTicks(processor);
    vmiPrintf("quantum start  : %fus\n", uS(quantumStartTicks / ((Flt64)tps)));
    vmiPrintf("quantum end    : %fus\n", uS(quantumEndTicks   / ((Flt64)tps)));
    vmiPrintf("local time     : %fus\n", uS(localTicks        / ((Flt64)tps)));
    vmiPrintf(“monotonic time : %fus\n", uS(monotonicTicks    / ((Flt64)tps)));
}
```

```
static VMI_MEM_WATCH_FN(watchRead) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, READ_DELAY, False);
    }
}

static VMI_MEM_WATCH_FN(watchWrite) {
    if(processor) {
        printStats(processor);
        vmirtAddSkipCount(processor, WRITE_DELAY, False);
    }
}

static VMI_QUANTUM_TIMER_FN(timerCB) {
    printStats(processor);
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback (domain, 0, 0, 0xffffffff, watchRead,  0);
    vmirtAddWriteCallback(domain, 0, 0, 0xffffffff, watchWrite, 0);
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}
```

Notes and Restrictions

1. This function exists only in VMI version 7.58.0 and above.
2. See also function `vmirtGetMonotonicTime`, which returns the equivalent value in simulated time.

QuantumLeap Semantics

Synchronizing

8.10 vmirtCreateQuantumTimer

Prototype

```
vmiQuantumTimerP vmirtCreateQuantumTimer(
    vmiProcessorP     processor,
    vmiQuantumTimerFn timerCB,
    void             *userData
)
```

Description

Function `vmirtCreateQuantumTimer` creates a timer that calls the function `timerCB` at every simulation quantum boundary. The function is of type `vmiQuantumTimerFn`, defined by the macro `VMI_QUANTUM_TIMER_FN` in `vmiTypes.h` as follows:

```
#define VMI_QUANTUM_TIMER_FN(_NAME) void _NAME( \
    vmiProcessorP    processor,     \
    vmiQuantumTimerP timer,         \
    void             *userData      \
)
typedef VMI_QUANTUM_TIMER_FN((*vmiQuantumTimerFn));
```

The callback function is passed the processor context provided when the timer was created and an application-specific data pointer.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static VMI_QUANTUM_TIMER_FN(timerCB) {
    vmiPrintf("quantum start  : %fus\n", uS(vmirtGetQuantumStartTime(processor)));
    vmiPrintf("quantum end    : %fus\n", uS(vmirtGetQuantumEndTime(processor)));
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}

VMIOS_POST_SIMULATE_FN(postSimulate) {
    vmirtDeleteQuantumTimer(object->timer);
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.35.0 and above.

QuantumLeap Semantics

Synchronizing

8.11 vmirtDeleteQuantumTimer

Prototype

```
void vmirtDeleteQuantumTimer(vmiQuantumTimerP quantumTimer)
```

Description

Function `vmirtDeleteQuantumTimer` deletes a timer previously created by `vmirtCreateQuantumTimer`.

Example

This example shows how this function could be used in an intercept library which accumulates timing information at quantum boundaries.

```
#include "vmi/vmiRt.h"

typedef struct vmiosObjectS {
    vmiQuantumTimerP timer;
} vmiosObject;

static Flt64 uS(vmiTime time) {
    return time*1000000;
}

// accumulate data in this function
static VMI_QUANTUM_TIMER_FN(timerCB) {
    vmiPrintf("quantum start  : %fus\n", uS(vmirtGetQuantumStartTime(processor)));
    vmiPrintf("quantum end    : %fus\n", uS(vmirtGetQuantumEndTime(processor)));
}

VMIOS_CONSTRUCTOR_FN(constructor) {
    object->timer = vmirtCreateQuantumTimer(processor, timerCB, 0);
}

VMIOS_POST_SIMULATE_FN(postSimulate) {
    vmirtDeleteQuantumTimer(object->timer);
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.35.0 and above.

QuantumLeap Semantics

Synchronizing

9 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 could be used with processor models or as part of an intercept library. There are two ways in which timing behavior can be modified:

1. Processors may be *derated*, indicating that they should operate at a lower simulated speed than the configured MIPS rate. For example, if a processor with a nominal MIPS rate of 100 is derated by 40%, it will run in the simulator with a simulated MIPS rate of 60 MIPS instead.

2. *Cycle delays* may be fed back into the simulation algorithm. A typical use would be to specify that a memory access takes additional number of cycles and that the processor should therefore skip this number of cycles before resuming simulation.

9.1 vmirtSetDerateFactor

Prototype

```
void vmirtSetDerateFactor(vmiProcessorP processor, Flt64 factor);
```

Description

Function `vmirtSetDerateFactor` sets the *deration factor* for a processor. This is the factor by which the processor executes instructions more slowly than the nominal MIPS rate. The factor is expressed as a floating point percentage in the range `0.0` to `100.0` (values outside this range are ignored).

As an example, a deration factor of `0.0` (the default) implies that the processor is not derated, and will execute instructions at the configured nominal MIPS rate.

As another example, a deration factor of `30.0` implies that the processor is derated by 30% and will therefore execute instructions at only 70% of the nominal MIPS rate. When such a processor is asked to execute for 100 clocks (for example, using `opProcessorSimulate` or the legacy function `icmSimulate`) it will execute only 70 instructions, and 30 will be added to the *skipped instruction count* for the processor.

If a deration factor of `100.0` is specified then the processor is *stalled* and will not execute instructions.

Example

This example shows how a model might typically convert from a model-specific priority code to a deration factor.

```
#include "vmi/vmiRt.h"

static void setThreadState(vmiProcessorP thread, Uns32 priority) {

    Flt64 derateFactor;

    // determine deration factor
    if(priority==0) {
        derateFactor = 100.0;
    } else if(priority==1) {
        derateFactor = 50.0;
    } else {
        derateFactor = 0.0;
    }

    // apply the deration factor
    if(derateFactor != vmirtGetDerateFactor(thread)) {
        vmirtSetDerateFactor(thread, derateFactor);
    }
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.23.0 and above.
2. See also function `vmirtAddSkipCount`, which allows instruction-specific timing effects to be emulated.

QuantumLeap Semantics

Synchronizing

9.2 vmirtGetDerateFactor

Prototype

```
Flt64 vmirtGetDerateFactor(vmiProcessorP processor);
```

Description

Function `vmirtGetDerateFactor` returns the current *deration factor* for a processor. See function `vmirtSetDerateFactor` for a description of deration factors.

Example

```
#include “vmi/vmiRt.h”

static void setThreadState(vmiProcessorP thread, Uns32 priority) {

    Flt64 derateFactor;

    // determine deration factor
    if(priority==0) {
        derateFactor = 100.0;
    } else if(priority==1) {
        derateFactor = 50.0;
    } else {
        derateFactor = 0.0;
    }

    // apply the deration factor
    if(derateFactor != vmirtGetDerateFactor(thread)) {
        vmirtSetDerateFactor(thread, derateFactor);
    }
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.23.0 and above.

QuantumLeap Semantics

Thread safe

9.3 *vmirtAddSkipCount*

Prototype

```
void vmirtAddSkipCount(
    vmiProcessorP processor,
    Uns64         skipCount,
    Bool          defer
);
```

Description

Function `vmirtAddSkipCount` adds to the count of *skipped nominal instructions* (or cycles) for a processor. This function will typically be used in a processor model or (more typically) an intercept library to approximate model timing effects.

The general use model is as follows:

1. Processors are instantiated with a *nominal MIPS rate*. This is specified by the `MIPS` parameter on the processor instance, or a default rate of 100 MIPS is used if this parameter is not set. In the absence of calls to `vmirtSetDerateFactor` or `vmirtAddSkipCount`, all instructions will be executed at this nominal rate.
2. If `vmirtAddSkipCount` is used, then the processor nominal instruction count is advanced by the specified count, reducing the budget available for execution of instructions.
3. If both a skip count and deration factor are in force, then the skip count is applied first followed by the deration factor. For example, if a processor with skip count 4 and deration factor 50.0 is required to execute ten nominal instructions, then only three true instructions will be executed, calculated as follows:
   Budget after skipped instructions: 10-4 = 6 nominal instructions
   Budget after deration: 6 x 0.5 = 3 true instructions

The `defer` argument indicates whether the count of skipped instructions should be updated immediately (if `defer` is `False`) or aggregated and applied at the start of the next quantum (if `defer` is `True`). Immediate application of skipped cycles may increase fidelity but reduces simulator performance.

Example

This example shows an intercept library using instruction attributes to adjust simulation timing. In this simple model, one extra nominal instruction is added for each load and two extra nominal instructions for each store.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiProcessorP processor;
} vmiosObject;

VMIOS_CONSTRUCTOR_FN(constructor) {
    object->processor = processor;
}
```

```
VMIOS_DESTRUCTOR_FN(destructor) {
    vmiPrintf(
        "ACCUMULATED TOTAL DELAY: "FMT_Au"\n",
        vmirtGetSkipCount(processor, VMIST_ALL)
    );
}

static void addToDelay(vmiosObjectP object, Uns32 delay) {
    vmirtAddSkipCount(object->processor, DELAY, False);
}

static VMIOS_MORPH_FN(morphCallback) {

    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, OCL_DS_ADDRESS, False);

    if(attrs) {

        octiaMemAccessP ma;
        Uns32           loads  = 0;
        Uns32           stores = 0;
        Uns32           delay;

        // count total number of loads ans stores
        for(
            ma = ocliaGetFirstMemAccess(attrs);
            ma;
            ma = ocliaGetNextMemAccess(ma)
        ) {
            switch(ocliaGetMemAccessType(ma)) {
                case OCL_MAT_LOAD:
                    loads++;
                    break;
                case OCL_MAT_STORE:
                    stores++;
                    break;
                default:
                    break;
            }
        }

        // assume loads take one extra cycle and stores two extra cycles
        delay = loads + (stores*2);

        // add to processor instruction count
        if(delay) {
            vmimtArgNatAddress(object);
            vmimtArgUns32(delay);
            vmimtCall((vmiCallFn)addToDelay);
        }
    }

    ocliaFreeAttrs(attrs);

    return 0;
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.23.0 and above; argument `defer` is present only from VMI version 7.11.0.

QuantumLeap Semantics

Synchronizing

9.4 vmirtGetSkipCount

Prototype

```
Uns64 vmirtGetSkipCount(vmiProcessorP processor, vmiSkipType type);
```

Description

Function `vmirtGetSkipCount` returns the number of skipped instructions accumulated by calls to `vmirtAddSkipCount` (or the equivalent morph-time primitives `vmimtAddSkipCountC` and `vmimtAddSkipCountR`) and as a result of any deration factor specified by `vmirtSetDerateFactor`.

Skipped instruction counts are in one of two states. *Committed* skipped instruction counts are reflected in the return value from `vmirtGetICount`, and in time access functions (for example, `vmirtGetLocalTime`). *Pending* skipped instruction counts are not yet accounted for by `vmirtGetICount`, or by time access functions. At the start of each quantum, any pending count is reflected in the committed count, and the pending count correspondingly reduced. The same commit process happens when function `vmirtAddSkipCount` is used with a `defer` argument of `False`. Deferring commitment of skipped instruction counts in this way greatly improves simulator performance.

The `vmiSkipType` type specifies the exact count to be returned by this function. It is defined in `vmiTypes.h` as follows:

```
typedef enum vmiSkipTypeE {
    VMIST_COMMITTED,    // committed skipped cycles
    VMIST_PENDING,      // pending skipped cycles
    VMIST_ALL,          // committed and pending skipped cycles
} vmiSkipType;
```

If the `type` argument has value `VMIST_COMMITTED`, this function will return the current *committed* skipped instruction count.

If the `type` argument has value `VMIST_PENDING`, this function will return the current *pending* skipped instruction count.

If the `type` argument has value `VMIST_ALL`, this function will return the sum of the current *committed* and *pending* skipped instruction counts.

Example

This example shows an intercept library using instruction attributes to adjust simulation timing. In this simple model, one extra nominal instruction is added for each load and two extra nominal instructions for each store. `vmirtGetSkipCount` is used at simulation end to return the total number of extra cycles added.

```
#include “vmi/vmiRt.h”

typedef struct vmiosObjectS {
    vmiProcessorP processor;
} vmiosObject;

VMIOS_CONSTRUCTOR_FN(constructor) {
    object->processor = processor;
}

VMIOS_DESTRUCTOR_FN(destructor) {
    vmiPrintf(
        "ACCUMULATED TOTAL DELAY: "FMT_Au"\n",
        vmirtGetSkipCount(processor, VMIST_ALL)
    );
}

static void addToDelay(vmiosObjectP object, Uns32 delay) {

    // explicit accounting
    object->totalDelay += delay;

    // implicit accounting
    vmirtAddSkipCount(object->processor, DELAY, False);
}

static VMIOS_MORPH_FN(morphCallback) {

    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, OCL_DS_ADDRESS, False);

    if(attrs) {

        octiaMemAccessP ma;
        Uns32           loads  = 0;
        Uns32           stores = 0;
        Uns32           delay;

        // count total number of loads ans stores
        for(
            ma = ocliaGetFirstMemAccess(attrs);
            ma;
            ma = ocliaGetNextMemAccess(ma)
        ) {
            switch(ocliaGetMemAccessType(ma)) {
                case OCL_MAT_LOAD:
                    loads++;
                    break;
                case OCL_MAT_STORE:
                    stores++;
                    break;
                default:
                    break;
            }
        }

        // assume loads take one extra cycle and stores two extra cycles
        delay = loads + (stores*2);

        // add to processor instruction count
        if(delay) {
            vmimtArgNatAddress(object);
            vmimtArgUns32(delay);
            vmimtCall((vmiCallFn)addToDelay);
        }
    }

    ocliaFreeAttrs(attrs);

    return 0;
}
```

Notes and Restrictions

1. This function exists only in VMI version 6.23.0 and above; argument `type` is present only from VMI version 7.11.0.

QuantumLeap Semantics

Synchronizing

10 Simulated Memory Access

The VMI morph time interface (described in `vmiMt.h`) allows simple memory accesses such as load and store instructions to be modeled very efficiently by translated native code. See the *VMI Morph Time Function Reference* for more details about this.

Sometimes the required behavior is more complex and cannot easily or efficiently be modeled using the morph time interface. For example, a processor might implement a single instruction that is able to move many bytes of data in memory – for example, the x86 `movs` instruction. In this case, run time functions for modeling memory accesses are available, described in this section.

10.1 vmirtReadNByteDomain

Prototype

```
memMapped vmirtReadNByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    void          *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    memAccessAttrs attrs
);
```

Description

This function reads `size` bytes starting at address `simAddress` from the passed domain into the buffer. Typically, this will be called from an embedded function call (created by `vmimtCall` or similar at morph time).

If `cachedRegion` is non-`NULL`, then it should point to a *memory region cache*, typically in the processor structure. This region cache is an opaque type, `memRegionP`, which can be used in combination with `vmirtGetReadNByteSrc` for faster access when the same area of memory is used by successive reads.

If buffer is `NULL`, then the function performs all access checks required for the read access without returning the value read.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

If the value is `MEM_AA_TRUE`, then this is a *true processor read* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact read* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

If the value is `MEM_AA_TRUE|MEM_AA_FETCH`, then this is a *true processor fetch* that should generate a call to the processor simulated exception handler on an access violation.

If the value is `MEM_AA_FALSE|MEM_AA_FETCH`, then this is an *artifact fetch* and not a true processor access (usually, an access made by the JIT code generator).

Additionally, the argument controls the value passed as the `processor` argument to read callback functions: see `vmirtAddReadCallback` for more details.

The return value of the function is a member of the `memMapped` enumeration:

```
typedef enum memMappedE {
    MEM_MAP_NONE = 0,              // no addresses in the range mapped
    MEM_MAP_PART = 1,              // some addresses in the range mapped
    MEM_MAP_FULL = 2,              // all addresses in the range mapped
} memMapped;
```

If no address in the range is readable, the function returns `MEM_MAP_NONE`; if all addresses in the range are readable, the function returns `MEM_MAP_FULL`; otherwise, the function returns `MEM_MAP_PART`.

Example

```
#include “vmi/vmiRt.h”

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;      // source region cache
    memRegionP dstRegion;      // destination region cache
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src;
    Uns8 *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. There is no automatic validation that the target buffer is large enough to hold `size` bytes. It is the calling function’s responsibility to verify this.

2. If the value size is 1, 2, 4 or 8 bytes, use the appropriate `vmirtRead[1248]ByteDomain` function instead (which also allows the endianness of the value in memory to be specified).
3. `memAccessAttrs` value `MEM_AA_IGNORE_PRIV` previously supported by the VMI API is now deprecated and redefined to be equivalent to `MEM_AA_TRUE`.

QuantumLeap Semantics

Synchronizing

10.2 vmirtReadNByteDomainVM

Prototype

```
memMapped vmirtReadNByteDomainVM(
    memDomainP     domain,
    Addr           simAddress,
    void          *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    Addr           VA,
    memAccessAttrs attrs
);
```

Description

This function is identical to `vmirtReadNByteDomain`, except that it additionally takes a nominal virtual address argument (`VM`). If callbacks of type `vmiMemReadFn` or `vmiMemWatchFn` (or equivalent platform callbacks) are activated by this call, then this nominal virtual address will be passed as the `VM` argument to those callbacks. By contrast, `vmirtReadNByteDomain` always passes the same value as both `address` and `VA` to such callbacks.

This function is intended to be used when structures such as cache models are implemented using the VMI API. It allows virtual addresses reported as the `VA` argument of a function of type `vmiMemReadFn` that implements the cache to passed on to structures beyond the cache in the memory subsystem.

For details of other arguments to this function, see notes for function `vmirtReadNByteDomain`.

Example

```
#include “vmi/vmiRt.h”

static void readMemTrue(
    cacheInfoP cInfo,
    Addr       pa,
    Addr       va,
    void      *data,
    Uns32      nBytes,
    Bool       isFetch
) {
    memAccessAttrs attrs;
    memDomainP     domain;
    memRegionPP    rCache;

    if(isFetch) {
        attrs  = MEM_AA_TRUE | MEM_AA_FETCH;
        domain = cInfo->cacheBackDomain.d[MPT_CODE];
        rCache = &cInfo->rCacheCode;
    } else {
        attrs  = MEM_AA_TRUE;
        domain = cInfo->cacheBackDomain.d[MPT_DATA];
        rCache = &cInfo->rCacheData;
    }

    vmirtReadNByteDomainVA(domain, pa, data, nBytes, rCache, va, attrs);
}
```

Notes and Restrictions
See notes for function `vmirtReadNByteDomain`.

QuantumLeap Semantics
Synchronizing

10.3 vmirtWriteNByteDomain

Prototype

```
memMapped vmirtWriteNByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    const void    *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    memAccessAttrs attrs
);
```

Description

This function writes `size` bytes to address `simAddress` in the passed domain from the buffer. Typically, this will be called from an embedded function call (created by `vmimtCall` or similar at morph time).

If `cachedRegion` is non-`NULL`, then it should point to a *memory region cache*, typically in the processor structure. This region cache is an opaque type, `memRegionP`, which can be used in combination with `vmirtGetWriteNByteDst` for faster access when the same area of memory is used by successive writes.

If buffer is `NULL`, then the function performs all access checks required for the write access without updating the value in memory.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

Value `MEM_AA_FETCH` is used only for read accesses and ignored for writes.

If the value is `MEM_AA_TRUE`, then this is a *true processor write* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact write* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

Additionally, the argument controls the value passed as the `processor` argument to write callback functions: see `vmirtAddWriteCallback` for more details.

The return value of the function is a member of the `memMapped` enumeration:

```
typedef enum memMappedE {
    MEM_MAP_NONE = 0,               // no addresses in the range mapped
    MEM_MAP_PART = 1,               // some addresses in the range mapped
    MEM_MAP_FULL = 2,               // all addresses in the range mapped
} memMapped;
```

If no address in the range is writable, the function returns `MEM_MAP_NONE`; if all addresses in the range are writable, the function returns `MEM_MAP_FULL`; otherwise, the function returns `MEM_MAP_PART`.

Example

```
#include "vmi/vmiRt.h"

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;       // source region cache
    memRegionP dstRegion;       // destination region cache
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src;
    Uns8 *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP) x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. There is no automatic validation that the source buffer is large enough to hold `size` bytes. It is the calling function's responsibility to verify this.
2. If the value size is 1, 2, 4 or 8 bytes, use the appropriate `vmirtWrite[1248]ByteDomain` function instead (which also allows the endianness of the value in memory to be specified).
3. `memAccessAttrs` value `MEM_AA_IGNORE_PRIV` previously supported by the VMI API is now deprecated and redefined to be equivalent to `MEM_AA_TRUE`.

QuantumLeap Semantics

Synchronizing

10.4 vmirtWriteNByteDomainVM

Prototype

```
memMapped vmirtWriteNByteDomainVM(
    memDomainP     domain,
    Addr           simAddress,
    const void    *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    Addr           VM,
    memAccessAttrs attrs
);
```

Description

This function is identical to `vmirtWriteNByteDomain`, except that it additionally takes a nominal virtual address argument (`VM`). If callbacks of type `vmiMemWriteFn` or `vmiMemWatchFn` (or equivalent platform callbacks) are activated by this call, then this nominal virtual address will be passed as the `VM` argument to those callbacks. By contrast, `vmirtWriteNByteDomain` always passes the same value as both `address` and `VA` to such callbacks.

This function is intended to be used when structures such as cache models are implemented using the VMI API. It allows virtual addresses reported as the `VA` argument of a function of type `vmiMemWriteFn` that implements the cache to passed on to structures beyond the cache in the memory subsystem.

For details of other arguments to this function, see notes for function `vmirtWriteNByteDomain`.

Example

```
#include “vmi/vmiRt.h”

static void writeMemTrue(
    cacheInfoP  cInfo,
    Addr        pa,
    Addr        va,
    const void *data,
    Uns32       nBytes
) {
    memAccessAttrs attrs  = MEM_AA_TRUE;
    memDomainP     domain = cInfo->cacheBackDomain.d[MPT_DATA];
    memRegionPP    rCache = &cInfo->rCacheData;

    vmirtWriteNByteDomainVA(domain, pa, data, nBytes, rCache, va, attrs);
}
```

Notes and Restrictions

See notes for function `vmirtWriteNByteDomain`.

QuantumLeap Semantics

Synchronizing

10.5 vmirtRead[1248]ByteDomain

Prototypes

```
Uns8 vmirtRead1ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memAccessAttrs attrs
);
Uns16 vmirtRead2ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    memAccessAttrs attrs
);
Uns32 vmirtRead4ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    memAccessAttrs attrs
);
Uns64 vmirtRead8ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    memAccessAttrs attrs
);
```

Description

These four routines read (respectively) 1, 2, 4 and 8 byte data words from the passed address in the passed memory domain. For the 2, 4 and 8 byte versions, the read is performed with the specified endianness.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

If the value is `MEM_AA_TRUE`, then this is a *true processor read* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact read* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

If the value is `MEM_AA_TRUE|MEM_AA_FETCH`, then this is a *true processor fetch* that should generate a call to the processor simulated exception handler on an access violation.

If the value is `MEM_AA_FALSE|MEM_AA_FETCH`, then this is an *artifact fetch* and not a true processor access (usually, an access made by the JIT code generator).

Additionally, the argument controls the value passed as the `processor` argument to read callback functions: see `vmirtAddReadCallback` for more details.

These functions are very useful in intercept libraries since they allow intercept code to be written that is independent of processor endianness (see example).

Example

This example shows how the OVP ARM model uses `vmirtRead4ByteDomain` when accessing translation table entries in memory:

```
static Uns32 transTableRead4(
    armP           arm,
    Uns64          address,
    Uns32         *valueP,
    memAccessAttrs attrs,
    Uns32          level
) {
    // save stage 1 level if this is not a stage 2 access in stage 1 context
    if(!arm->S1PTW) {
        arm->PTWS1Level = level;
    }

    Uns32 faultTypeDomain = transTableTranslateS2(
        arm, &address, MEM_PRIV_R, attrs
    );

    if(!faultTypeDomain) {

        arm->physicalAccess = True;

        memDomainP domain = getTransTablePhysicalDomain(arm);
        memEndian  endian = getTransTableEndian(arm);

        // read value
        Uns32 value = vmirtRead4ByteDomain(domain, address, endian, attrs);

        // return by-ref address
        *valueP = value;

        arm->physicalAccess = False;
    }

    return faultTypeDomain;
}
```

Notes and Restrictions

1. In general, routines from the morph time interface (defined in the *VMI Morph Time Function Reference*) should be used in preference to these functions in processor models wherever possible because they are much faster. As an example, refer see function `vmimtLoadRRO` which generates optimized code to perform a memory read.
2. `memAccessAttrs` value `MEM_AA_IGNORE_PRIV` previously supported by the VMI API is now deprecated and redefined to be equivalent to `MEM_AA_TRUE`.

QuantumLeap Semantics

Synchronizing

10.6 vmirtWrite[1248]ByteDomain

Prototypes

```
void vmirtWrite1ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    Uns8           value,
    memAccessAttrs attrs
);
void vmirtWrite2ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    Uns16          value,
    memAccessAttrs attrs
);
void vmirtWrite4ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    Uns32          value,
    memAccessAttrs attrs
);
void vmirtWrite8ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    Uns64          value,
    memAccessAttrs attrs
);
```

Description

These four routines write (respectively) 1, 2, 4 and 8 byte data words to the passed address in the passed memory domain. For the 2, 4 and 8 byte versions, the write is performed with the specified endianness.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

Value `MEM_AA_FETCH` is used only for read accesses and ignored for writes.

If the value is `MEM_AA_TRUE`, then this is a *true processor write* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact write* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

Additionally, the argument controls the value passed as the `processor` argument to write callback functions: see `vmirtAddWriteCallback` for more details.

These functions are very useful in intercept libraries since they allow intercept code to be written that is independent of processor endianness (see example).

Example

This example shows how the OVP ARM model uses `vmirtRead4ByteDomain` when accessing translation table entries in memory:

```
static Uns32 transTableWrite4(
    armP           arm,
    Uns64          address,
    Uns32          value,
    memAccessAttrs attrs
) {
    Uns32 faultTypeDomain = transTableTranslateS2(
        arm, &address, MEM_PRIV_W, attrs
    );

    if(!faultTypeDomain) {

        arm->physicalAccess = True;

        vmirtWrite4ByteDomain(
            getTransTablePhysicalDomain(arm),
            address,
            getTransTableEndian(arm),
            value,
            attrs
        );

        arm->physicalAccess = False;
    }

    return faultTypeDomain;
}
```

Notes and Restrictions

1. In general, routines from the morph time interface (defined in the *VMI Morph Time Function Reference*) should be used in preference to these functions in processor models wherever possible because they are much faster. As an example, refer see function `vmimtStoreRRO` which generates optimized code to perform a memory write.
2. `memAccessAttrs` value `MEM_AA_IGNORE_PRIV` previously supported by the VMI API is now deprecated and redefined to be equivalent to `MEM_AA_TRUE`.

QuantumLeap Semantics

Synchronizing

10.7 vmirtGetReadNByteSrc

Prototype

```
void *vmirtGetReadNByteSrc(
    memRegionP region,
    Addr       simLow,
    Addr       size,
    Bool       trueAccess
);
```

Description

Given a memory region, this function determines whether the addresses from `simLow` to `simLow+size-1` are represented in that region and are readable using native C functions (for example, `memcpy`). If so, it returns a pointer to the native memory from which the data bytes can be read. If not, it returns `NULL`.

The purpose of this routine is to accelerate memory reads where the same region of memory is read repeatedly. The memory region (represented by the opaque type `memRegionP`) is normally a cache variable in the processor structure. It is updated by a call to `vmirtReadNByteDomain`.

The returned pointer has limited lifetime: it is invalidated by any subsequent call that may modify the simulator's simulated memory mapping. In particular, any call to the following functions will cause invalidation:

```
vmirtReadNByteDomain
vmirtWriteNByteDomain
vmirtGetString
vmirtAliasMemory
vmirtUnaliasMemory
vmirtProtectMemory
vmirtAddReadCallback
vmirtAddWriteCallback
vmirtAddFetchCallback
vmirtRemoveReadCallback
vmirtRemoveWriteCallback
vmirtRemoveFetchCallback
```

The `trueAccess` argument indicates whether this function is being called to model actual processor behavior (`True`) or an artifact of simulation (`False`). For example, if the call is being used to model a block move instruction, the argument should be `True`; if it is being called to move data in order to semi-host some system function, the argument should be `False`. This information is required when the processor model is simulated in conjunction with memory model components such as caches: only transactions corresponding to actual processor behavior should modify the cache state.

Example

```
#include “vmi/vmiRt.h”

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;       // source region cache
    memRegionP dstRegion;       // destination region cache
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src;
    Uns8 *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // unoptimized case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP) x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. If `vmirtGetReadNByteSrc` returns `NULL` it does not imply that memory is not readable: it just means that the address range does not lie in the cached region and a call to the slow read routine `vmirtReadNByteDomain` is required to perform the read. This routine merely improves performance for typical accesses.
2. Models must take great care not to use the pointer returned by this function beyond its valid lifetime. To do so may cause obscure errors or simulator crashes.

QuantumLeap Semantics

Synchronizing

10.8 vmirtGetWriteNByteDst

Prototype

```
void *vmirtGetWriteNByteDst(
    memRegionP region,
    Addr       simLow,
    Addr       size,
    Bool       trueAccess
);
```

Description

Given a memory region, this function determines whether the addresses from `simLow` to `simLow+size-1` are represented in that region and are writable using native C functions (for example, `memcpy`). If so, it returns a pointer to the native memory to which the data bytes can be written. If not, it returns `NULL`.

The purpose of this routine is to accelerate memory reads where the same region of memory is written repeatedly. The memory region (represented by the opaque type `memRegionP`) is normally a cache variable in the processor structure, updated by a call to `vmirtWriteNByteDomain`.

The returned pointer has limited lifetime: it is invalidated by any subsequent call that may modify the simulator's simulated memory mapping. In particular, any call to the following functions will cause invalidation:

```
vmirtReadNByteDomain
vmirtWriteNByteDomain
vmirtGetString
vmirtAliasMemory
vmirtUnaliasMemory
vmirtProtectMemory
vmirtAddReadCallback
vmirtAddWriteCallback
vmirtAddFetchCallback
vmirtRemoveReadCallback
vmirtRemoveWriteCallback
vmirtRemoveFetchCallback
```

The `trueAccess` argument indicates whether this function is being called to model actual processor behavior (`True`) or an artifact of simulation (`False`). For example, if the call is being used to model a block move instruction, the argument should be `True`; if it is being called to move data in order to semi-host some system function, the argument should be `False`. This information is required when the processor model is simulated in conjunction with memory model components such as caches: only transactions corresponding to actual processor behavior should modify the cache state.

Example

```
#include "vmi/vmiRt.h"

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;       // source region cache
    memRegionP dstRegion;       // destination region cache
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src, *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // unoptimized case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. If `vmirtGetWriteNByteDst` returns `NULL` it does not imply that memory is not writable: it just means that the address range does not lie in the cached region and a call to the slow write routine `vmirtWriteNByteDomain` is required to perform the write. This routine merely improves performance for typical accesses.
2. Models must take great care not to use the pointer returned by this function beyond its valid lifetime. To do so may cause obscure errors or simulator crashes.

QuantumLeap Semantics

Synchronizing

10.9 vmirtGetString

Prototype

```
const char *vmirtGetString(memDomainP domain, Addr simAddress);
```

Description

This routine returns a pointer to a `NULL`-terminated string extracted from simulated memory starting at address `simAddress` in the passed memory domain. It is typically used in semihost libraries.

Results are returned by `vmirtGetString` using a cyclic buffer of eight values; on the ninth call, memory allocated for the first result will be reused. Therefore, if string results are required to be persistent after this point, the caller is responsible for allocating a permanent copy.

Example

This example is from the Newlib semihost library for the OR1K processor. The function implements the `open` system call:

```
static VMIOS_INTERCEPT_FN(openOr1k) {

    Uns32 pathnameAddr;
    Uns32 sp;

    // obtain function arguments
    getArg(processor, object, 0, &pathnameAddr);
    vmiosRegRead(processor, object->sp, &sp);

    // get file name from data domain
    memDomainP  domain   = vmirtGetProcessorDataDomain(processor);
    memEndian   endian   = vmirtGetProcessorDataEndian(processor);
    const char *pathname = vmirtGetString(domain, pathnameAddr);

    // get flags & mode from data domain
    Int32 flags = vmirtRead4ByteDomain(domain, sp,   endian, MEM_AA_FALSE);
    Int32 mode  = vmirtRead4ByteDomain(domain, sp+4, endian, MEM_AA_FALSE);

    // implement open
    Int32 vmiosFlags = (flags & 0x003)
                     | ((flags & 0x200) ? VMIOS_O_CREAT : 0)
                     | ((flags & 0x008) ? VMIOS_O_APPEND : 0)
                     | ((flags & 0x400) ? VMIOS_O_TRUNC : 0);

    Int32 result = vmiosOpen(processor, pathname, vmiosFlags, mode);

    if(result != -1) {

        // save file descriptor in simulated descriptor table
        Int32 fdMap = newFileDescriptor(object, context);
        if (fdMap != -1) {
            object->fileDescriptors[fdMap] = result;
        }

        // Return the fileDescriptor (or error)
        result = fdMap;
    }

    // return result
    setErrnoAndResult(processor, object, result, 0);
}
```

Notes and Restrictions

1. If the simulated address passed to `vmirtGetString` is uninitialized, the function will return `NULL`.
2. Apart from the uninitialized memory check, there is no automatic test that the addressed memory contains a valid string: it is up to the application and the model to validate this.

QuantumLeap Semantics

Synchronizing

11 Simulated Memory Management

The `vmiRt.h` interface defines functions to allow simulation of MMUs and related aspects of virtual memory management. This section describes these functions.

A key concept in simulated memory management modeling is the *memory domain*. A memory domain describes an address space and related properties such as access permissions.

At initialization, simulated processors are provided with a physical *code domain* (from where instructions should be fetched) and a physical *data domain* (from which data should be read and written). For many processors, code and data domains are identical. The processor constructor may create various further *virtual domains* as required to efficiently simulate an MMU, MPU or other structures. For example, a processor may create a virtual *kernel domain* and a virtual *user domain*, for simulation of kernel and user mode accesses respectively. Within each virtual domain, regions of memory may be dynamically mapped and unmapped, and also access permissions may be updated. The simulator efficiently validates mappings and access permissions and calls the *memory access exception handlers* defined in the processor attribute structure for invalid accesses.

Memory domains are closely related to code dictionaries, described in the *VMI Morph Time Function Reference*. For a modal processor (for example, one with user and kernel modes) there is typically a memory domain associated with each mode. A processor mode switch will require both the current data and code domain and the current code dictionary to be switched. An outline template showing how to describe a processor with memory and code domains is shown below.

Processor Attributes

The processor attribute structure should provide names for the dictionaries in use by the processor. This name list is a `NULL`-terminated list of strings. For example:

```
// this model has two modes: kernel (mode 0) and user (mode 1).
static const char *dictNames[] = {"KERNEL", "USER", 0};

//
// Configuration block for instruction-accurate modeling
//
const vmiIASAttr cpuxIASAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION & SIZE ATTRIBUTES
    ////////////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,
    .dictNames     = dictNames,
    . . . etc . . .
}
```

Virtual Memory Constructor

The virtual memory constructor is defined with the `VMI_VMINIT_FN` macro and supplied as the `vmInitCB` field of the processor attributes structure. It is passed a pointer to an array of code domains and a pointer to an array of data domains. Each array has the same number of entries as the number of dictionary names in the `dictNames` field provided in the processor attributes structure (in this example, each array will have two entries). By default, all entries in the code array are initialized to the physical code domain derived from the platform and all entries in the data array are initialized to the physical data domain derived from the platform. The constructor may override these defaults with virtual domains if virtual memory management using an MMU or MPU is to be simulated.

```
typedef enum cpuxModeE {
    CPM_KERNEL = 0,
    CPM_USER   = 1
} cpuxMode;

#define PA_LOW_ADDR  0x0000000000ULL
#define PA_HIGH_ADDR 0x1FFFFFFFFFULL
#define KM_ADDR      0x1000000000000ULL

// Processor VM initialization routine. Arguments:
//     processor:   the current processor
//     codeDomains: array of 2 code domains
//     dataDomains: array of 2 data domains
VMI_VMINIT_FN(cpuxVMInit)
{
    cpuxP cpux = (cpuxP)processor;

    memDomainP physicalCodeDomain = codeDomains[0];
    memDomainP physicalDataDomain = dataDomains[0];
    memDomainP kernelDomain;
    memDomainP userDomain;

    ASSERT(
        physicalCodeDomain==physicalDataDomain,
        "expected code & data domains to match"
    );

    // save the physical domain in the processor structure for later use
    cpux->physicalDomain = physicalCodeDomain;

    // define new 56-bit virtual address spaces
    kernelDomain = vmirtNewDomain(“kernel”, 56);
    userDomain   = vmirtNewDomain(“user”    56);

    // set up direct mapped image of physical memory range PA_LOW_ADDR:
    // PA_HIGH_ADDR at virtual address KM_ADDR in physical domain
    vmirtAliasMemory(
        physicalCodeDomain,     // physical domain
        kernelDomain,           // virtual domain
        PA_LOW_ADDR,            // low physical address
        PA_HIGH_ADDR,           // high physical address
        KM_ADDR,                // low virtual address
        0                       // no MRU code management
    );

    // specify the domains to use with this processor
    codeDomains[CPM_KERNEL] = kernelDomain;
    dataDomains[CPM_KERNEL] = kernelDomain;
    codeDomains[CPM_USER]   = userDomain;
    dataDomains[CPM_USER]   = userDomain;
}
```

Mode Switch Instructions

When processing an instruction that causes a mode switch, the translated native code should be constructed to contain an embedded call to a function that does the mode switch. The function will do this by calling `vmirtSetMode`, described in a later section. When the mode is switched, the current processor data and code domains will be updated to the domains with matching index in the `codeDomains` and `dataDomains` arrays. For example, a switch to mode 0 will set the current data domain to `dataDomains[0]` and a switch to mode 1 will set the current data domain to `dataDomains[1]`.

```
static void vmic_SwitchMode(cpuxP cpux, cpuxMode newMode) {

    if(cpux->mode!=newMode) {
        // switch to the new operating mode – this will change
        // dictionary and domain.
        vmirtSetMode((vmiProcessorP)cpux, newMode);
        cpux->mode = newMode;
    }
}
```

Non-Paged and Paged Mappings

There are two distinct types of memory mappings that are established between virtual and physical memory domains: *non-paged* and *paged*.

Non-paged mappings represent translations that are not managed by structures such as TLBs: they are simple aliases of an address range in one domain to an address range in another. They have the following characteristics:

1. The aliases can be of any size, even down to a single byte;
2. They can be hierarchical: a non-paged mapping can be made to an address range in a domain that is itself a non-paged mapping to a range in another domain, and so on;
3. They are relatively static at run time: there is typically some effort in the processor virtual memory constructor to establish the non-paged mappings, which then usually persist for the simulation;
4. They have global scope and are not dependent on processor ASID (*address space id*).

Paged mappings represent virtual memory translations managed by structures such as TLBs. They have the following characteristics:

1. They are typically a multiple of some basic page size (1K, 4K etc);
2. They are not hierarchical: memory mapping cannot be made to a page-mapped region;
3. They are very dynamic at run time: mappings come and go rapidly as the operating system runs;
4. They can have global scope or be limited to a particular processor ASID.

These functions should be used when managing non-paged mappings:

```
vmirtAliasMemory
vmirtUnaliasMemory
```

These functions should be used when managing paged mappings:

```
      vmirtSetProcessorASID
      vmirtGetProcessorASID
      vmirtAliasMemoryVM
      vmirtUnaliasMemoryVM
      vmirtGetDomainMappedASID
      vmirtGetMRUStateTable
      vmirtGetNthStateIndex
```

11.1 vmirtGetProcessorCodeDomain

Prototype

```
memDomainP vmirtGetProcessorCodeDomain(vmiProcessorP processor);
```

Description

This routine returns the current code domain for a processor. The code domain is the source of instruction fetches. This function is generally used in intercept libraries, but can be useful in processor models in debug functions.

Example

This example is from the OVP V850 processor. It shows a debug function that is used to print memory mappings after an MPU update:

```
static void memoryMapping(v850P v850) {

    if(V850_DEBUG_MPU(v850)) {

        vmiPrintf("Code Domain");
        vmirtDebugDomain(vmirtGetProcessorCodeDomain((vmiProcessorP)v850));
        vmiPrintf("Data Domain");
        vmirtDebugDomain(vmirtGetProcessorDataDomain((vmiProcessorP)v850));
        vmiPrintf("\n");
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

11.2 vmirtGetProcessorDataDomain

Prototype

```
memDomainP vmirtGetProcessorDataDomain(vmiProcessorP processor);
```

Description

This routine returns the current data domain for a processor. The data domain is used for load and store instructions specified using the VMI morph-time function API where no domain has been explicitly specified (for example, using function `vmimtLoadRRO`).

Example

This example is from the OVP RISC-V processor. It is used to implement the memory privilege override feature, controlled by `mstatus.MPRV` and `mstatus.MPP`:

```
void riscvVMRefreshMPRVDomain(riscvP riscv) {

    // get current VM enable and mode
    Bool       VM     = RD_CSR_FIELD(riscv, satp, MODE);
    riscvMode  mode   = getCurrentMode(riscv);
    memDomainP domain = 0;

    // if mstatus.MPRV is set, use that mode
    if(getMPRV(riscv)) {

        // get raw value of mstatus.MPP
        riscvMode modeMPP = getMPP(riscv);

        // clamp to implemented mode
        if(!riscvHasMode(riscv, modeMPP)) {
            modeMPP = riscvGetMinMode(riscv);
        }

        mode = modeMPP;
    }

    // look for virtual domain for this mode if required
    if(VM) {
        domain = riscv->vmDomains[mode][0];
    }

    // look for physical domain for this mode if MMU is not enabled or the
    // domain is not VM-managed
    if(!domain) {
        domain = riscv->physDomains[mode][0];
    }

    // switch to the indicated domain if it is not current
    if(domain && (domain!=vmirtGetProcessorDataDomain((vmiProcessorP)riscv))) {
        vmirtSetProcessorDataDomain((vmiProcessorP)riscv, domain);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

11.3 vmirtGetProcessorExternalCodeDomain

Prototype

```
memDomainP vmirtGetProcessorExternalCodeDomain(vmiProcessorP processor);
```

Description

This routine returns the *external* code domain for a processor. The external code domain is the domain associated with the externally-connected instruction bus. Typically, this function will be used in intercept libraries that monitor instruction fetches.

Example

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

Notes and Restrictions

1. In previous versions of the VMI interface, this function was called `vmirtGetProcessorPhysicalCodeDomain`. This name is deprecated and redefined to the new name.

QuantumLeap Semantics

Non-self-synchronizing

11.4 vmirtGetProcessorExternalDataDomain

Prototype

```
memDomainP vmirtGetProcessorExternalDataDomain(vmiProcessorP processor);
```

Description

This routine returns the *external* data domain for a processor. The external data domain is the domain associated with the externally-connected data bus. Typically, this function will be used in intercept libraries that monitor data accesses.

Example

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

Notes and Restrictions

1. In previous versions of the VMI interface, this function was called `vmirtGetProcessorPhysicalDataDomain`. This name is deprecated and redefined to the new name.

QuantumLeap Semantics

Non-self-synchronizing

11.5 vmirtGetProcessorInternalCodeDomain

Prototype

```
memDomainP vmirtGetProcessorInternalCodeDomain(vmiProcessorP processor);
```

Description

This routine returns the *internal* code domain for a processor. The internal code domain is a model-defined memory domain, specified using function `vmirtSetProcessorInternalCodeDomain` in the processor virtual memory constructor.

The purpose of the internal code domain is to allow intercept libraries and platform fetch callbacks to monitor raw physical fetch accesses *before* such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected instruction bus.

Example

This example shows how an intercept library can be written that monitors read, write and fetch accesses on the processor internal data and code domains.

```
#include “vmi/vmiRt.h”

//
// Report a processor access
//
static void report(
    const char   *desc,
    vmiProcessorP processor,
    Uns32         bytes,
    const Uns8   *value
) {
    // report only non-artifact accesses
    if(processor) {

        Uns32 i;

        vmiPrintf("%s: %s [", vmirtProcessorName(processor), desc);

        for(i=0; i<bytes; i++) {
            vmiPrintf("%02x", value[i]);
        }

        vmiPrintf("]\n");
    }
}

//
// Memory callback
//
static VMI_MEM_WATCH_FN(watchCB) {
    report(userData, processor, bytes, value);
}

//
// Constructor
//
VMIOS_CONSTRUCTOR_FN(constructor) {

    // get internal code and data domains
    memDomainP ICDomain = vmirtGetProcessorInternalCodeDomain(processor);
    memDomainP IDDomain = vmirtGetProcessorInternalDataDomain(processor);
    // add internal callbacks
    vmirtAddFetchCallback(ICDomain, processor, 0, -1, watchCB, "IFETCH");
    vmirtAddWriteCallback(IDDomain, processor, 0, -1, watchCB, "IWRITE");
    vmirtAddReadCallback (IDDomain, processor, 0, -1, watchCB, "IREAD ");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

11.6 vmirtGetProcessorInternalDataDomain

Prototype

```
memDomainP vmirtGetProcessorInternalDataDomain(vmiProcessorP processor);
```

Description

This routine returns the *internal* data domain for a processor. The internal data domain is a model-defined memory domain, specified using function `vmirtSetProcessorInternalDataDomain` in the processor virtual memory constructor.

The purpose of the internal data domain is to allow intercept libraries and platform fetch callbacks to monitor raw physical fetch accesses *before* such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected instruction bus.

Example

This example shows how an intercept library can be written that monitors read, write and fetch accesses on the processor internal data and code domains.

```
#include “vmi/vmiRt.h”

//
// Report a processor access
//
static void report(
    const char   *desc,
    vmiProcessorP processor,
    Uns32         bytes,
    const Uns8   *value
) {
    // report only non-artifact accesses
    if(processor) {

        Uns32 i;

        vmiPrintf("%s: %s [", vmirtProcessorName(processor), desc);

        for(i=0; i<bytes; i++) {
            vmiPrintf("%02x", value[i]);
        }

        vmiPrintf("]\n");
    }
}

//
// Memory callback
//
static VMI_MEM_WATCH_FN(watchCB) {
    report(userData, processor, bytes, value);
}

//
// Constructor
//
VMIOS_CONSTRUCTOR_FN(constructor) {

    // get internal code and data domains
    memDomainP ICDomain = vmirtGetProcessorInternalCodeDomain(processor);
    memDomainP IDDomain = vmirtGetProcessorInternalDataDomain(processor);
    // add internal callbacks
    vmirtAddFetchCallback(ICDomain, processor, 0, -1, watchCB, "IFETCH");
    vmirtAddWriteCallback(IDDomain, processor, 0, -1, watchCB, "IWRITE");
    vmirtAddReadCallback (IDDomain, processor, 0, -1, watchCB, "IREAD ");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

11.7 vmirtSetProcessorInternalCodeDomain

Prototype

```
vmirtSetProcessorInternalCodeDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine defines the *internal* code domain for a processor. The internal code domain is a model-defined memory domain that records raw physical instruction fetch accesses before such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected instruction bus. The purpose of this function is to allow a processor model to register a domain that can be used by intercept libraries and platform memory watch callbacks to monitor processor activity. See function `vmirtGetProcessorInternalCodeDomain` for an example use of internal domains in an intercept library.

If `vmirtSetProcessorInternalCodeDomain` is not called in the processor virtual memory constructor, then the domain associated with the externally-connected instruction bus is used as the internal domain also.

Example

This example is extracted from the OVP ARM processor model. The model creates memory domains to model TCM and FCSE structures between the externally connected domains and the internal domains.

```
#include “vmi/vmiRt.h”

VMI_VMINIT_FN(armVMInit) {

    armP       arm           = (armP)processor;
    memDomainP extCodeDomain = codeDomains[0];
    memDomainP extDataDomain = dataDomains[0];

    . . . lines deleted . . .

    // save physical memDomains on processor structure
    arm->ids.external = extCodeDomain;
    arm->ids.postTCM  = postTCMCodeDomain;
    arm->ids.postFCSE = postFCSECodeDomain;
    arm->dds.external = extDataDomain;
    arm->dds.postTCM  = postTCMDataDomain;
    arm->dds.postFCSE = postFCSEDataDomain;

    // set internal code and data domains
    vmirtSetProcessorInternalCodeDomain(processor, postFCSECodeDomain);
    vmirtSetProcessorInternalDataDomain(processor, postFCSEDataDomain);

    . . . lines deleted . . .
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing

11.8 vmirtSetProcessorInternalDataDomain

Prototype

```
vmirtSetProcessorInternalDataDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine defines the *internal* data domain for a processor. The internal data domain is a model-defined memory domain that records raw physical load and store accesses before such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected data bus. The purpose of this function is to allow a processor model to register a domain that can be used by intercept libraries and platform memory watch callbacks to monitor processor activity. See function `vmirtGetProcessorInternalDataDomain` for an example use of internal domains in an intercept library.

If `vmirtSetProcessorInternalDataDomain` is not called in the processor virtual memory constructor, then the domain associated with the externally-connected data bus is used as the internal domain also.

Example

This example is extracted from the OVP ARM processor model. The model creates memory domains to model TCM and FCSE structures between the externally connected domains and the internal domains.

```
#include "vmi/vmiRt.h"

VMI_VMINIT_FN(armVMInit) {

    armP       arm           = (armP)processor;
    memDomainP extCodeDomain = codeDomains[0];
    memDomainP extDataDomain = dataDomains[0];

    . . . lines deleted . . .

    // save physical memDomains on processor structure
    arm->ids.external = extCodeDomain;
    arm->ids.postTCM  = postTCMCodeDomain;
    arm->ids.postFCSE = postFCSECodeDomain;
    arm->dds.external = extDataDomain;
    arm->dds.postTCM  = postTCMDataDomain;
    arm->dds.postFCSE = postFCSEDataDomain;

    // set internal code and data domains
    vmirtSetProcessorInternalCodeDomain(processor, postFCSECodeDomain);
    vmirtSetProcessorInternalDataDomain(processor, postFCSEDataDomain);

    . . . lines deleted . . .
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing

11.9 vmirtSetProcessorCodeDomain

Prototype

```
void vmirtSetProcessorCodeDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine allows the current processor code domain to be set to the passed domain. The new code domain will be used for all instruction fetches until `vmirtSetProcessorCodeDomain` is called again with a different domain or `vmirtSetMode` is called with a different mode (in which case the code domain will be set to the one associated with that mode by the virtual memory constructor or by `vmirtSetProcessorCodeDomains`).

Example

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

Notes and Restrictions

1. If the processor implements virtual memory mappings (created with `vmirtAliasMemoryVM`) then the physical domain associated with old and new domains must be the same.

QuantumLeap Semantics

Non-self-synchronizing

11.10 vmirtSetProcessorDataDomain

Prototype

```
void vmirtSetProcessorDataDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine allows the current processor data domain to be set to the passed domain. The new data domain will be used for all loads and stores until `vmirtSetProcessorDataDomain` is called again with a different domain or `vmirtSetMode` is called with a different mode (in which case the data domain will be set to the one associated with that mode by the virtual memory constructor or by `vmirtSetProcessorDataDomains`).

Example

This example is from the OVP RISC-V processor. It is used to implement the memory privilege override feature, controlled by `mstatus.MPRV` and `mstatus.MPP`:

```
void riscvVMRefreshMPRVDomain(riscvP riscv) {

    // get current VM enable and mode
    Bool       VM     = RD_CSR_FIELD(riscv, satp, MODE);
    riscvMode  mode   = getCurrentMode(riscv);
    memDomainP domain = 0;

    // if mstatus.MPRV is set, use that mode
    if(getMPRV(riscv)) {

        // get raw value of mstatus.MPP
        riscvMode modeMPP = getMPP(riscv);

        // clamp to implemented mode
        if(!riscvHasMode(riscv, modeMPP)) {
            modeMPP = riscvGetMinMode(riscv);
        }

        mode = modeMPP;
    }

    // look for virtual domain for this mode if required
    if(VM) {
        domain = riscv->vmDomains[mode][0];
    }

    // look for physical domain for this mode if MMU is not enabled or the
    // domain is not VM-managed
    if(!domain) {
        domain = riscv->physDomains[mode][0];
    }

    // switch to the indicated domain if it is not current
    if(domain && (domain!=vmirtGetProcessorDataDomain((vmiProcessorP)riscv))) {
        vmirtSetProcessorDataDomain((vmiProcessorP)riscv, domain);
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing

11.11 vmirtSetProcessorCodeDomains

Prototype

```
void vmirtSetProcessorCodeDomains(
    vmiProcessorP processor,
    memDomainPP   domains
);
```

Description

This routine allows the code domains associated with every processor dictionary to be replaced with the passed domains. The `domains` pointer must have one entry for every processor mode. Unlike `vmirtSetProcessorCodeDomain`, the effect of this function is permanent: the new domains will be used with their respective processor modes henceforth, even if the processor switches mode.

Example

This example is taken from the OVP MIPS model. This model supports microthreaded processors that can migrate from one VPE context to another. On each migration, the domains that the TC microthread uses for code and data accesses are modified to reflect the new VPE context.

```
static MTC0_FN(writeTCBind) {

    mipsP tc       = tgt;
    mipsP cpu      = GET_CPU(tgt);
    Uns32 oldValue = COP0_REG(tc, TCBind);
    mipsP newVPE;

    // get the old value of the CurVPE field
    Uns8 oldCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    // update the register, preserving read-only bits
    UPDATE_COP0_MASKED(tc, TCBind, oldValue, newValue, C0_MASK_TCBind);

    // get the new value of the CurVPE field
    Uns8 newCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    if(newCurVPE==oldCurVPE) {

        . . . lines deleted . . .

    } else if(tcCxt && !COP0_FIELD(cpu, MVPControl, VPC)) {

        . . . lines deleted . . .

    } else if((newVPE=findVPE(cpu, newCurVPE))) {

        mipsDisable haltReason    = 0;
        mipsDisable restartReason = 0;

        // reassign to the new parent
        vmirtSetSMPParent((vmiProcessorP)tc, (vmiProcessorP)newVPE);
        tc->vpe = newVPE;

        // processor unable to run if in XTC mode and VPEConf0.XTC doesn't match
        // tcIndex
        if(getEXLERLClear(newVPE) && COP0_FIELD(newVPE, VPEControl, TE)) {
            restartReason |= MD_SUSPEND_XTC;
        } else if(getSMPIndex(tc) == COP0_FIELD(newVPE, VPEConf0, XTC)) {
            restartReason |= MD_SUSPEND_XTC;
        } else {
            haltReason    |= MD_SUSPEND_XTC;
        }

        // processor unable to run if VPEConf0.VPA is clear
        if(!COP0_FIELD(newVPE, VPEConf0, VPA)) {
            haltReason    |= MD_SUSPEND_VPA;
        } else {
            restartReason |= MD_SUSPEND_VPA;
        }

        // update processor halt reason if required
        if(haltReason) {
            haltTC(tc, haltReason);
        }

        // update processor restart reason if required
        if(restartReason) {
            restartTC(tc, restartReason);
        }

        // update blockMask for the new parent context (might not have FPU, for
        // example)
        mipsSetBlockMask(tc);

        // update memory domains
        mipsSetProcessorCodeDomains(tc, newVPE->domains);
        mipsSetProcessorDataDomains(tc, newVPE->domains);

        // Trigger the tcmove event
        vmirtTriggerViewEvent(tc->tcMoveEvent);
    }
}
```

Notes and Restrictions

1. If the processor implements virtual memory mappings (created with `vmirtAliasMemoryVM`) then the physical domain associated with old and new domains must be the same.

QuantumLeap Semantics

Non-self-synchronizing

11.12 vmirtSetProcessorDataDomains

Prototype

```
void vmirtSetProcessorDataDomains(
    vmiProcessorP processor,
    memDomainPP   domains
);
```

Description

This routine allows the data domains associated with every processor dictionary to be replaced with the passed domains. The `domains` pointer must have one entry for every processor mode. Unlike `vmirtSetProcessorDataDomain`, the effect of this function is permanent: the new domains will be used with their respective processor modes henceforth, even if the processor switches mode.

Example

This example is taken from the OVP MIPS model. This model supports microthreaded processors that can migrate from one VPE context to another. On each migration, the domains that the TC microthread uses for code and data accesses are modified to reflect the new VPE context.

```
static MTC0_FN(writeTCBind) {

    mipsP tc        = tgt;
    mipsP cpu       = GET_CPU(tgt);
    Uns32 oldValue = COP0_REG(tc, TCBind);
    mipsP newVPE;

    // get the old value of the CurVPE field
    Uns8 oldCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    // update the register, preserving read-only bits
    UPDATE_COP0_MASKED(tc, TCBind, oldValue, newValue, C0_MASK_TCBind);

    // get the new value of the CurVPE field
    Uns8 newCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    if(newCurVPE==oldCurVPE) {

        . . . lines deleted . . .

    } else if(tcCxt && !COP0_FIELD(cpu, MVPControl, VPC)) {

        . . . lines deleted . . .

    } else if((newVPE=findVPE(cpu, newCurVPE))) {

        mipsDisable haltReason    = 0;
        mipsDisable restartReason = 0;

        // reassign to the new parent
        vmirtSetSMPParent((vmiProcessorP)tc, (vmiProcessorP)newVPE);
        tc->vpe = newVPE;

        // processor unable to run if in XTC mode and VPEConf0.XTC doesn't match
        // tcIndex
        if(getEXLERLClear(newVPE) && COP0_FIELD(newVPE, VPEControl, TE)) {
            restartReason |= MD_SUSPEND_XTC;
        } else if(getSMPIndex(tc) == COP0_FIELD(newVPE, VPEConf0, XTC)) {
            restartReason |= MD_SUSPEND_XTC;
        } else {
            haltReason    |= MD_SUSPEND_XTC;
        }

        // processor unable to run if VPEConf0.VPA is clear
        if(!COP0_FIELD(newVPE, VPEConf0, VPA)) {
            haltReason    |= MD_SUSPEND_VPA;
        } else {
            restartReason |= MD_SUSPEND_VPA;
        }

        // update processor halt reason if required
        if(haltReason) {
            haltTC(tc, haltReason);
        }

        // update processor restart reason if required
        if(restartReason) {
            restartTC(tc, restartReason);
        }

        // update blockMask for the new parent context (might not have FPU, for
        // example)
        mipsSetBlockMask(tc);

        // update memory domains
        mipsSetProcessorCodeDomains(tc, newVPE->domains);
        mipsSetProcessorDataDomains(tc, newVPE->domains);

        // Trigger the tcmove event
        vmirtTriggerViewEvent(tc->tcMoveEvent);
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing

11.13 vmirtGetProcessorCodeEndian

Prototype

```
memEndian vmirtGetProcessorCodeEndian(vmiProcessorP processor);
```

Description

This routine returns the endianness of memory accesses that the processor makes when *fetching instructions*, either `MEM_ENDIAN_BIG` or `MEM_ENDIAN_LITTLE`. This is typically used in intercept libraries

Example

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

Notes and Restrictions

1. Note that in some processors, endianness can be switched at run time.

QuantumLeap Semantics

Non-self-synchronizing

11.14 vmirtGetProcessorDataEndian

Prototype

```
memEndian vmirtGetProcessorDataEndian(vmiProcessorP processor);
```

Description

This routine returns the endianness of memory accesses that the processor makes when *reading or writing values*, either `MEM_ENDIAN_BIG` or `MEM_ENDIAN_LITTLE`. This is typically used in intercept libraries.

Example

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

Notes and Restrictions

1. Note that in some processors, endianness can be switched at run time.
2. In VMI versions prior to 2.0.17, this function was called `vmirtGetProcessorEndian`.

QuantumLeap Semantics

Non-self-synchronizing

11.15 vmirtNewDomain

Prototype

```
memDomainP vmirtNewDomain(const char *name, Uns8 addressBits);
```

Description

This routine creates a new memory domain object with the passed name. The domain has addresses constrained to `addressBits` wide.

This routine will typically be used during processor virtual memory initialization (within the callback of type `vmiVMInitFn` defined by the processor model). The domain will, by default, be associated with the processor passed to the virtual memory initialization function, or with any current processor, if called outside this context: it will have a hierarchical name including that processor name component. It is possible to specify that the domain should instead be associated with a *different* processor – see function `vmirtSetCreateDomainContext`.

Example

This example is taken from the OVP RISC-V model. It shows `vmirtNewDomain` being used to create a physical memory protection (PMP) domain.

```
static memDomainP createDomain(
    riscvMode   mode,
    const char *type,
    Uns32       bits,
    Bool        isCode,
    Bool        unified
) {
    char name[64];

    // fill domain name
    getDomainName(name, mode, type, isCode, unified);

    // create the domain
    return vmirtNewDomain(name, bits);
}

static Bool createPMPDomain(riscvP riscv, riscvMode mode, Bool isCode) {

    memDomainP pmaDomain   = getPMADomainCorD(riscv, mode,  isCode);
    memDomainP otherDomain = getPMADomainCorD(riscv, mode, !isCode);
    Bool       unified     = (pmaDomain==otherDomain);
    memDomainP pmpDomain   = pmaDomain;

    // create PMP domain only if PMP registers are implemented
    if(getNumPMPs(riscv)) {

        Uns32 pmpBits = 64;
        Uns64 extMask = getAddressMask(riscv->extBits);

        // create domain of width pmpBits
        pmpDomain = createDomain(mode, "PMP", pmpBits, isCode, unified);

        // create mapping to external domain with no access privileges
        vmirtAliasMemoryPriv(pmaDomain, pmpDomain, 0, extMask, 0, MEM_PRIV_NONE);
    }

    // save domain
    riscv->pmpDomains[mode][isCode] = pmpDomain;

    return unified;
}
```

Notes and Restrictions

1. Memory domains are restricted to a maximum of 64 bits wide.

QuantumLeap Semantics

Synchronizing

11.16 vmirtSetCreateDomainContext

Prototype

```
void vmirtSetCreateDomainContext(vmiProcessorP processor);
```

Description

This function specifies the processor that should be associated with memory domains created by *subsequent calls to `vmirtNewDomain`*. It is typically used during processor virtual memory initialization (within the callback of type `vmiVMInitFn` defined by the processor model).

By default, calls to `vmirtNewDomain` within the virtual memory initialization function will associate domains with the processor passed as an argument to that function. If virtual memory initialization is being performed for a complex multi-core processor, this is sometimes not what is wanted: for example, it may be required to create domain objects that are shared by a number of cores, and which should therefore be associated with a parent container instead of a leaf processor.

Example

This example is taken from the OVP RISC-V model. In this model, `vmirtSetDomainContext` is used to specify that all domains subsequently created in the constructor should be associated with the current RISC-V core.

```
VMI_VMINIT_FN(riscvVMInit) {

    riscvP     riscv      = (riscvP)processor;
    memDomainP codeDomain = codeDomains[0];
    memDomainP dataDomain = dataDomains[0];
    Bool       unified    = (codeDomain==dataDomain);
    Uns32      codeBits   = vmirtGetDomainAddressBits(codeDomain);
    Uns32      dataBits   = vmirtGetDomainAddressBits(dataDomain);
    riscvMode  mode;

    // use core context for domain creation
    vmirtSetCreateDomainContext(processor);

    // save size of physical domain
    riscv->extBits = (codeBits<dataBits) ? codeBits : dataBits;

    for(mode=RISCV_MODE_SUPERVISOR; mode<RISCV_MODE_LAST; mode++) {

        // create PMP data domain for this mode
        createPMPDomain(riscv, mode, False, unified, dataDomain);

        . . . lines deleted . . .
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

11.17 vmirtAliasMemory

Prototype

```
Bool vmirtAliasMemory(
    memDomainP physicalDomain,
    memDomainP virtualDomain,
    Addr       physicalLowAddr,
    Addr       physicalHighAddr,
    Addr       virtualLowAddr,
    memMRUSetP mruSet
);
```

Description

This routine creates an alias of a region in a *physical domain* so that that physical memory is visible within a *virtual domain*. The region to be aliased has the address range `physicalLowAddr:physicalHighAddr` inclusive in the physical domain. Within the virtual domain, the same memory appears starting at address `virtualLowAddr`. The function returns a Boolean indicating whether the mapping succeeded. The mapping inherits access permissions from the physical domain; use function `vmirtAliasMemoryPriv` if different access privileges are required.

The same physical memory may be mapped at one or more virtual addresses in one or more virtual domains.

The `mruSet` argument is deprecated in this function and should always be 0.

Example

This example is taken from the OVP RISC-V model. It shows `vmirtAliasMemory` being used to create mappings in the physical domain, which is mapped to the PMP domain:

```
static Bool createPhysicalDomain(riscvP riscv, riscvMode mode, Bool isCode) {

    memDomainP pmpDomain   = getPMPDomainCorD(riscv, mode,  isCode);
    memDomainP otherDomain = getPMPDomainCorD(riscv, mode, !isCode);
    Bool       unified     = (pmpDomain==otherDomain);
    Uns32      physBits    = riscvGetXlenArch(riscv);
    Uns64      physMask    = getAddressMask(physBits);

    // create domain of width physBits
    memDomainP physDomain = createDomain(
        mode, "Physical", physBits, isCode, unified
    );

    // create mapping to PMP domain
    vmirtAliasMemory(pmpDomain, physDomain, 0, physMask, 0, 0);

    // save domain
    riscv->physDomains[mode][isCode] = physDomain;

    return unified;
}
```

Notes and Restrictions

1. Access permissions in the new virtual memory region are inherited from the physical memory region; use function `vmirtAliasMemoryPriv` if more restricted access permissions are required.
2. An alias specified with `vmirtAliasMemory` will replace any previously-existing alias for the same virtual address range: there is no need to remove a previous mapping (using `vmirtUnaliasMemory`) first.
3. This function should be used only to establish non-paged mappings: use `vmirtAliasMemoryVM` to establish a paged mapping.

QuantumLeap Semantics

Synchronizing

11.18 vmirtAliasMemoryPriv

Prototype

```
Bool vmirtAliasMemoryPriv(
    memDomainP physicalDomain,
    memDomainP virtualDomain,
    Addr       physicalLowAddr,
    Addr       physicalHighAddr,
    Addr       virtualLowAddr,
    memPriv    privilege
);
```

Description

This routine creates an alias of a region in a *physical domain* so that that physical memory is visible within a *virtual domain*. The region to be aliased has the address range `physicalLowAddr:physicalHighAddr` inclusive in the physical domain. Within the virtual domain, the same memory appears starting at address `virtualLowAddr`. The function returns a Boolean indicating whether the mapping succeeded. The mapping is created with access privileges indicated by argument `privilege`; effective access privilege is the *combined* permission in the two domains.

The same physical memory may be mapped at one or more virtual addresses in one or more virtual domains.

Example

This example is taken from the OVP RISC-V model. It shows `vmirtAliasMemoryPriv` being used to create mappings in a physical memory protection (PMP) domain.

```
static Bool createPMPDomain(riscvP riscv, riscvMode mode, Bool isCode) {

    memDomainP pmaDomain   = getPMADomainCorD(riscv, mode,  isCode);
    memDomainP otherDomain = getPMADomainCorD(riscv, mode, !isCode);
    Bool       unified     = (pmaDomain==otherDomain);
    memDomainP pmpDomain   = pmaDomain;

    // create PMP domain only if PMP registers are implemented
    if(getNumPMPs(riscv)) {

        Uns32 pmpBits = 64;
        Uns64 extMask = getAddressMask(riscv->extBits);

        // create domain of width pmpBits
        pmpDomain = createDomain(mode, "PMP", pmpBits, isCode, unified);

        // create mapping to external domain with no access privileges
        vmirtAliasMemoryPriv(pmaDomain, pmpDomain, 0, extMask, 0, MEM_PRIV_NONE);
    }

    // save domain
    riscv->pmpDomains[mode][isCode] = pmpDomain;

    return unified;
}
```

Notes and Restrictions

1. Access permissions in the new virtual memory region are the combined permissions from the virtual and physical domains. Changing privileges in *either* domain can alter the effective privileges in the virtual domain.
2. An alias specified with `vmirtAliasMemory` will replace any previously-existing alias for the same virtual address range: there is no need to remove a previous mapping (using `vmirtUnaliasMemory`) first.
3. This function should be used only to establish non-paged mappings: use `vmirtAliasMemoryVM` to establish a paged mapping.

QuantumLeap Semantics

Synchronizing

11.19 vmirtUnaliasMemory

Prototype

```
Bool vmirtUnaliasMemory(
    memDomainP virtualDomain,
    Addr       virtualLowAddr,
    Addr       virtualHighAddr
);
```

Description

This routine removes the address range `virtualLowAddr:virtualHighAddr` inclusive from the virtual domain, and returns a boolean indicating whether the unmapping succeeded. The addresses must previously have been mapped by a call to `vmirtAliasMemory` or `vmirtAliasMemoryVM`. Once the address range has been removed from the domain, any attempt to read, write or fetch using addresses in the range will cause the corresponding *memory access exception handler* for the processor model to be called.

Example

This example is taken from the OVP ARMM model. Function vmirtUnaliasMemory is used to remove a mapping when creating memory-mapped SCS region registers:

```
void armSysCreateSCSRegion(armP arm, memDomainP domain) {

    armSCSRegId id;
    SCSRegAddr  scsRegAddrs[SCS_ID(Size)];
    Uns32       prevLow = PPB_LOW;
    Uns32       numRegs = 0;
    Uns32       i;

    // remove any existing PPB region mapping
    vmirtUnaliasMemory(domain, PPB_LOW, PPB_HIGH);

    // install system registers
    for(id=0; id<SCS_ID(Size); id++) {

        // is the system register supported on this variant?
        if(armGetSysRegSupported(id, arm)) {

            const scsRegAttrs *attrs   = &scsRegInfo[id];
            SCSRegAddrP        regAddr = &scsRegAddrs[numRegs++];

            // save register bounds
            regAddr->lowAddr  = attrs->address;
            regAddr->highAddr = regAddr->lowAddr + 3;

            // install callbacks to implement the register
            vmirtMapCallbacks(
                domain, regAddr->lowAddr, regAddr->highAddr, readSys, writeSys,
                (void *)attrs
            );
        }
    }

    // sort register descriptions in ascending address order
    qsort(scsRegAddrs, numRegs, sizeof(scsRegAddrs[0]), compareRegAddr);

    // fill unmapped sections in PPB region with default callback
    for(i=0; i<numRegs; i++) {
        SCSRegAddrP regAddr = &scsRegAddrs[i];

        if(regAddr->lowAddr > prevLow) {
            vmirtMapCallbacks(
                domain, prevLow, regAddr->lowAddr-1, readPPB, writePPB, 0
            );
        }

        prevLow = regAddr->highAddr+1;
    }

    // map final PPB subregion using default callbacks
    if(prevLow<PPB_HIGH) {
        vmirtMapCallbacks(domain, prevLow, PPB_HIGH, readPPB, writePPB, 0);
    }
}
```

Notes and Restrictions

1. There is no requirement for address ranges specified with `vmirtUnaliasMemory` to exactly match ranges previously specified using `vmirtAliasMemory`. For example, it is legal to use `vmirtAliasMemory` to specify a large range and then `vmirtUnaliasMemory` to unmap a “hole” within the large range.
2. If this function is used to remove a mapping created by `vmirtAliasMemoryVM`, the mapping will be removed *irrespective of the ASID which was specified when the mapping was created*. Use function `vmirtUnaliasMemoryVM` to remove a mapping only for a specific ASID.

QuantumLeap Semantics

Synchronizing

11.20 vmirtProtectMemory

Prototype

```
Bool vmirtProtectMemory(
    memDomainP    domain,
    Addr          lowAddr,
    Addr          highAddr,
    memPriv       privilege,
    memPrivAction action
);
```

Description

This routine updates access permissions on the address range `lowAddr:highAddr` inclusive in the memory domain. Privilege values are defined by the type `memPriv` in `vmiTypes.h`:

```
//
// Memory access privilege (bitmask)
//
typedef enum memPrivE {


    // access privilege options
    MEM_PRIV_NONE    = 0x0000,  // no access permitted
    MEM_PRIV_R       = 0x0001,  // read permitted
    MEM_PRIV_W       = 0x0002,  // write permitted
    MEM_PRIV_X       = 0x0004,  // execute permitted
    MEM_PRIV_RW      = (MEM_PRIV_R|MEM_PRIV_W           ),
    MEM_PRIV_RX      = (MEM_PRIV_R|           MEM_PRIV_X),
    MEM_PRIV_WX      = (           MEM_PRIV_W|MEM_PRIV_X),
    MEM_PRIV_RWX     = (MEM_PRIV_R|MEM_PRIV_W|MEM_PRIV_X),

    // alignment constraints
    MEM_PRIV_ALIGN_M = 0x0008,  // check alignment, report memory address
    MEM_PRIV_ALIGN_P = 0x0010,  // check alignment, report access address
    MEM_PRIV_ALIGN   = (MEM_PRIV_ALIGN_M|MEM_PRIV_ALIGN_P),

    // user-defined constraints
    MEM_PRIV_USER1   = 0x0100,  // user-defined restricted access 1
    MEM_PRIV_USER2   = 0x0200,  // user-defined restricted access 2
    MEM_PRIV_USER3   = 0x0400,  // user-defined restricted access 3
    MEM_PRIV_USER4   = 0x0800,  // user-defined restricted access 4
    MEM_PRIV_USER5   = 0x1000,  // user-defined restricted access 5
    MEM_PRIV_USER6   = 0x2000,  // user-defined restricted access 6
    MEM_PRIV_USER7   = 0x4000,  // user-defined restricted access 7
    MEM_PRIV_USER8   = 0x8000,  // user-defined restricted access 8
    MEM_PRIV_USER    = (MEM_PRIV_USER1|MEM_PRIV_USER2|
                        MEM_PRIV_USER3|MEM_PRIV_USER4|
                        MEM_PRIV_USER5|MEM_PRIV_USER6|
                        MEM_PRIV_USER7|MEM_PRIV_USER8),

    // legacy user-defined constraint
    MEM_PRIV_DEVICE  = MEM_PRIV_USER1,

} memPriv;
```

Values `MEM_PRIV_R`, `MEM_PRIV_W` and `MEM_PRIV_X` specify read, write and execute privilege, respectively; compound privileges `MEM_PRIV_RW`, `MEM_PRIV_RX`, `MEM_PRIV_WX` and `MEM_PRIV_RWX` specify combinations of these basic privileges.

Values `MEM_PRIV_ALIGN_M` and `MEM_PRIV_ALIGN_P` specify that accesses to the address range must be *aligned to the access size*; this is required by some processors (e.g. ARM processors with a TLB) in which misaligned accesses are not allowed to regions of memory with certain properties. If `MEM_PRIV_ALIGN_M` is specified, then on an unaligned access failure the processor alignment handler will be passed *the lowest address that was indicated as requiring alignment*. If `MEM_PRIV_ALIGN_P` is specified and `MEM_PRIV_ALIGN_M` is *not* specified, then on an unaligned access failure the processor alignment handler will be passed the lowest address of the *unaligned access*, even if that address was not specified to require aligned access. The addresses passed to the handler differ only in the case that an unaligned access is made that straddles the boundary between a lower-address region that *does not* require aligned access and a higher-address region that *does* require aligned access.

Values `MEM_PRIV_USER1` to `MEM_PRIV_USER8` specify that this region of memory has user-defined restricted access permissions: see the *Imperas Morph Time Function Reference* for more information about how user-defined region access is constrained. Value `MEM_PRIV_DEVICE` is present for legacy reasons and equivalent to `MEM_PRIV_USER1`.

The `action` argument (also from `vmiTypes.h`) defines how the supplied privilege should be used:

```
//
// How privilege mask is to be applied
//
typedef enum memPrivActionE {
    MEM_PRIV_SET,                   // newPriv = privilege
    MEM_PRIV_ADD,                   // newPriv = oldPriv | privilege
    MEM_PRIV_SUB                    // newPriv = oldPriv & ~privilege
} memPrivAction;
```

Any attempt by a simulated program to access memory in a way which is not permitted will cause one of the memory exception handlers in the `vmiAttrs` attribute structure to be called:

```
vmiRdWrSnapFn  rdSnapCB;            // read alignment snap function
vmiRdWrSnapFn  wrSnapCB;            // write alignment snap function
. . .
vmiRdPrivExceptFn  rdPrivExceptCB;  // read privilege exception
vmiWrPrivExceptFn  wrPrivExceptCB;  // write privilege exception
vmiRdAlignExceptFn rdAlignExceptCB; // read alignment exception
vmiWrAlignExceptFn wrAlignExceptCB; // write alignment exception
. . .
vmiIFetchFn        ifetchExceptCB;  // execute privilege exception
```

See the *OVP Processor Modeling Guide* for a detailed description of address snap and exception handlers.

Example

The OVP ARC model uses this function when code protection registers are updated:

```
void arcVMUpdateCodeProtection(arcP arc) {

    memDomainP domain     = arc->codeProtectionDomain;
    Uns32      regionSize = arc->msbAddrMask >> 3;
    Uns32      mask       = arc->codeProtectionMask;
    Uns32      lowAddr    = 0;
    Uns32      i;

    for(i=0; i<16; i++) {

        Uns32         highAddr = lowAddr+regionSize-1;
        Bool          protect  = mask & 1;
        memPrivAction action   = protect ? MEM_PRIV_SUB : MEM_PRIV_ADD;

        // update region protections
        vmirtProtectMemory(domain, lowAddr, highAddr, MEM_PRIV_RW, action);

        // step to the next region
        lowAddr += regionSize;
        mask >>= 1;
    }
}
```

Notes and Restrictions

1. If a no memory at all is defined at a particular physical address range, then any attempt to change access permissions for that physical address range will have no effect – such *void* regions always have no read, write or execute permission.
2. If an address range in a virtual memory domain is unmapped (no physical equivalent has been specified by `vmirtAliasMemory` or `vmirtAliasMemoryVM`) then any attempt to change access permissions for that virtual address range will have no effect – such *void* regions always have no read, write or execute permission.

QuantumLeap Semantics

Synchronizing

11.21 vmirtSetLoadStoreMask

Prototype

```
void vmirtSetLoadStoreMask(
    memDomainP   domain,
    Uns32        topBit,
    memLSMAction zeroAction,
    memLSMAction oneAction
);
```

Description

This routine causes the address used when a load or store is made to the domain to be modified so that all bits above the indicated top address bit are either set to zero, set to one, or sign extended from the next lower bit. Argument `topBit` specifies the index of the most-significant unmodified bit. Argument `zeroAction` specifies the required action if this bit is zero; argument `oneAction` specifies the required action if this bit is one. The action is defined by the type `memLSMAction` in `vmiTypes.h`:

```
typedef enum memLSMActionE {
    MEM_LSM_NONE = 0,   // no action
    MEM_LSM_0    = 1,   // fill masked section of the address with 0
    MEM_LSM_1    = 2    // fill masked section of the address with 1
} memLSMAction;
```

Value `MEM_LSM_NONE` indicates that the address should be used unmodified. Value `MEM_LSM_0` indicates that all bits above the top bit should be filled with zero. Value `MEM_LSM_1` indicates that all bits above the top bit should be filled with one.

Example

The OVP ARM model uses this function to implement top-byte-ignored (TBI) behavior:

```
static void setAA64LoadStoreMask(
    armDomainSetP domainSet,
    Bool          AA64,
    Bool          NS,
    armEL         EL,
    memLSMAction  action0,
    memLSMAction  action1
) {
    memDomainP domain;

    if((domain=domainSet->mapped[AA64][EL][NS])) {
        vmirtSetLoadStoreMask(domain, 55, action0, action1);
    }
    if((domain=domainSet->unmapped[AA64][EL][NS])) {
        vmirtSetLoadStoreMask(domain, 55, action0, action1);
    }
}
```

Notes and Restrictions

1. This function affects loads and stores only. For fetches, see function `vmimtSetAddressMask` in the *VMI Morph Time Function Reference* manual.
2. If both `zeroAction` and `oneAction` are `MEM_LSM_NONE` then no address masking is done on loads or stores; in this case, the value of `topBit` is ignored.

QuantumLeap Semantics
Synchronizing

11.22 vmirtGetDomainAddressBits

Prototype

```
Uns8 vmirtGetDomainAddressBits(memDomainP domain);
```

Description

This routine returns the width of the domain (the number of significant bits with which that domain can be read or written).

The returned value is <= 64.

Example

The OVP RISC-V model uses this function the virtual memory constructor to obtain the size of the connected physical bus.

```
VMI_VMINIT_FN(riscvVMInit) {

    riscvP     riscv      = (riscvP)processor;
    memDomainP codeDomain = codeDomains[0];
    memDomainP dataDomain = dataDomains[0];
    Bool       unified    = (codeDomain==dataDomain);
    Uns32      codeBits   = vmirtGetDomainAddressBits(codeDomain);
    Uns32      dataBits   = vmirtGetDomainAddressBits(dataDomain);
    riscvMode  mode;

    // use core context for domain creation
    vmirtSetCreateDomainContext(processor);

    // save size of physical domain
    riscv->extBits = (codeBits<dataBits) ? codeBits : dataBits;

    . . . lines deleted . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

11.23 vmirtGetDomainPrivileges

Prototype

```
memPriv vmirtGetDomainPrivileges(memDomainP domain, Addr simAddr);
```

Description

Given an address in a memory domain, this function returns the access privileges currently applicable to that address. The returned value is defined by an enum in `vmiTypes.h`:

```
//
// Memory access privilege (bitmask)
//
typedef enum memPrivE {


    // access privilege options
    MEM_PRIV_NONE    = 0x0000,  // no access permitted
    MEM_PRIV_R       = 0x0001,  // read permitted
    MEM_PRIV_W       = 0x0002,  // write permitted
    MEM_PRIV_X       = 0x0004,  // execute permitted
    MEM_PRIV_RW      = (MEM_PRIV_R|MEM_PRIV_W           ),
    MEM_PRIV_RX      = (MEM_PRIV_R|           MEM_PRIV_X),
    MEM_PRIV_WX      = (           MEM_PRIV_W|MEM_PRIV_X),
    MEM_PRIV_RWX     = (MEM_PRIV_R|MEM_PRIV_W|MEM_PRIV_X),

    // alignment constraints
    MEM_PRIV_ALIGN_M = 0x0008,  // check alignment, report memory address
    MEM_PRIV_ALIGN_P = 0x0010,  // check alignment, report access address
    MEM_PRIV_ALIGN   = (MEM_PRIV_ALIGN_M|MEM_PRIV_ALIGN_P),

    // user-defined constraints
    MEM_PRIV_USER1   = 0x0100,  // user-defined restricted access 1
    MEM_PRIV_USER2   = 0x0200,  // user-defined restricted access 2
    MEM_PRIV_USER3   = 0x0400,  // user-defined restricted access 3
    MEM_PRIV_USER4   = 0x0800,  // user-defined restricted access 4
    MEM_PRIV_USER5   = 0x1000,  // user-defined restricted access 5
    MEM_PRIV_USER6   = 0x2000,  // user-defined restricted access 6
    MEM_PRIV_USER7   = 0x4000,  // user-defined restricted access 7
    MEM_PRIV_USER8   = 0x8000,  // user-defined restricted access 8
    MEM_PRIV_USER    = (MEM_PRIV_USER1|MEM_PRIV_USER2|
                        MEM_PRIV_USER3|MEM_PRIV_USER4|
                        MEM_PRIV_USER5|MEM_PRIV_USER6|
                        MEM_PRIV_USER7|MEM_PRIV_USER8),

    // legacy user-defined constraint
    MEM_PRIV_DEVICE  = MEM_PRIV_USER1,

} memPriv;
```

Values `MEM_PRIV_R`, `MEM_PRIV_W` and `MEM_PRIV_X` specify read, write and execute privilege, respectively; compound privileges `MEM_PRIV_RW`, `MEM_PRIV_RX`, `MEM_PRIV_WX` and `MEM_PRIV_RWX` specify combinations of these basic privileges.

Values `MEM_PRIV_ALIGN_M` and `MEM_PRIV_ALIGN_P` specify that accesses to the address range must be *aligned to the access size*; this is required by some processors (e.g. ARM processors with a TLB) in which misaligned accesses are not allowed to regions of memory with certain properties. If `MEM_PRIV_ALIGN_M` is specified, then on an unaligned access failure the processor alignment handler will be passed *the lowest address that was* *indicated as requiring alignment*. If `MEM_PRIV_ALIGN_P` is specified and `MEM_PRIV_ALIGN_M` is *not* specified, then on an unaligned access failure the processor alignment handler will be passed the lowest address of the *unaligned access*, even if that address was not specified to require aligned access. The addresses passed to the handler differ only in the case that an unaligned access is made that straddles the boundary between a lower-address region that *does not* require aligned access and a higher-address region that *does* require aligned access.

Values `MEM_PRIV_USER1` to `MEM_PRIV_USER8` specify that this region of memory has user-defined restricted access permissions: see the *Imperas Morph Time Function Reference* for more information about how user-defined region access is constrained. Value `MEM_PRIV_DEVICE` is present for legacy reasons and equivalent to `MEM_PRIV_USER1`.

Example

The ARM processor model example uses this function in the virtual memory support module to validate DMA region accessibility:

```
static Bool validateAddressDMA(
    armP        arm,
    memDomainP  virtualDomain,
    armDMAUnitP unit,
    Uns32       address,
    memPriv     priv
) {
    Bool isExternal = unit->isExternal;
    Bool inTCM      = addressInTCM(arm, virtualDomain, unit, address, priv);

    if(!(vmirtGetDomainPrivileges(virtualDomain, address) & priv)) {

        // no access at the required address (access permission failures are
        // higher priority than TCM address failures)
        return True;

    } else if(inTCM==isExternal) {

        . . .
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

11.24 vmirtIsExecutable

Prototype

```
Bool vmirtIsExecutable(vmiProcessorP processor, Addr simPC);
```

Description

Given an address, this function returns a boolean indicating if that address is executable in the current processor code domain. It is typically used in processor fetch exception handlers.

Example

The OR1K example model uses `vmirtIsExecutable` to determine whether execute privilege exceptions should be taken:

```
VMI_IFETCH_FN(or1kIFetchExceptionCB) {

    or1kP or1k = (or1kP)processor;

    if(or1k->reset) {

        // force a reset
        or1k->reset = False;
        or1kTakeException(or1k, RST_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(takeIEE(or1k)) {

        // external interrupt must be taken
        or1kTakeException(or1k, EXI_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(takeTEE(or1k)) {

        // tick timer interrupt must be taken
        or1kTakeException(or1k, TTI_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(address & 3) {

        // handle misaligned fetch exception
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, BUS_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(!vmirtIsExecutable(processor, address)) {

        // handle execute privilege exception
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, IPF_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else {

        // no fetch exception
        return VMI_FETCH_NONE;
    }
}
```

Notes and Restrictions

1. See also `vmirtGetDomainPrivileges` that allows domain-based privilege checks.

QuantumLeap Semantics

Synchronizing

11.25 vmirtIsAlias

Prototype

```
Bool vmirtIsAlias(memDomainP domain, Addr simAddr);
```

Description

Given an address `simAddr`, this function returns a Boolean indicating whether that address is an *alias* in the given domain – in other words, whether it lies in a region mapped as the `virtualDomain` of a call to `vmirtAliasMemory` or `vmirtAliasMemoryVM`.

This function is primarily useful for implementing *on-demand* region mapping: a domain object can be created in an initial unmapped state, and then have mappings applied as required for limited address ranges within read, write or fetch exception callbacks. This can improve simulation performance when immediate creation of the required aliases would otherwise result in a large number of region aliases of a single master region. This will typically only occur for highly-parallel systems where tens or hundreds of processors share parts of a single address space.

Example

The OVP RISC-V model uses this function to implement lazy mapping from a processor PMA bus to the external data or code bus. In this model, these mappings are created on-demand in pages of size specified by the `PMP_max_page` model parameter:

```
static void mapPMA(
    riscvP    riscv,
    riscvMode mode,
    memPriv   requiredPriv,
    Uns64     lowPA,
    Uns64     highPA
) {
    Uns64 clamp = riscv->configInfo.PMP_max_page;

    // handle lazy PMA region mapping if required
    if(clamp) {

        memDomainP extD    = riscv->extDomains[False];
        memDomainP extC    = riscv->extDomains[True];
        memDomainP pmaD    = getPMADomainCorD(riscv, mode, False);
        memDomainP pmaC    = getPMADomainCorD(riscv, mode, True);
        Uns64      extMask = getAddressMask(riscv->extBits);
        Uns64      mask    = clamp-1;
        Uns64      startPA = (lowPA  & ~mask);
        Uns64      endPA   = (highPA & ~mask) + clamp;
        Bool       doPMA   = False;
        Uns64      PA;

        for(PA=startPA; (PA<extMask) && (PA!=endPA); PA+=clamp) {

            // map PMA data page if required
            if(!vmirtIsAlias(pmaD, PA)) {
                aliasPMA(riscv, extD, pmaD, PA, PA+mask, doPMA);
            }

            // map PMA code page if required
            if((pmaD!=pmaC) && !vmirtIsAlias(pmaC, PA)) {
                aliasPMA(riscv, extC, pmaC, PA, PA+mask, doPMA);
            }
        }
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

11.26 vmirtGetDomainMapped

Prototype

```
memMapped vmirtGetDomainMapped(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr
);
```

Description

Given an address range `lowAddr:highAddr`, this function returns a member of the enumeration `memMapped` that indicates whether the address range is mapped in the domain:

```
typedef enum memMappedE {
    MEM_MAP_NONE = 0,               // no addresses in the range mapped
    MEM_MAP_PART = 1,               // some addresses in the range mapped
    MEM_MAP_FULL = 2,               // all addresses in the range mapped
} memMapped;
```

If no address in the range is accessible[7], the function returns `MEM_MAP_NONE`; if all addresses in the range are accessible, the function returns `MEM_MAP_FULL`; otherwise, the function returns `MEM_MAP_PART`.

Example

The OVP RISC-V model uses this function to determine whether addresses on the CSR external bus are implemented:

```
static Bool csrImplementExternal(riscvCSRId id, riscvP riscv, memPriv priv) {

    memDomainP domain = riscvGetExternalCSRDomain(riscv);

    if(riscv->externalActive) {

        // don't nest external access calls (assume that a nested read or write
        // should instead access the register inside the model)
        return False;

    } else if(domain) {

        Uns32 address = getCSRBusAddress(id);

        if(!vmirtGetDomainMapped(domain, address, address)) {
            return False;
        } else {
            return (vmirtGetDomainPrivileges(domain, address) & priv) && True;
        }

    } else {

        return False;
    }
}
```

[7] *Accessible* is defined as having one or more of read, write or execute permissions.

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

11.27 vmirtGetNextMappedRange

Prototype

```
Bool vmirtGetNextMappedRange(
    memDomainP domain,
    Addr       *lowAddrP,
    Addr       *highAddrP
);
```

Description

This function searches the address range `*lowAddrP:*highAddrP` to find the first *mapped memory range* in that range. A mapped memory range is one for which simulated memory has been allocated, typically because it has been read or written by a processor. This function will typically be used by an intercept library that wishes to save memory contents. It enables traversal of an address space to find only those regions which have been updated during a simulation.

Example

This example shows usage of this function in an intercept library. At the end of simulation, the destructor examines the processor data domain to determine regions that have been used during that simulation:

```
VMIOS_DESTRUCTOR_FN(destructor) {

    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    Addr       low    = 0;
    Addr       high   = -1;
    Bool       done   = False;

    while(!done && vmirtGetNextMappedRange(domain, &low, &high)) {

        vmiPrintf("Mapped 0x"FMT_Ax":"FMT_Ax"\n", low, high);

        low  = high+1;
        high = -1;
        done = !low;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

11.28 vmirtMapVAToPA

Prototype

```
Addr vmirtMapVAToPA(
    memDomainP  virtualDomain,
    Addr        virtualAddr,
    memDomaonPP physicalDomain,
    memRegionPP cachedRegion
);
```

Description

Given a virtual domain and a virtual address, this function returns the *physical address* and *physical domain* to which that virtual address corresponds (in other words, the address and domain derived from any mapping set up for this virtual address using a previous call to `vmirtAliasMemory` or `vmirtAliasMemoryVM`).

The `cachedRegion` argument, if non-null, should be a pointer to a region cache that can be used by the function to accelerate the lookup process (it saves the result of any previous translation at this location and tries this mapping result first on any subsequent call).

This function is typically used in cache and MMU models.

Example

The OVP ARM model uses this function to determine whether a virtual address lies within a TCM:

```
static Bool addressInTCM(
    armP        arm,
    memDomainP  virtualDomain,
    armDMAUnitP unit,
    Uns64       address,
    memPriv     priv
) {
    memPriv       privTCM = unit->DMAControl.fields.TR ? MEM_PRIV_X : MEM_PRIV_R;
    armDomainSetP set     = getDomainSetPriv(arm, privTCM);
    memDomainP    physicalDomain;
    Uns32         i;

    // establish virtual mapping if required (note that this will reinstate
    // any permissions removed by removePermissionsCurrent)
    if(!vmirtGetDomainPrivileges(virtualDomain, address)) {
        armVMMiss(arm, virtualDomain, priv, address, 1, True);
    }

    // get physical domain for any currently-established mapping
    vmirtMapVAToPA(virtualDomain, address, &physicalDomain, 0);

    // determine whether the address is in any TCM domain
    for(i=0; i<set->tcmNum; i++) {
        if(physicalDomain==set->tcms[i].port->vmiPort.domain) {
            return True;
        }
    }

    // not a TCM address
    return False;
}
```

Notes and Restrictions

1. If there is no valid mapping for the virtual address, then the original address is returned and the `physicalDomain` by-ref argument set to `NULL`.
2. In the case that a virtual mapping has been established to an address that is itself virtually mapped, the address and domain returned will be that of the ultimate, non-virtually-mapped address.

QuantumLeap Semantics

Synchronizing

11.29 vmirtMapMemory

Prototype

```
Bool vmirtMapMemory(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr,
    memType    type
);
```

Description

Function `vmirtMapMemory` specifies that a the address range `lowAddr:highAddr` in the memory domain should be mapped either as ROM (if `type` is `MEM_ROM`) or as RAM (if `type` is `MEM_RAM`). The backing store used for the address range is managed entirely by the simulator.

This function is most often used to populate memory domains created by `vmirtNewDomain` which are intended to be used to model (for example) register banks or local processor memories such as boot ROMs.

Example

The OVP ARM model uses this function to populate TCM memory buses connected to the processor via ports:

```
static void initializeTCMs(armP arm, armDomainSetP domainSet) {

    armTCMInfoP tcms = domainSet->tcms;
    Uns32       i;

    for(i=0; i<domainSet->tcmNum; i++) {

        armTCMInfoP tcm      = &tcms[i];
        armBusPortP tcmPort  = tcm->port;
        const char *portName = tcmPort->vmiPort.name;

        // allocate default domain if port is unconnected externally
        if(!tcmPort->vmiPort.domain) {
            char tmpName[256];
            sprintf(tmpName, "%s internal", portName);
            tcmPort->vmiPort.domain = newDomain(tmpName, 32);
        }

        // assume domain is externally mapped if address 0 is mapped
        Bool mappedExternal = vmirtGetDomainMapped(tcmPort->vmiPort.domain, 0, 0);

        // create default internal TCM domain if required, mapped to RAM
        if(arm->configInfo.useInternalTCMs || !mappedExternal) {
            vmirtMapMemory(tcmPort->vmiPort.domain, 0, 0xffffffff, MEM_RAM);
            tcm->saveRestore = True;
        }
    }
}
```

Notes and Restrictions

1. Only memory that is not currently mapped may be mapped using `vmirtMapMemory`.

2. ROM memories are not writable by processors but can be written by `vmirtWriteNByteDomain` etc if the `trueAccess` parameter is `False`. This allows a ROM to be initialized.

QuantumLeap Semantics

Synchronizing

11.30 vmirtMapCallbacks

Prototype

```
Bool vmirtMapCallbacks(
    memDomainP    domain,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemReadFn  readCB,
    vmiMemWriteFn writeCB,
    void         *userData
);
```

Description

Function `vmirtMapCallbacks` specifies that a the address range `lowAddr:highAddr` in the memory domain should be mapped using the passed `readCB` and `writeCB` callback functions. Either callback may be omitted. If both are omitted, the function is equivalent to a call to `vmirtMapMemory` with memory type `MEM_RAM`.

This function is most often used to populate memory domains created by `vmirtNewDomain` which are intended to be used to model callback-activated devices.

The type `vmiMemReadFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_READ_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    Addr           address,     \
    Uns32          bytes,       \
    void          *value        \
    void          *userData,    \
    Addr           VA,          \
    Bool           isFetch,     \
    memAccessAttrs attrs        \
)
typedef VMI_MEM_READ_FN((*vmiMemReadFn));
```

When called, the `readCB` is passed the following arguments:

1. `processor`: the processor performing the read, or `NULL` if there is no processor context.
2. `address`: the *physical* address of the first byte being read.
3. `bytes`: a count of the bytes being read (typically 1, 2, 4 or 8).
4. `value`: a pointer to a buffer that should be filled by the callback with the result of the read.
5. `userData`: model-specific data, supplied as the last argument to `vmirtMapCallbacks`.
6. `VA`: the *virtual* address of the first byte being read.
7. `isFetch`: whether this read is the result of an instruction fetch.
8. `attrs`: the attributes of the read, which is a member of the `memAccessAttrs` enumeration, described below (*new from VMI version 4.2.0*).

The type `vmiMemWriteFn` is also defined in `vmiTypes.h`:

```
#define VMI_MEM_WRITE_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    Addr           address,     \
    Uns32          bytes,       \
    const void    *value,       \
    void          *userData,    \
    Addr           VA,          \
    memAccessAttrs attrs        \
)
typedef VMI_MEM_WRITE_FN((*vmiMemWriteFn));
```

When called, the `writeCB` is passed the following arguments:

1. `processor`: the processor performing the read, or `NULL` if there is no processor context.
2. `address`: the *physical* address of the first byte being written.
3. `bytes`: a count of the bytes being written (typically 1, 2, 4 or 8).
4. `value`: a pointer to a buffer containing the bytes being written.
5. `userData`: model-specific data, supplied as the last argument to `vmirtMapCallbacks`.
6. `VA`: the *virtual* address of the first byte being written.
7. `attrs`: the attributes of the read, which is a member of the `memAccessAttrs` enumeration, described below (*new from VMI version 4.2.0*).

The type `vmiMemAccessAttrs` is defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

The read and write callbacks can be invoked in one of three state combinations, defined by the processor and attrs arguments as follows:

1. `processor=NULL`, `attrs=MEM_AA_FALSE`: this combination indicates an artifact read/write (e.g. from a debugger) with no processor context.
2. `processor=non-NULL`, `attrs=MEM_AA_FALSE`: this combination indicates an artifact read/write (e.g. from a debugger) with a processor context. This will typically be an access using `opProcessorRead` or `opProcessorWrite`, or legacy `icmDebugReadProcessorMemory` or `icmDebugWriteProcessorMemory` in the platform.
3. `processor=non-NULL`, `attrs==MEM_AA_TRUE`: this combination indicates a true processor read/write access.
4. `attrs=MEM_AA_FALSE|MEM_AA_FETCH`: this combination indicates an artifact fetch (usually, by the JIT code generator) with processor context.
5. `attrs=MEM_AA_TRUE|MEM_AA_FETCH`: this combination indicates an true processor fetch.

Example

This example is taken from the OVP MIPS processor model. It creates the memory mapped Global Configuration Registers (GCRs) for a 1004K processor.

```
static mipsGCRGlobalP allocGCRGlobals(mipsP tc, memDomainP physicalDomain) {
    mipsP          vpe        = VPE_FOR_TC(tc);
    mipsGCRGlobalP gcrGlobals = vpe->gcrGlobals;

    if(!gcrGlobals) {

        mipsP cpu = CPU_FOR_VPE(vpe);

        gcrGlobals = cpu->gcrGlobals;

        if(!gcrGlobals) {

            mipsP cmp = CMP_FOR_CPU(cpu);

            gcrGlobals = cmp->gcrGlobals;

            if(!gcrGlobals) {

                gcrGlobals = TYPE_CALLOC(mipsGCRGlobal);

                // link gcrGlobals to CMP
                . . . code omitted for clarity . . .

                // create GCR and CMP domains
                memDomainP gcrDomain = vmirtNewDomain("GCR", 32);
                memDomainP cmpDomain = vmirtNewDomain("CMP", VM_BITS);

                // implement the GCR region using callbacks
                vmirtMapCallbacks(
                    gcrDomain, 0, GCB_SIZE-1, readGCR, writeGCR, gcrGlobals
                );

                . . . code omitted for clarity . . .
}
```

Notes and Restrictions

1. Only memory that is not currently mapped may be mapped natively using `vmirtMapCallbacks`.

QuantumLeap Semantics

Synchronizing

11.31 vmirtMapNativeMemory

Prototype

```
Bool vmirtMapNativeMemory(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr,
    void      *memory
);
```

Description

The simulator usually maintains the backing store used for simulated memory contents itself – memory pages are allocated on demand when required by loads and stores from previously-unmapped regions, and might be relocated if merging of adjacent memory regions occurs.

Sometimes, it may be required that specific address ranges are not managed by the simulator in this way but should instead be mapped to specific native host memory buffers. As an example, it might be that it is more convenient to maintain the contents of a frame buffer at a fixed native address so that it can easily be processed by a renderer implemented using a PSE[8] or by C code in an OP or legacy ICM platform.

Function `vmirtMapNativeMemory` specifies that a native memory buffer specified as the `memory` argument should be used to hold the contents of the address range `lowAddr:highAddr` in the memory domain.

Example

```
#define FRAME_BUF_SIZE 0x100000
static Uns8 frameBuffer[FRAME_BUF_SIZE];

static void vmic_MapFrameBuffer(cpuxP cpux, Uns32 fbLow) {

    memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);
    Uns32      fbHigh = fbLow+FRAME_BUF_SIZE-1;

    // use native buffer ‘frameBuffer’ as backing store for the read buffer
    // of the passed processor
    vmirtMapNativeMemory(domain, fbLow, fbHigh, frameBuffer);
}
```

Notes and Restrictions

1. Be careful to ensure the native memory is large enough to cover the required address range – the simulator does not check this.
2. Only memory that is not currently mapped may be mapped natively using `vmirtMapNativeMemory`.

QuantumLeap Semantics

Synchronizing

[8] A PSE is a *peripheral simulation engine*, used to implement peripheral models in the Imperas environment.

11.32 vmirtAddReadCallback

Prototype

```
void vmirtAddReadCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr           lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine installs a model-specific *read watch point function*, `watchCB`, on the address range `lowAddr:highAddr` in `domain`. The callback will be invoked whenever any address in the range is read during simulation.

Prior to VMI version 7.55.0, the watch point function was called for true loads from the address range only. From VMI version 7.55.0 onwards, the watch point function is also called for try-load accesses (initiated by `vmimtTryLoadRC`, for example). The value argument is non-`NULL` for a load and `NULL` for a try-load.

If `scope` is `NULL`, the callback will be activated on any access to the memory range by *any processor* using any address alias. If scope is non-`NULL` and not the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the *given processor* accesses the memory range using any address alias. If `scope` is the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the domain is accessed by a *master processor* of the domain. A processor is a master processor of a domain if it directly accesses the domain, or accesses a domain that has been aliased to the domain using functions `vmirtAliasMemory` or `vmirtAliasMemoryVM` at any level of indirection.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr           address,      \
    Uns32          bytes,        \
    const void    *value,        \
    void          *userData,     \
    Addr           VA            \
)
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

When called, the `watchCB` is passed the following arguments:

1. `processor`: the processor performing the read, or `NULL` if this is an artifact read (for example, resulting from a call to `vmirtReadNByteDomain` with the `attrs== MEM_AA_FALSE`, or an access by the simulator core).
2. `address`: the *physical* address of the first byte being read or written.
3. `bytes`: a count of the bytes being read (typically 1, 2, 4 or 8).

4. `value`: a pointer to a buffer containing the bytes being read, or `NULL` for a try-load access.
5. `userData`: model-specific data, supplied as the first argument to `vmirtAddReadCallback`.
6. `VA`: the *virtual* address of the first byte being read[9].

Example

```
static VMI_MEM_WATCH_FN(trapRead) {
    vmiPrintf(
        "%s: read %u bytes at address 0xllu\n",
        userData, bytes, address
    );
}

static void vmic_InstallTrapRead(cpuxP cpux, Addr low, Addr high) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback(domain, processor, low, high, trapRead, "TRAP READ");
}
```

Notes and Restrictions

1. It is legal to install more than one callback on the same address range using this function. In this case, callbacks will be called in the order that they were installed.
2. If a callback is installed on a virtual memory domain, the callback will be invoked whenever the *physical memory* to which that address range is mapped is read, irrespective of the source of the read.
3. Callbacks may be installed on virtual memory ranges that currently have no mapping. If those ranges are subsequently mapped, the callbacks will be invoked whenever the physical memory to which that address range is mapped is read, irrespective of the source of the read, as described above.
4. Callbacks remain active on a domain through all mapping and unmapping operations that may follow, until removed by `vmirtRemoveReadCallback`.
5. Argument scope is present only for VMI version 5.8.0 onwards.
6. Behavior for try-load accesses has changed from VMI version 7.55.0 onwards – see above.

QuantumLeap Semantics

Synchronizing

[9] To be precise, this is the *address with which the read was issued*, which may or may not be a valid virtual address in the domain in which the callback is installed.

11.33 vmirtAddWriteCallback

Prototype

```
void vmirtAddWriteCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine installs a model-specific *write watch point function*, `watchCB`, on the address range `lowAddr:highAddr` in `domain`. The callback will be invoked whenever any address in the range is written during simulation.

Prior to VMI version 7.55.0, the watch point function was called for true stores to the address range only. From VMI version 7.55.0 onwards, the watch point function is also called for try-store accesses (initiated by `vmimtTryStoreRC`, for example). The value argument is non-`NULL` for a store and `NULL` for a try-store.

If `scope` is `NULL`, the callback will be activated on any access to the memory range by *any processor* using any address alias. If scope is non-`NULL` and not the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the *given processor* accesses the memory range using any address alias. If `scope` is the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the domain is accessed by a *master processor* of the domain. A processor is a master processor of a domain if it directly accesses the domain, or accesses a domain that has been aliased to the domain using functions `vmirtAliasMemory` or `vmirtAliasMemoryVM` at any level of indirection.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr          address,      \
    Uns32         bytes,        \
    const void   *value,        \
    void         *userData,     \
    Addr          VA            \
)
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

When called, the `watchCB` is passed the following arguments:

1. `processor`: the processor performing the write, or `NULL` if this is an artifact write (for example, resulting from a call to `vmirtWriteNByteDomain` with `attrs==MEM_AA_FALSE`).
2. `address`: the address of the first byte being written.
3. `bytes`: a count of the bytes being written (typically 1, 2, 4 or 8).

4. `value`: a pointer to a buffer containing the bytes being written, or `NULL` for a try-store access.
5. `userData`: model-specific data, supplied as the first argument to `vmirtAddWriteCallback`.
6. `VA`: the *virtual* address of the first byte being written[10].

Example

```
#include "vmi/vmiRt.h"
#include "vmi/vmiMessage.h"

static VMI_MEM_WATCH_FN(trapWrite) {
    vmiPrintf(
        "%s: wrote %u bytes at address 0x%llu\n",
        userData, bytes, address
    );
}

static void vmic_InstallTrapWrite(cpuxP cpux, Addr low, Addr high) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddWriteCallback(domain, processor, low, high, trapWrite, "TRAP WRITE");
}
```

Notes and Restrictions

1. It is legal to install more than one callback on the same address range using this function. In this case, callbacks will be called in the order that they were installed.
2. If a callback is installed on a virtual memory domain, the callback will be invoked whenever the *physical memory* to which that address range is mapped is written, irrespective of the source of the write.
3. Callbacks may be installed on virtual memory ranges that currently have no mapping. If those ranges are subsequently mapped, the callbacks will be invoked whenever the physical memory to which that address range is mapped is written, irrespective of the source of the write, as described above.
4. Callbacks remain active on a domain through all mapping and unmapping operations that may follow, until removed by `vmirtRemoveWriteCallback`.
5. Argument scope is present only for VMI version 5.8.0 onwards.
6. Behavior for try-store accesses has changed from VMI version 7.55.0 onwards – see above.

QuantumLeap Semantics

Synchronizing

[10] To be precise, this is the *address with which the write was issued*, which may or may not be a valid virtual address in the domain in which the callback is installed.

11.34 vmirtAddFetchCallback

Prototype

```
void vmirtAddFetchCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine installs a model-specific *fetch watch point function*, `watchCB`, on the address range `lowAddr:highAddr` in `domain`. The callback will be invoked whenever any instruction in the address in the range is fetched during simulation.

If `scope` is `NULL`, the callback will be activated on any access to the memory range by *any processor* using any address alias. If scope is non-`NULL` and not the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the *given processor* accesses the memory range using any address alias. If `scope` is the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the domain is accessed by a *master processor* of the domain. A processor is a master processor of a domain if it directly accesses the domain, or accesses a domain that has been aliased to the domain using functions `vmirtAliasMemory` or `vmirtAliasMemoryVM` at any level of indirection.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr          address,      \
    Uns32         bytes,        \
    const void   *value,        \
    void         *userData,     \
    Addr          VA            \
)
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

When called, the `watchCB` is passed the following arguments:

1. `processor`: the processor performing the fetch, or `NULL` if this is an artifact fetch.
2. `address`: the address of the first byte being fetched.
3. `bytes`: a count of the bytes being fetched (typically 1, 2, 4 or 8).
4. `value`: a pointer to a buffer containing the bytes being fetched.
5. `userData`: model-specific data, supplied as the first argument to `vmirtAddFetchCallback`.
6. `VA`: the *virtual* address of the first byte being fetched.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiMessage.h”

static VMI_MEM_WATCH_FN(trapFetch) {
    vmiPrintf(
        “%s: fetched %u bytes at address 0xllu\n”,
        userData, bytes, address
    );
}

static void vmic_InstallWatchFetch(cpuxP cpux, Addr low, Addr high) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddFetchCallback(domain, processor, low, high, trapFetch, “TRAP FETCH”);
}
```

Notes and Restrictions

1. It is legal to install more than one callback on the same address range using this function. In this case, callbacks will be called in the order that they were installed.
2. Callbacks may be installed on virtual memory ranges that currently have no mapping. If those ranges are subsequently mapped, the callbacks will be invoked whenever a fetch is made to an address in that range.
3. Callbacks remain active on a domain through all mapping and unmapping operations that may follow, until removed by `vmirtRemoveFetchCallback`.
4. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

11.35 vmirtRemoveReadCallback

Prototype

```
void vmirtRemoveReadCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine uninstalls any previously-installed read watch point function `watchCB` on the address range `lowAddr:highAddr`. Only calls that were installed with exactly matching `userData` and `scope` will be uninstalled.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr          address,      \
    Uns32         bytes,        \
    const void   *value,        \
    void         *userData,     \
    Addr          VA            \
)
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Example

```
static VMI_MEM_WATCH_FN(trapRead) {
    vmiPrintf(
        “%s: read %u bytes at address 0xllu\n”,
        userData, bytes, address
    );
}

static void vmic_InstallTrapRead(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback(domain, processor, low, high, trapRead, id);
}
static void vmic_RemoveTrapRead(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtRemoveReadCallback(domain, processor, low, high, trapRead, id);
}

static char *id1 = “id1”;
static char *id2 = “id2”;

static void testReadTraps(cpuxP cpux, Addr low, Addr high) {
    // install two read watch point callbacks with userData “id1” and “id2”
    vmic_InstallTrapRead(cpux, low, high, id1);
    vmic_InstallTrapRead(cpux, low, high, id2);
    // remove read watch point callback with userData “id1”
    vmic_RemoveTrapRead(cpux, low, high, id1);
}
```

Notes and Restrictions

1. Address ranges used when installing and uninstalling callbacks do not need to match. For example, it is legal to install a callback on a wide range and uninstall it only from a narrower range. In this case, the callback will remain active on the part or parts of the install range that are excluded from the uninstall range.
2. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

11.36 vmirtRemoveWriteCallback

Prototype

```
void vmirtRemoveWriteCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine uninstalls any previously-installed write watch point function `watchCB` on the address range `lowAddr:highAddr`. Only calls that were installed with exactly matching `userData` and `scope` will be uninstalled.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr          address,      \
    Uns32         bytes,        \
    const void   *value,        \
    void         *userData,     \
    Addr          VA            \
)
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Example

```
static VMI_MEM_WATCH_FN(trapWrite) {
    vmiPrintf(
        "%s: write %u bytes at address 0xllu\n",
        userData, bytes, address
    );
}

static void vmic_InstallTrapWrite(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddWriteCallback(domain, processor, low, high, trapWrite, id);
}
static void vmic_RemoveTrapWrite(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtRemoveWriteCallback(domain, Processor, low, high, trapWrite, id);
}

static char *id1 = "id1";
static char *id2 = "id2";

static void testWriteTraps(cpuxP cpux, Addr low, Addr high) {
    // install two write watch point callbacks with userData "id1" and "id2"
    vmic_InstallTrapWrite(cpux, low, high, id1);
    vmic_InstallTrapWrite(cpux, low, high, id2);
    // remove write watch point callback with userData "id1"
    vmic_RemoveTrapWrite(cpux, low, high, id1);
}
```

Notes and Restrictions

1. Address ranges used when installing and uninstalling callbacks do not need to match. For example, it is legal to install a callback on a wide range and uninstall it only from a narrower range. In this case, the callback will remain active on the part or parts of the install range that are excluded from the uninstall range.
2. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

11.37 vmirtRemoveFetchCallback

Prototype

```
void vmirtRemoveFetchCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine uninstalls any previously-installed fetch watch point function `watchCB` on the address range `lowAddr:highAddr`. Only calls that were installed with exactly matching `userData` and `scope` will be uninstalled.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    Addr          address,      \
    Uns32         bytes,        \
    const void   *value,        \
    void         *userData,     \
    Addr          VA            \
)
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiMessage.h”

static VMI_MEM_WATCH_FN(trapFetch) {
    vmiPrintf(
        “%s: fetch %u bytes at address 0xllu\n”,
        userData, bytes, address
    );
}

static void vmic_InstallTrapFetch(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddFetchCallback(domain, processor, low, high, trapFetch, id);
}
static void vmic_RemoveTrapFetch(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtRemoveFetchCallback(domain, processor, low, high, trapFetch, id);
}

static char *id1 = “id1”;
static char *id2 = “id2”;

static void testFetchTraps(cpuxP cpux, Addr low, Addr high) {
    // install two fetch watch point callbacks with userData “id1” and “id2”
    vmic_InstallTrapFetch(cpux, low, high, id1);
    vmic_InstallTrapFetch(cpux, low, high, id2);
    // remove fetch watch point callback with userData “id1”
    vmic_RemoveTrapFetch(cpux, low, high, id1);
}
```

Notes and Restrictions

1. Address ranges used when installing and uninstalling callbacks do not need to match. For example, it is legal to install a callback on a wide range and uninstall it only from a narrower range. In this case, the callback will remain active on the part or parts of the install range that are excluded from the uninstall range.
2. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

11.38 vmirtAliasMemoryVM

Prototype

```
Bool vmirtAliasMemoryVM(
    memDomainP physicalDomain,
    memDomainP virtualDomain,
    Addr       physicalLowAddr,
    Addr       physicalHighAddr,
    Addr       virtualLowAddr,
    memMRUSetP mruSet,
    memPriv    privilege,
    Uns64      ASIDMaskOrG,
    Uns64      ASID
);
```

Description

This routine creates a virtual memory page alias of a region in a *physical domain* so that that physical memory is visible within a *virtual domain*. The region to be aliased has the address range `physicalLowAddr:physicalHighAddr` inclusive in the physical domain. Within the virtual domain, the same memory appears starting at address `virtualLowAddr`. The new mapping is created with access privileges specified by `privilege`:

```
//
// Memory access privilege (bitmask)
//
typedef enum memPrivE {

    // access privilege options
    MEM_PRIV_NONE    = 0x0000,  // no access permitted
    MEM_PRIV_R       = 0x0001,  // read permitted
    MEM_PRIV_W       = 0x0002,  // write permitted
    MEM_PRIV_X       = 0x0004,  // execute permitted
    MEM_PRIV_RW      = (MEM_PRIV_R|MEM_PRIV_W           ),
    MEM_PRIV_RX      = (MEM_PRIV_R|           MEM_PRIV_X),
    MEM_PRIV_WX      = (           MEM_PRIV_W|MEM_PRIV_X),
    MEM_PRIV_RWX     = (MEM_PRIV_R|MEM_PRIV_W|MEM_PRIV_X),

    // alignment constraints
    MEM_PRIV_ALIGN_M = 0x0008,  // check alignment, report memory address
    MEM_PRIV_ALIGN_P = 0x0010,  // check alignment, report access address
    MEM_PRIV_ALIGN   = (MEM_PRIV_ALIGN_M|MEM_PRIV_ALIGN_P),

    // user-defined constraints
    MEM_PRIV_USER1   = 0x0100,  // user-defined restricted access 1
    MEM_PRIV_USER2   = 0x0200,  // user-defined restricted access 2
    MEM_PRIV_USER3   = 0x0400,  // user-defined restricted access 3
    MEM_PRIV_USER4   = 0x0800,  // user-defined restricted access 4
    MEM_PRIV_USER5   = 0x1000,  // user-defined restricted access 5
    MEM_PRIV_USER6   = 0x2000,  // user-defined restricted access 6
    MEM_PRIV_USER7   = 0x4000,  // user-defined restricted access 7
    MEM_PRIV_USER8   = 0x8000,  // user-defined restricted access 8
    MEM_PRIV_USER    = (MEM_PRIV_USER1|MEM_PRIV_USER2|
                        MEM_PRIV_USER3|MEM_PRIV_USER4|
                        MEM_PRIV_USER5|MEM_PRIV_USER6|
                        MEM_PRIV_USER7|MEM_PRIV_USER8),

    // legacy user-defined constraint
    MEM_PRIV_DEVICE  = MEM_PRIV_USER1,

} memPriv;
```

Values `MEM_PRIV_R`, `MEM_PRIV_W` and `MEM_PRIV_X` specify read, write and execute privilege, respectively; compound privileges `MEM_PRIV_RW`, `MEM_PRIV_RX`, `MEM_PRIV_WX` and `MEM_PRIV_RWX` specify combinations of these basic privileges.

Values `MEM_PRIV_ALIGN_M` and `MEM_PRIV_ALIGN_P` specify that accesses to the address range must be *aligned to the access size*; this is required by some processors (e.g. ARM processors with a TLB) in which misaligned accesses are not allowed to regions of memory with certain properties. If `MEM_PRIV_ALIGN_M` is specified, then on an unaligned access failure the processor alignment handler will be passed *the lowest address that was indicated as requiring alignment*. If `MEM_PRIV_ALIGN_P` is specified and `MEM_PRIV_ALIGN_M` is *not* specified, then on an unaligned access failure the processor alignment handler will be passed the lowest address of the *unaligned access,* even if that address was not specified to require aligned access. The addresses passed to the handler differ only in the case that an unaligned access is made that straddles the boundary between a lower-address region that *does not* require aligned access and a higher-address region that *does* require aligned access.

Values `MEM_PRIV_USER1` to `MEM_PRIV_USER8` specify that this region of memory has user-defined restricted access permissions: see the *Imperas Morph Time Function Reference* for more information about how user-defined region access is constrained. Value `MEM_PRIV_DEVICE` is present for legacy reasons and equivalent to `MEM_PRIV_USER1`.

The type of mapping to create is specified using the `ASIDMaskOrG` argument. Prior to VMI version 4.5.0, this argument was a simple boolean, `isGlobal`. A value of `True` indicated that the mapping was a *global* mapping (always valid); a value of `False` indicated that the mapping was only valid when the processor ASID matched the passed `ASID` (see function `vmirtSetProcessorASID`). From VMI version 4.5.0, the parameter has been changed to an `Uns64`, `ASIDMaskOrG`. This is interpreted as follows:

1. **`ASIDMaskOrG=1 (True)`**: A value of 1 indicates that the mapping is global and the `ASID` is ignored, as previously.
2. **`ASIDMaskOrG=0 (False)`**: A value of 0 indicates that the mapping is ASID-mapped and that the processor ASID must exactly match the passed `ASID` for the mapping to be valid, as previously.
3. **`ASIDMaskOrG=<another value>`**: If any other value is specified for the `ASIDMaskOrG` argument, then this value is used as a *mask* to apply to the processor ASID to determine if the region matches. Specifically, the region is deemed to match only if:
```
(REGION_ASID&ASIDMaskOrG)==(PROCESSOR_ASID&ASIDMaskOrG)
```

This modified mapping behavior simplifies implementation of processor models that support *hardware hypervisors*. Such a processor usually allows memory regions to be created in different classes:

1. Global regions;
2. ASID-mapped regions;

2. Regions which must match a *virtual machine ID* (VMID);
3. Regions that must match a VMID and an ASID.

Typically, such processors also have two levels of TLB in some operating modes: a *guest* TLB that maps addresses to intermediate physical addresses (IPAs) and a *second stage* TLB that maps the IPA to a true physical address. Using an ASID mask enables both TLB stages to be implemented as a single `vmirtAliasMemoryVM` call with the ASID mask used to select the parts of a simulated ASID that are relevant for this region. The simulated ASID passed to `vmirtAliasMemoryVM` will usually contain:

1. The region ASID as a bitfield;
2. The processor VMID as a bitfield;
3. Further bits indicated the enabled status of the guest and second stage TLBs.

The same physical memory may be mapped at one or more virtual addresses in one or more virtual domains.

The `mruSet` argument is used by the simulator's native MRU/LRU management, typically to implement a TLB LRU replacement policy – See the *Imperas Processor Modeling Guide* for more information and examples.

The function returns a boolean indicating whether the mapping succeeded.

Example

The OVP RISC-V processor model uses this function to create TLB mappings:

```
static void mapTLBEntry(
    riscvP      riscv,
    tlbEntryP   entry,
    memDomainP  domainV,
    riscvMode   mode,
    memPriv     requiredPriv,
    tlbMapInfoP miP
) {
    memDomainP domainP    = getPMPDomainPriv(riscv, mode, requiredPriv);
    Uns64      lowPA      = getEntryLowPA(entry);
    Uns64      highPA     = getEntryHighPA(entry);
    Uns64      lowVA      = getEntryLowVA(entry);
    Uns32      ASIDMask   = getEntryASIDMask(entry, mode);
    Uns32      ASID       = getEntrySimASID(entry);
    memPriv    priv       = miP->priv;
    Uns64      size       = highPA-lowPA+1;
    Uns64      vmiPageMax = 0x100000000ULL;

    // restrict mapping size to VMI maximum (4Gb)
    if(size>vmiPageMax) {

        Uns64 VAtoPA = lowPA-lowVA;

        size   = vmiPageMax;
        lowVA  = miP->lowVA & -size;
        lowPA  = lowVA + VAtoPA;
        highPA = lowPA + size - 1;
    }

    // create virtual mapping
    vmirtAliasMemoryVM(
        domainP, domainV, lowPA, highPA, lowVA, 0, priv, ASIDMask, ASID
    );
    // update PMP mapping if required
    mapPMP(riscv, mode, lowPA, highPA);

    // indicate entry is mapped in this mode
    entry->isMapped |= getModeMask(mode);

    // indicate mapped range
    miP->lowVA  = lowVA;
    miP->highVA = lowVA + size - 1;
}
```

Notes and Restrictions

1. Access permissions in the new virtual memory region are those of the physical memory region, restricted further by the passed `privilege`. If the passed `privilege` specifies an access permission that is not present in the physical region, that access permission is not granted.
2. Paged alias regions must have a size that is an exact power of two and must be aligned to a boundary that is a multiple of the same power of two.
3. Paged alias regions are limited to a maximum size of 4Gb (2^32 bytes).
4. `vmirtAliasMemoryVM` may not be used to create an alias to a physical domain that itself has paged aliases.
5. If a virtual domain contains any paged alias regions, then all such regions must refer to the same physical domain (it is illegal to create paged aliases to two or more physical domains from one virtual domain).

QuantumLeap Semantics

Synchronizing

11.39 vmirtUnaliasMemoryVM

Prototype

```
Bool vmirtUnaliasMemoryVM(
    memDomainP virtualDomain,
    Addr       virtualLowAddr,
    Addr       virtualHighAddr,
    Uns64      ASIDMaskOrG,
    Uns64      ASID
);
```

Description

This routine removes the page-mapped address range `virtualLowAddr:virtualHighAddr` inclusive from the virtual domain, and returns a boolean indicating whether the unmapping succeeded. For the mapping to be removed, the addresses must previously have been mapped by a call to `vmirtAliasMemoryVM`, using the same values for `ASIDMaskOrG` and `ASID`. Once the address range has been removed from the domain, any attempt to read, write or fetch using addresses in the range will cause the corresponding *memory access exception handler* for the processor model to be called. When implementing an MMU model, these exception handlers will typically remap memory using `vmirtAliasMemoryVM` or perform other actions required to model the MMU.

Refer to the documentation of `vmirtAliasMemoryVM` for a detailed description of the `ASID` and `ASIDMaskOrG` arguments.

Example

The OVP RISC-V processor model uses this function to delete TLB mappings:

```
static void deleteTLBEntryMappingsMode(
    riscvP    riscv,
    tlbEntryP entry,
    riscvMode mode
) {
    Uns32 modeMask = getModeMask(mode);

    // action is only needed if the TLB entry is mapped in this mode
    if(entry->isMapped & modeMask) {

        memDomainP dataDomain = riscv->vmDomains[mode][0];
        memDomainP codeDomain = riscv->vmDomains[mode][1];
        Uns64      lowVA      = getEntryLowVA(entry);
        Uns64      highVA     = getEntryHighVA(entry);
        Uns32      fullASID   = getEntrySimASID(entry);
        Uns32      ASIDMask   = getEntryASIDMask(entry, mode);

        if(dataDomain) {
            vmirtUnaliasMemoryVM(dataDomain, lowVA, highVA, ASIDMask, fullASID);
        }

        if(codeDomain && (codeDomain!=dataDomain)) {
            vmirtUnaliasMemoryVM(codeDomain, lowVA, highVA, ASIDMask, fullASID);
        }

        // indicate entry is no longer mapped in this mode
        entry->isMapped &= ~modeMask;
    }
}
```

Notes and Restrictions

1. Address ranges specified with `vmirtUnaliasMemoryVM` must exactly match the ranges previously specified using `vmirtAliasMemoryVM` when the mapping was established.
2. Using `vmirtUnaliasMemoryVM` to remove an alias where none was established has no effect.
3. Using `vmirtUnaliasMemoryVM` to remove an alias which was established with different `isGlobal` or `ASID` values has no effect. Use `vmirtUnaliasMemory` to remove an alias irrespective of `isGlobal` and `ASID`.

QuantumLeap Semantics

Synchronizing

11.40 vmirtGetDomainMappedASID

Prototype

```
memMappedASID vmirtGetDomainMappedASID(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr,
    Uns64      ASID
);
```

Description

This routine returns an enumeration value indicating whether any address in the range `lowAddr:highAddr` is ASID-mapped in the `domain` with the passed `ASID`. Type `memMappedASID` is defined in `vmiTypes.h` as follows:

```
typedef enum memMappedASIDE {
    ASID_MAP_NONE,                  // address not ASID-mapped
    ASID_MAP_INACTIVE,              // address ASID-mapped but not active
    ASID_MAP_ACTIVE                 // address ASID-mapped and active
} memMappedASID;
```

A return value of `ASID_MAP_NONE` means that no address in the range is mapped with that ASID. A return value of `ASID_MAP_INACTIVE` means that there is some address in that range mapped using the ASID, but the mapped memory has not been referenced since the processor last switched to the ASID (in other words, the memory has not been recently addressed and is stale). A return value of `ASID_MAP_ACTIVE` means that there is some address in that range mapped using the ASID and that the referenced memory has been referenced since the processor last switched to the ASID (in other words, the memory is not stale).

This function is typically used for simulation efficiency reasons to determine whether or not a TLB mapping is valid, or should be preserved.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

11.41 vmirtSetProcessorASID

Prototype

```
void vmirtSetProcessorASID(vmiProcessorP processor, Uns64 ASID);
```

Description

This routine sets the processor ASID (*address space ID*) to the passed value. If a non-global page-mapped alias is created with `vmirtAliasMemoryVM`, then that mapping will be valid only if the ASID matches the current processor ASID, taking into account any ASID mask.

Example

The OVP ARM processor model uses this function to set a simulated ASID, which combines ASID, VMID and other control bits:

```
void armVMSetASID(armP arm) {

    Uns32      ASID    = getASID(arm);
    armSimASID simASID = getSimASID(arm, ASID);

    vmirtSetProcessorASID((vmiProcessorP)arm, simASID.u64);
}
```

Notes and Restrictions

1. The initial processor ASID is 0.

QuantumLeap Semantics

Non-self-synchronizing

11.42 vmirtGetProcessorASID

Prototype

```
Uns64 vmirtGetProcessorASID(vmiProcessorP processor);
```

Description

This routine returns the processor ASID (*address space ID*) previously set by `vmirtSetProcessorASID`.

Example

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

Notes and Restrictions

1. The initial processor ASID is 0.

QuantumLeap Semantics

Non-self-synchronizing

11.43 vmirtGetMRUStateTable

Prototype

```
const Uns32 *vmirtGetMRUStateTable(Uns32 numEntries, Uns32 entryIndex);
```

Description

This function is used when modeling constructs such as *translation lookaside buffers* (TLBs) with *least-recently-used* (LRU) replacement policies. It returns a transition table to be used for entry `entryIndex` of a LRU-managed set with `numEntries` entries.

It is possible to associate a memory region mapping with a transition table / state variable pair using the `mruSet` argument of `vmirtAliasMemory`. When this association has been made, the simulator will automatically maintain the state variable so that the least-recently-used region of a set of regions can be determined using `vmirtGetNthStateIndex`.

Example

See the *Imperas Processor Modeling Guide* for detailed examples using this function.

Notes and Restrictions

1. Currently, a value of `numEntries` greater than `8` is not supported.
2. `entryIndex` must lie in the range `0..numEntries-1`.

QuantumLeap Semantics

Synchronizing

11.44 vmirtGetNthStateIndex

Prototype

```
Uns8 vmirtGetNthStateIndex(Uns32 numEntries, Uns32 state, Uns32 position);
```

Description

This function is used when modeling constructs such as *translation lookaside buffers* (TLBs) with *least-recently-used* (LRU) replacement policies. Given a state in a table with `numEntries` previously allocated by `vmirtGetMRUStateTable`, it returns the entry index with the passed position implied by `state`. The most-recently-used entry has position `0`; the least-recently-used entry has position `numEntries-1`.

It is possible to associate a memory region mapping with a transition table / state variable pair using the `mruSet` argument of `vmirtAliasMemory`. When this association has been made, the simulator will automatically maintain the state variable so that the least-recently-used region of a set of regions can be determined using this function.

Example

See the *Imperas Processor Modeling Guide* for detailed examples using this function.

Notes and Restrictions

1. `position` must lie in the range `0..numEntries-1`.
2. This function returns a correct state index only for state transition tables allocated with `vmirtGetMRUStateTable`.

QuantumLeap Semantics

Synchronizing

11.45 vmirtAddPCCallback

Prototype

```
Bool vmirtAddPCCallback(
    vmiProcessorP processor,
    Addr          simPC,
    vmiPCWatchFn  watchCB,
    void         *userData
);
```

Description

This routine registers a callback function `watchCB` that is called whenever the processor executes as instruction at virtual address `simPC`. Type `vmiPCWatchFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_PC_WATCH_FN(_NAME) void _NAME( \
    void          *userData,  \
    vmiProcessorP processor, \
    Addr          thisPC     \
)
typedef VMI_PC_WATCH_FN((*vmiPCWatchFn));
```

The function return address indicates whether the callback was successfully added. The callback can be removed using `vmirtRemovePCCallback`.

This function is most commonly used in combination with the file/line query API to implement coverage tools: see the example.

Example

This example shows the core of a *line coverage tool*, implemented as a binary intercept library.

```
static VMI_PC_WATCH_FN(incrementCount) {
    coverLineP line = (coverLineP)userData;
    line->count++;
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiFileLineCP fl;
    coverLineP    line = &object->lines[0];

    // iterate over the file/line pairs recorded
    // in the DWARF information for the processor executable
    // and register a callback when the indicated address is executed
    for(
        fl=vmirtNextFLByAddr(processor, 0);
        fl!=NULL && line<=&object->lines[MAX_LINES];
        fl=vmirtNextFLByAddr(processor, fl), line++, object->numLines++
    ) {

        Addr address = vmirtGetFLAddr(fl);

        // Set the file/line pointer for this entry
        line->fl = fl;

        // Add callback on the address for this file/line
        // with the appropriate coverLineP to increment
        vmirtAddPCCallback(processor, address, incrementCount, line);
    }
}
```

Notes and Restrictions

1. This function is available only with the Imperas Professional Tools.
2. It is possible to add many callbacks with the same `processor` and `simPC` arguments. Each will be called in turn when the processor executes at the given simulated address.

QuantumLeap Semantics

Synchronizing

11.46 vmirtRemovePCCallback

Prototype

```
Bool vmirtRemovePCCallback(
    vmiProcessorP processor,
    Addr          simPC,
    vmiPCWatchFn  watchCB,
    void         *userData
);
```

Description

This routine uninstalls a callback function `watchCB` that was previously installed using the passed `processor`, `simPC`, `watchCB` and `userData` arguments. Type `vmiPCWatchFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_PC_WATCH_FN(_NAME) void _NAME( \
    void         *userData,  \
    vmiProcessorP processor, \
    Addr          thisPC     \
)
typedef VMI_PC_WATCH_FN((*vmiPCWatchFn));
```

The function return address indicates whether the callback was successfully removed: al value of `False` indicates that no callback was previously installed with matching arguments.

This function is most commonly used to implement tools that require simulated PC tracking to change dynamically (for example, a tool that performs function call/return tracing).

Example

This example shows part of a function call/return tracing tool, implemented as a binary intercept library. A callback is installed on every function return address; when called, this function takes appropriate action to track the return and then uninstalls itself.

```
static VMI_PC_WATCH_FN(exitFunc) {

    callContextP context = (callContextP)userData;
    Uns32        stackPointer;

    vmiosRegRead(processor, context->object->sp, &stackPointer);

    // If stack pointer matches then we have found
    // the matching return for this context
    if(stackPointer==context->sp) {

        // get the function result
        Int32 result;
        vmiosRegRead(processor, context->object->result, &result);

        // Report the function return (including elapsed instructions)
        vmiPrintf(
            "*** breturn-plugin: cpu %s: Function %s returned with result %d "
            "(" FMT_64u " instructions)\n",
            vmirtProcessorName(processor),
            TRACE_FUNC,
            result,
            vmirtGetICount(processor)-context->entryICount
        );

        // Remove this callback
        vmirtRemovePCCallback(processor, thisPC, exitFunc, context);

        // Free the memory allocated for the context
        free(context);
    }
}
```

Notes and Restrictions

1. This function is available only with the Imperas Professional Tools.
2. If there are several callbacks installed with the same `processor`, `simPC`, `watchCB` and `userData` arguments, only one instance will be uninstalled by each call to `vmirtRemovePCCallback`. Installing identical callbacks in this way is not recommended.

QuantumLeap Semantics

Synchronizing

11.47 vmirtUpdatePCCallbackCondition

Prototype

```
Bool vmirtUpdatePCCallbackCondition(
    vmiProcessorP processor,
    Addr           simPC,
    vmiPCWatchFn  watchCB,
    void         *userData,
    const char   *condition
);
```

Description

This routine updates a *watch condition* on a callback function `watchCB` that was previously installed using the passed `processor`, `simPC`, `watchCB` and `userData` arguments. See function `vmirtAddPCCallback` for more information on the use of PC callback functions.

The *watch condition* restricts the conditions under which the callback is executed to a particular processor mode, or a particular ASID, or a combination of mode and ASID. This is especially useful in an intercept library that is tracking behavior in a user process running in a simulated operating system such as Linux: in this case, the callback can be conditioned so that it is only activated in that user process context.

A watch condition is specified as a colon separated pair of the form:

```
"<mode>:<ASID>"
```

The `<mode>` is the current processor mode as defined by:

```
vmirtGetCurrentMode(processor)->code
```

The `<ASID>` is the current processor ASID as defined by:

```
vmirtGetProcessorASID(processor)
```

Either mode or ASID may be omitted, as shown in the following examples.

`"3:14"`	specifies processor mode 3 and ASID 14
`"7"`	specifies processor mode 7 and any ASID
`":26"`	specifies any processor mode and ASID 26

It is possible to specify a *mask* for either mode or ASID by concatenating an ampersand and a second numeric value. In this case, the current processor mode or ASID value is masked using bitwise-and with the given mask before comparing with the specified pattern in order to determine if the pattern matches. For example, the pattern:

```
":14&0xffff"
```

matches when:

```
(vmirtGetProcessorASID(processor) & 0xffff) == 14
```

The condition argument to `vmirtUpdatePCCallbackCondition` can take four forms:
1. *Replacement form*. In this case, the condition replaces any existing watch condition on the callback function. To do this, pass a watch condition string exactly as described above; for example:

```
“3:14”
```

2. *Inclusive form*. In this case, the given condition explicitly specifies a match, in addition to any existing condition on the callback function. In other words, the callback will be activated if either a previously-specified condition matches or the newly-supplied condition matches. To do this, prepend a + character to the condition, for example:

```
“+3:14”
```

3. *Exclusive form*. In this case, the given condition is explicitly excluded from matching. In other words, the callback will *not* be activated if this condition matches, even if a previously-specified inclusive condition matches. To do this, prepend a - character to the condition, for example:

```
“-3:14”
```

4. *Clear form*. This special case clears any active watch condition so that the callback is unconditional:

```
“clear”
```

The function returns a Boolean value indicating success. It will return `False` if either there is no installed callback function that matches the passed `processor`, `simPC`, `watchCB` and `userData` arguments, or if the syntax of the `condition` argument is invalid.

Example

This example shows part of a mode monitor tool, implemented as a binary intercept library. A callback is installed on a particular function address; when called, the function increments a count of unique mode/ASID contexts found and then uses an exclusive form argument to `vmirtUpdatePCCallbackCondition` to remove the current mode and ASID from its own activation condition. This ensures it will not be activated again for this mode/ASID pair.

```
static const char *getActiveScope(
    vmiProcessorP processor,
    Addr          thisPC,
    Void         *userData
) {
    return vmirtGetPCCallbackCondition(processor, thisPC, watchPC, userData);
}

static VMI_PC_WATCH_FN(watchPC) {

    vmiosObjectP  object = (vmiosObjectP)userData;
    vmiModeInfoCP mode   = vmirtGetCurrentMode(processor);
    Uns32         ASID   = vmirtGetProcessorASID(processor);
    char          updateScope[32];

    if(!mode) {

        vmiPrintf("*** processor does not implement modes ***\n");

    } else {

        vmiPrintf("processor is in mode %u (%s)\n", mode->code, mode->name);

        // get processor scope as exclusive form argument
        sprintf(updateScope, "-%u:%u", mode->code, ASID);
        object->count++;

        // print scope before it is updated
        vmiPrintf(
            "Active scope before: %s\n",
            getActiveScope(processor, thisPC, userData)
        );

        // update scope
        vmirtUpdatePCCallbackCondition(
            processor, thisPC, watchPC, userData, updateScope
        );

        // print scope after it is updated
        vmiPrintf(
            "Active scope after : %s\n",
            getActiveScope(processor, thisPC, userData)
        );
    }
}
```

Notes and Restrictions

1. This function is available only with the Imperas Professional Tools.
2. If there are several callbacks installed with the same `processor`, `simPC`, `watchCB` and `userData` arguments, only one instance will be affected by each call to `vmirtUpdatePCCallbackCondition`. Installing identical callbacks in this way is not recommended.

QuantumLeap Semantics

Synchronizing

11.48 vmirtGetPCCallbackCondition

Prototype

```
const char *vmirtGetPCCallbackCondition(
    vmiProcessorP processor,
    Addr          simPC,
    vmiPCWatchFn  watchCB,
    void         *userData
);
```

Description

This routine returns the current *watch condition* on a callback function `watchCB` that was previously installed using the passed `processor`, `simPC`, `watchCB` and `userData` arguments. See function `vmirtAddPCCallback` for more information on the use of PC callback functions, and function `vmirtUpdatePCCallbackCondition` for more information about the purpose of watch conditions.

The watch condition is returned as a space-separated list of individual conditions, possibly negated and separated with `and`/`or` keywords. Some examples:

`“3:14”`
indicates processor mode 3 and ASID 14

`“3:14 or 3:15 or 3:16”`
indicates processor mode 3 and ASID 14, 15 or 16.

`“!3:14”`
indicates any mode except mode 3 with ASID 14

`“!3:14 and !3:15 and !3:16”`
indicates any mode except mode 3 with ASID 14, 15 or 16

`“clear”`
indicates that there is no watch condition

Example

This example shows part of a mode monitor tool, implemented as a binary intercept library. A callback is installed on a particular function address; when called, the function increments a count of unique mode/ASID contexts found and then uses an exclusive form argument to `vmirtUpdatePCCallbackCondition` to remove the current mode and ASID from its own activation condition. This ensures it will not be activated again for this mode/ASID pair. The tool uses `vmirtGetPCCallbackCondition` to display the active watch condition before and after the condition update.

```
static const char *getActiveScope(
    vmiProcessorP processor,
    Addr          thisPC,
    Void         *userData
) {
    return vmirtGetPCCallbackCondition(processor, thisPC, watchPC, userData);
}

static VMI_PC_WATCH_FN(watchPC) {

    vmiosObjectP  object = (vmiosObjectP)userData;
    vmiModeInfoCP mode   = vmirtGetCurrentMode(processor);
    Uns32         ASID   = vmirtGetProcessorASID(processor);
    char          updateScope[32];

    if(!mode) {

        vmiPrintf("*** processor does not implement modes ***\n");

    } else {

        vmiPrintf("processor is in mode %u (%s)\n", mode->code, mode->name);

        // get processor scope as exclusive form argument
        sprintf(updateScope, "-%u:%u", mode->code, ASID);

        object->count++;

        // print scope before it is updated
        vmiPrintf(
            "Active scope before: %s\n",
            getActiveScope(processor, thisPC, userData)
        );

        // update scope
        vmirtUpdatePCCallbackCondition(
            processor, thisPC, watchPC, userData, updateScope
        );

        // print scope after it is updated
        vmiPrintf(
            "Active scope after : %s\n",
            getActiveScope(processor, thisPC, userData)
        );
    }
}
```

Notes and Restrictions

1. This function is available only with the Imperas Professional Tools.
2. If there are several callbacks installed with the same `processor`, `simPC`, `watchCB` and `userData` arguments, only conditions on one instance will be returned by each call to `vmirtGetPCCallbackCondition`. Installing identical callbacks in this way is not recommended.

QuantumLeap Semantics

Synchronizing

11.49 vmirtDebugDomain

Prototype

```
void vmirtDebugDomain(memDomainP domain);
```

Description

Given an argument of type `memDomainP`, this function emits information about the regions into which the address space of that domain is broken. The intended use is for debugging address mapping or permission errors when developing models.

The output from `vmirtDebugDomain` is quite verbose; as an example:

```
DOMAIN dummy/S2 REGIONS:
      1:     0x00000000:0x00000fff type:RAM  priv:rwx raw:rwx opt:r. cb:...
device:.. align:.. flags:MDC. blocks:1    (master dummy/M,0x00000000:0x00000fff)
(parent dummy/S1,0x00000000:0xffffffff)
      2:     0x00001000:0x0000ffff type:RAM  priv:rwx raw:rwx opt:rw cb:...
device:.. align:.. flags:MD.. blocks:0    (master dummy/M,0x00001000:0x0000ffff)
(parent dummy/S1,0x00000000:0xffffffff)
      3:     0x00010000:0x000fffff type:RAM  priv:--- raw:rwx opt:.. cb:...
device:.. align:.. flags:...U blocks:0    (master dummy/M,0x00010000:0x000fffff)
(parent dummy/S1,0x00000000:0xffffffff)
      4:     0x00100000:0x0010ffff type:RAM  priv:rwx raw:rwx opt:rw cb:...
device:.. align:.. flags:MD.. blocks:0    (master dummy/M,0x00100000:0x0010ffff)
(parent dummy/S1,0x00000000:0xffffffff)
      5:     0x00110000:0x001fffff type:RAM  priv:--- raw:rwx opt:.. cb:...
device:.. align:.. flags:...U blocks:0    (master dummy/M,0x00110000:0x001fffff)
(parent dummy/S1,0x00000000:0xffffffff)
      6:     0x00200000:0x0020ffff type:RAM  priv:rwx raw:rwx opt:rw cb:...
device:.. align:.. flags:MD.. blocks:0    (master dummy/M,0x00200000:0x0020ffff)
(parent dummy/S1,0x00000000:0xffffffff)
      7:     0x00210000:0xffffffff type:RAM  priv:--- raw:rwx opt:.. cb:...
device:.. align:.. flags:...U blocks:0    (master dummy/M,0x00210000:0xffffffff)
(parent dummy/S1,0x00000000:0xffffffff)
```

This example domain contains seven regions of RAM memory. Each region has an *effective* privilege (`priv`) and a *raw* privilege (`raw`), which is the privilege set on that region by `vmirtProtectMemory`. The effective privilege may be more restricted than the raw privilege if this region is an alias of another with lower privileges (see region 3 in the above example). The `opt` field indicates whether the simulator can perform optimized read/write accesses, or whether more complex checking is required. The `cb` field indicates whether the region has installed watch callbacks for read, write or execute. The `device` field indicates whether the region is modeled using a callback. The `align` field indicates whether the region has any memory-side alignment checks enforced. The `flags` field indicates implementation details of the region: important values are `M` (implying the region is *mapped*, in other words that is it has true memory associated with it) and `C` (implying that there are JIT-compiled code blocks associated with it). If there are JIT-compiled blocks, the `blocks` field specifies how many. Remaining fields indicate if the region has a *parent* in another domain (in other words, it is an alias of that region) and also the ultimate target of a hierarchy of aliases (the *master*).

Example

The OVP V850 processor model uses this function in a debug mode to validate MPU region changes:

```
static void memoryMapping(v850P v850) {
    if(V850_DEBUG_MPU(v850)) {
        vmiPrintf("Code Domain");
        vmirtDebugDomain(vmirtGetProcessorCodeDomain((vmiProcessorP)v850));
        vmiPrintf("Data Domain");
        vmirtDebugDomain(vmirtGetProcessorDataDomain((vmiProcessorP)v850));
        vmiPrintf("\n");
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

12 Accessing and Setting the Program Counter

The simulator does not directly model the processor program counter (to do so would make simulation much slower). Instead, it provides a run time interface to get and set the program counter (that is, to perform a jump) if required. This section describes functions related to program counter management.

12.1 vmirtGetPC

Prototype

```
Addr vmirtGetPC(vmiProcessorP processor);
```

Description

This routine returns the processor program counter.

Example

The OVP RISC-V model uses this function when modeling NMI exceptions:

```
inline static Uns64 getPC(riscvP riscv) {
    return vmirtGetPC((vmiProcessorP)riscv);
}

static void doNMI(riscvP riscv) {

    // restart the processor from any halted state
    restartProcessor(riscv, RVD_RESTART_NMI);

    // switch to Machine mode
    riscvSetMode(riscv, RISCV_MODE_MACHINE);

    // update cause register (to zero)
    WR_CSR(riscv, mcause, 0);

    // update mepc to hold next instruction address
    WR_CSR(riscv, mepc, getPC(riscv));

    // indicate the taken exception
    riscv->exception = 0;

    // set address at which to execute
    vmirtSetPCException((vmiProcessorP)riscv, riscv->configInfo.nmi_address);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

12.2 vmirtGetPCDS

Prototype

```
Addr vmirtGetPCDS(vmiProcessorP processor, Uns8 *delaySlotOffset);
```

Description

This routine returns the current processor program counter and delay slot offset (using a byref argument, which must be non-`NULL`). It is required in order to correctly model interrupt behavior on processors with delay slots. There are two distinct modes:

1. If the current instruction is *not* a delay slot instruction, the function returns the current program counter and sets the byref argument `delaySlotOffset` to 0.
2. If the current instruction is a delay slot instruction, the function returns the address of the *branch instruction for which this is a delay slot* and sets the byref argument `delaySlotOffset` to *the offset of the delay slot instruction from the branch instruction*. The address of the delay slot instruction can therefore be calculated if required by adding `delaySlotOffset` to the function result.

As an example, on a machine with a 4-byte instruction format and a single delay slot instruction per branch, when called from a delay slot instruction `vmirtGetPCDS` will return the address of the branch instruction and set `delaySlotOffset` to `4`.

Example

The OVP OR1K model uses this function when taking an exception:

```
void or1kTakeException(or1kP or1k, or1kException exception, Uns32 pcOffset) {

    Uns8  simD;
    Uns32 simPC = (Uns32)vmirtGetPCDS((vmiProcessorP)or1k, &simD);

    or1kEnterSupervisorMode(or1k);
    or1k->EPC = simPC + pcOffset;

    // set sr[DSX] for exception in a delay slot
    if(simD) {
        or1k->SR |= SPR_SR_DSX;
    }

    // jump to the vector
    vmirtSetPCException((vmiProcessorP)or1k, exceptions[exception].code);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

12.3 vmirtSetPC

Prototype

```
void vmirtSetPC(vmiProcessorP processor, Addr newPC);
```

Description

This routine sets the current program counter. This effectively performs a *branch* operation from within a run time call. When called from an embedded function call from translated native code, the branch takes effect just after the return from the embedded function call. Any code after the embedded function call in the translated native code is not executed.

This function differs from `vmirtSetPCException` in that exception watchpoints installed for the processor will *not* be triggered (see the OVP document *Simulation Control of Platforms and Modules User Guide* for more information about watchpoints).

Example

The OVP RISC-V model uses this function when returning from an exception:

```
inline static void setPC(riscvP riscv, Uns64 newPC) {
    vmirtSetPC((vmiProcessorP)riscv, newPC);
}

void riscvMRET(riscvP riscv) {

    Uns32     MPP     = RD_CSR_FIELD(riscv, mstatus, MPP);
    riscvMode minMode = riscvGetMinMode(riscv);
    riscvMode newMode = getERETMode(riscv, MPP, minMode);

    // clear any active exclusive access
    clearEA(riscv);

    // restore previous MIE
    WR_CSR_FIELD(riscv, mstatus, MIE, RD_CSR_FIELD(riscv, mstatus, MPIE))

    // MPIE=1
    WR_CSR_FIELD(riscv, mstatus, MPIE, 1);

    // MPP=<minimum_supported_mode>
    WR_CSR_FIELD(riscv, mstatus, MPP, minMode);

    // switch to target mode
    riscvSetMode(riscv, newMode);

    // jump to exception address
    setPC(riscv, RD_CSR_FIELD(riscv, mepc, value));

    // check for pending interrupts
    riscvTestInterrupt(riscv);
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPC` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*). Therefore, use `vmirtSetPC` only in

circumstances where the jump behavior cannot be described by morph time calls (for example, in exception handlers as above).

QuantumLeap Semantics
Non-self-synchronizing

12.4 vmirtSetPCDS

Prototype

```
void vmirtSetPCDS(
    vmiProcessorP processor,
    Addr          newPC1,
    Addr          newPC2,
    vmiPostSlotFn slotCB
);
```

Description

This routine sets the current program counter to `newPC1`. This effectively performs a *branch* operation from within a run time call. Once the instruction at `newPC1` has executed, control is immediately transferred to `newPC2`. When called from an embedded function call from translated native code, the branch to `newPC1` takes effect just after the return from the embedded function call. Any code after the embedded function call in the translated native code is not executed.

This function is typically used in processor models that support return from an exception handler directly to a delay slot instruction.

Example

This example is from the ARC processor model, which supports return from an exception handler directly to a delay slot instruction. In such cases, a status flag `status32.DE` is non-zero and the post-delay-slot address is in the `bta` register:

```
static void setPC(arcP arc, Uns32 pc) {

    vmiProcessorP processor = (vmiProcessorP)arc;
    Bool          DE        = AUX_FIELD(arc, status32, DE);
    Bool          ES        = AUX_FIELD(arc, status32, ES);

    // align target address and set other PC-related state
    pc = arcSetPCState(arc, pc);

    if(DE || ES) {

        // return to slot instruction
        vmirtSetPCDS(processor, pc, AUX_REG(arc, bta), 0);

        // indicate whether in execution slot (EI_S) or delay slot
        arc->ES = ES;

        // indicate whether in execution slot or delay slot
        arc->DE = ES || DE;

        // clear status32.DE and status32.ES (always derived)
        AUX_FIELD(arc, status32, DE) = 0;
        AUX_FIELD(arc, status32, ES) = 0;

    } else {

        // return to non-slot instruction
        vmirtSetPC(processor, pc);
    }
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPCDS` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*). Therefore, use `vmirtSetPCDS` only in circumstances where the jump behavior cannot be described by morph time calls (for example, in exception-like handlers as above).

QuantumLeap Semantics

Non-self-synchronizing

12.5 vmirtSetPCException

Prototype

```
void vmirtSetPCException(vmiProcessorP processor, Addr newPC);
```

Description

This routine sets the current program counter in order to take an exception. This effectively performs a *branch* operation from within a run time call. When called from an embedded function call from translated native code, the branch takes effect just after the return from the embedded function call. Any code after the embedded function call in the translated native code is not executed.

This function differs from `vmirtSetPC` in that any exception watchpoints installed for the processor will be triggered before the next simulated instruction is executed (see the OVP document *Simulation Control of Platforms and Modules User Guide* for more information about watchpoints).

Example

The OVP RISC-V model uses this function when modeling NMI exceptions:

```
static void doNMI(riscvP riscv) {

    // restart the processor from any halted state
    restartProcessor(riscv, RVD_RESTART_NMI);

    // switch to Machine mode
    riscvSetMode(riscv, RISCV_MODE_MACHINE);

    // update cause register (to zero)
    WR_CSR(riscv, mcause, 0);

    // update mepc to hold next instruction address
    WR_CSR(riscv, mepc, getPC(riscv));

    // indicate the taken exception
    riscv->exception = 0;

    // set address at which to execute
    vmirtSetPCException((vmiProcessorP)riscv, riscv->configInfo.nmi_address);
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPCException` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*).

QuantumLeap Semantics

Non-self-synchronizing

12.6 vmirtSetPCFlushTarget

Prototype

```
void vmirtSetPCFlushTarget(vmiProcessorP processor, Addr newPC);
```

Description

This routine is identical to `vmirtSetPC` except that it discards the target address in the current code dictionary before performing the jump. In other words, it forces code for the target address to be translated afresh using the morph time interface defined in the *VMI Morph Time Function Reference*.

This function is required in order to implement some classes of exception correctly. As an example, suppose that processor behavior is such that any pending exception is taken after the *next* instruction fetch and furthermore that an instruction generating such an exception is located in the middle of a sequence of instructions comprising a code block (see the *VMI Morph Time Function Reference* for a definition of a code block and how the simulator determines the instructions that are represented in each one). Suppose that, when executing a code block, an embedded model callback changes the system state so that there is now an exception that should be handled after the next instruction fetch. It would obviously be incorrect to do nothing in this case, as the simulator would then execute remaining instructions in the current code block instead of taking the exception. It would also be wrong to use `vmirtSetPC` to jump directly to the exception handler because that would not model the next instruction fetch that should be performed before the exception is taken. The solution is to use `vmirtSetPCFlushTarget` to jump to the *next* instruction. This will have the following effect:

1. The target code block (which is also the currently executing code block in this case) will be discarded. This means that no previously-translated instructions after the current instruction in this code block will be executed.
2. The processor program counter will be set to the next instruction address.
3. The processor execution exception handler (`ifetchExceptionCB` in the processor attribute structure) will be called for the next instruction address. This should call `vmirtSetPC` to reset the processor program counter to the appropriate simulated exception handler address (and perform any other required actions, such as entering kernel mode).

In this way it is possible to correctly model exceptions taken on the next instruction address. The example below outlines an embedded function call that might be inserted to perform the first of the three steps described above.

Example

```
static void vmic_TakeInterruptNext(cpuxP cpux) {

    if(exceptionPending(cpux)) {

        vmiProcessorP processor = (vmiProcessorP)cpux;
        Addr          thisPC    = vmirtGetPC(processor);
        Addr          nextPC    = getNextPC(thisPC);

        vmirtSetPCFlushTarget(processor, nextPC);
    }
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPCFlushTarget` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*). Therefore, use `vmirtSetPCFlushTarget` only in circumstances where the jump behavior cannot be described by morph time calls (for example, in exception handlers as described above).
2. `vmirtSetPCFlushTarget` is also less efficient than `vmirtSetPC` since it forces cached native code blocks to be discarded and retranslated.
   Only use `vmirtSetPCFlushTarget` when it is *mandatory* that the target is regenerated to get correct behavior

QuantumLeap Semantics

Synchronizing

12.7 vmirtSetPCFlushDict

Prototype

```
void vmirtSetPCFlushDict(vmiProcessorP processor, Addr newPC);
```

Description

This routine sets the current program counter and flushes all code blocks from the current dictionary. It is required when the processor has undergone some state change that would make any previously-generated code blocks in the current dictionary potentially invalid.

It is usually much more efficient to implement modal behavior of this kind by having multiple code dictionaries to represent different modes (see the section *Simulated Memory Access* elsewhere in this document for a detailed explanation of code dictionaries and their interaction with processor modes). However, it occasionally makes sense to implement mode changes by flushing a dictionary instead. An example might be simulation of a processor with a hardware mode that enables a trace buffer to record a sequence of recent branch addresses. In almost all normal operation, the trace buffer is disabled. In the very rare case that the trace buffer is enabled and must be simulated, every branch must be preceded in translated native code by an embedded call to simulate the buffer. In these circumstances, it is often more convenient to simply flush the current dictionary when the trace buffer is enabled or disabled by a processor state change instead of incurring the overhead of another dictionary to model the trace buffer.

Example

```
// embedded call made on status register (CPUX_SR) change
static void vmic_TestTraceModeChange(cpuxP cpux) {

    Bool simTraceEnabled = cpux->regs[CPUX_SR] & HW_TRACE_MODE;

    if(simTraceEnabled!=cpux->simTraceEnabled) {

        vmiProcessorP processor = (vmiProcessorP)cpux;
        Addr          thisPC    = vmirtGetPC(processor);
        Addr          nextPC    = getNextPC(thisPC);

        vmirtSetPCFlushDict((vmiProcessorP)cpux, nextPC);
        cpux->simTraceEnabled = simTraceEnabled;
    }
}
```

Notes and Restrictions

1. `vmirtSetPCFlushDict` is very slow because all code blocks in a dictionary must be discarded and retranslated. Use `vmirtSetPCFlushDict` only in very rare circumstances as described above.
2. See also the related routine `vmirtFlushDict`, which allows the dictionary to be flushed without changing the program counter.

QuantumLeap Semantics

Synchronizing

12.8 vmirtRegisterBranchNotifier

Prototype

```
void vmirtRegisterBranchNotifier(
    vmiProcessorP     processor,
    vmiBranchReasonFn notifier,
    void             *userData
);
```

Description

This routine registers a *branch notifier callback* function, which allows branch behavior of a program to be analyzed. Branch notifiers will typically be used either by models of performance monitor units or by external tools that implement branch analysis capabilities.

The `vmiBranchReasonFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_BRANCH_REASON_FN(_NAME) void _NAME( \
    vmiProcessorP   processor,  \
    Addr            prevPC,     \
    Addr            thisPC,     \
    vmiBranchReason reason,     \
    void           *userData    \
)
typedef VMI_BRANCH_REASON_FN((*vmiBranchReasonFn));
```

The callback is called on the instruction *after* every branch or potential branch. Argument `prevPC` is the program counter of the branch instruction and `thisPC` is the program counter of the current instruction. The callback is passed a reason for the branch, defined by an enumeration in `vmiTypes.h` as follows:

```
typedef enum vmiBranchReasonE {
    VMIBR_UNCOND,           // unconditional branch taken
    VMIBR_COND_TAKEN,       // conditional branch taken
    VMIBR_COND_NOT_TAKEN,   // conditional branch not taken
    VMIBR_PC_SET,           // PC set by non instruction route (e.g. exception)
    VMIBR_LAST,             // KEEP LAST
} vmiBranchReason;
```

Value `VMIBR_UNCOND` indicates that an unconditional branch has been taken.
Value `VMIBR_COND_TAKEN` indicates that a conditional branch has been taken.
Value `VMIBR_COND_NOT_TAKEN` indicates that a conditional branch has not been taken.
Value `VMIBR_PC_SET` indicates that a branch has been taken by some route that is not directly encoded in an instruction (for example, entry to or exit from an exception handler).

Example

This example shows how `vmirtRegisterBranchNotifier` might be used in a generic plugin to monitor branch behavior.

```
typedef struct vmiosObjectS {
    Uns64 counts[VMIBR_LAST];
    Bool  trace;
} vmiosObject;

//
// Branch notifier - add to appropriate count
//
static VMI_BRANCH_REASON_FN(notifier) {

    vmiosObjectP object = userData;

    // table mapping reason codes to strings
    static const char *reasonStrings[] = {
        [VMIBR_UNCOND]         = "unconditional branch taken",
        [VMIBR_COND_TAKEN]     = "conditional branch taken",
        [VMIBR_COND_NOT_TAKEN] = "conditional branch not taken",
        [VMIBR_PC_SET]         = "PC set by non instruction route",
    };

    // emit verbose debug if required
    if(object->trace) {
        vmiMessage("I", "BAV",
            "REASON (%s): %s (previous PC:0x"FMT_Ax" this PC:0x"FMT_Ax")\n",
            vmirtProcessorName(processor),
            reasonStrings[reason],
            prevPC,
            thisPC
        );
    }

    object->counts[reason]++;
}

//
// Constructor - register branch notifier callback
//
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmirtRegisterBranchNotifier(processor, notifier, object);
}

//
// Destructor - report counts
//
static VMIOS_DESTRUCTOR_FN(destructor) {

    // unregister the notifier
    vmirtUnregisterBranchNotifier(processor, notifier, object);

    // print statistics
    vmiMessage("I", "BAV", "----------------------------------------------------");
    vmiMessage("I", "BAV", "BRANCH ANALYSIS (%s)", vmirtProcessorName(processor));
    vmiMessage("I", "BAV", "----------------------------------------------------");
    vmiMessage("I", "BAV", "unconditional branch taken   : %s",
        getUns64CommaString(object->counts[VMIBR_UNCOND]));
    vmiMessage("I", "BAV", "conditional branch taken     : %s",
        getUns64CommaString(object->counts[VMIBR_COND_TAKEN]));
    vmiMessage("I", "BAV", "conditional branch not taken : %s",
        getUns64CommaString(object->counts[VMIBR_COND_NOT_TAKEN]));
    vmiMessage("I", "BAV", "other (e.g. exception)       : %s",
        getUns64CommaString(object->counts[VMIBR_PC_SET]));
}
```

When run on an ARM processor booting Linux with tracing enabled, output like the following is produced:

```
Info 839115688: 'cpu_CPU0', 0x0000000076f3abe4: e3530000 cmp     r3,#0
Info 839115689: 'cpu_CPU0', 0x0000000076f3abe8: 0affffd5 beq     76f3ab44
Info (BAV) REASON (cpu_CPU0): conditional branch taken (previous PC:0x76f3abe8 this
PC:0x76f3ab44)
```

```
Info 839115690: 'cpu_CPU0', 0x0000000076f3ab44: e3a00000 mov     r0,#0
Info 839115691: 'cpu_CPU0', 0x0000000076f3ab48: e8bd8070 ldm     sp!,{r4,r5,r6,pc}
Info (BAV) REASON (cpu_CPU0): unconditional branch taken (previous PC:0x76f3ab48
this PC:0x76f340ec)
Info 839115692: 'cpu_CPU0', 0x0000000076f340ec: e3500000 cmp     r0,#0
Info 839115693: 'cpu_CPU0', 0x0000000076f340f0: 1a00003c bne     76f341e8
Info (BAV) REASON (cpu_CPU0): conditional branch not taken (previous PC:0x76f340f0
this PC:0x76f340f4)
Info 839115694: 'cpu_CPU0', 0x0000000076f340f4: e1a0c004 mov     ip,r4
Info 839115695: 'cpu_CPU0', 0x0000000076f340f8: e59f2c1c ldr     r2,[pc,#3100]
Info 839115696: 'cpu_CPU0', 0x0000000076f340fc: e24b304c sub     r3,fp,#76
Info 839115697: 'cpu_CPU0', 0x0000000076f34100: e24b0034 sub     r0,fp,#52
Info 839115698: 'cpu_CPU0', 0x0000000076f34104: e08a2002 add     r2,sl,r2
Info 839115699: 'cpu_CPU0', 0x0000000076f34108: e24b1030 sub     r1,fp,#48
Info 839115700: 'cpu_CPU0', 0x0000000076f3410c: e50bc03c str     ip,[fp,#-60]
Info 839115701: 'cpu_CPU0', 0x0000000076f34110: eb0003f2 bl      76f350e0
Info (BAV) REASON (cpu_CPU0): unconditional branch taken (previous PC:0x76f34110
this PC:0x76f350e0)
Info 839115702: 'cpu_CPU0', 0x0000000076f350e0: *** FETCH EXCEPTION ***
Info (BAV) REASON (cpu_CPU0): PC set by non instruction route (previous
PC:0x76f350e0 this PC:0xffff000c)
Info 839115703: 'cpu_CPU0', 0x00000000ffff000c: ea0000bb b       ffff0300
Info (BAV) REASON (cpu_CPU0): unconditional branch taken (previous PC:0xffff000c
this PC:0xffff0300)
Info 839115704: 'cpu_CPU0', 0x00000000ffff0300: e24ee004 sub     lr,lr,#4
```

Notes and Restrictions

1. `vmirtRegisterBranchNotifier` is very slow because all code blocks in a dictionary must be discarded and retranslated when notifiers are installed or uninstalled. Avoid calling this function repeatedly if possible.

QuantumLeap Semantics

Synchronizing

12.9 vmirtUnregisterBranchNotifier

Prototype

```
void vmirtUnregisterBranchNotifier(
    vmiProcessorP     processor,
    vmiBranchReasonFn notifier,
    void             *userData
);
```

Description

This routine unregisters a previously-registered *branch notifier callback* function, which allows branch behavior of a program to be analyzed. A notifier will be unregistered only if the `userData` matches the data supplied when it was installed.

Example

See `vmirtRegisterBranchNotifier` for an example.

Notes and Restrictions

1. `vmirtUnregisterBranchNotifier` is very slow because all code blocks in a dictionary must be discarded and retranslated when notifiers are installed or uninstalled. Avoid calling this function repeatedly if possible.

QuantumLeap Semantics

Synchronizing

13 Processor Modes

Processor models can be modal – for example, a processor may support both *user* and *kernel* modes, in which some instructions behave differently, or in which memory access permissions have different restrictions. Modal models can be written in three different ways:

1. All behavior that is mode-dependent can be modeled explicitly in the morphed code emitted for each instruction. For example, code for a privileged instruction might explicitly test a processor mode bit, and, depending on the value of that bit, either implement the privileged behavior or implement a jump to an exception vector. This approach is not recommended because the generated code is usually much more verbose than necessary and contains many more branches than necessary, both of which badly degrade performance.
2. The model can be designed to support multiple *code dictionaries*, one for each mode, with calls to `vmirtSetMode` (documented in this section) to switch modes on processor state changes. This is the recommended method for implementing modal code in most cases. See the *OVP Processor Modeling Guide* for extensive examples of the use of processor modes.
3. In cases where there are only minor differences in behavior between what would otherwise be different modes, and where it is very likely that translated code will always be executed with the processor in the *same* mode, block masks can be used to implement modal instructions without the cost of additional dictionaries and mode switches.

The concept of *processor scope* is also defined. A processor scope is a mode/ASID combination that can (for example) be used to identify a specific process running in user model in a simulated Linux. This scope information can be used with functions `vmiosUpdateScope`, `vmiosGetScope` and `vmiosMatchScope` to constrain the scope if intercept libraries or plugins – see the *VMI OS Support Function Reference* for more information.

13.1 vmirtSetMode

Prototype

```
Bool vmirtSetMode(vmiProcessorP processor, Uns32 dictionaryIndex);
```

Description

Processors are often modal – for example, many have *user* and *kernel* modes in which instruction implementations are different. Register writes may succeed in kernel mode but cause an instruction privilege trap in user mode. In addition, virtual-to-physical memory maps or access permissions often differ between modes.

The simulator allows such modal processors to be simulated efficiently by supporting multiple *dictionary modes*. Each dictionary mode has an associated *dictionary* of JIT-translated code blocks and *memory domain* defining virtual-to-physical address translations. The currently-active dictionary mode can be changed by calling `vmirtSetMode` to indicate the new code dictionary and associated memory domain to be used. This is explained in detail in the section *Simulated Memory Management* elsewhere in this document.

The argument `dictionaryIndex` is an index number of the new dictionary mode. Valid index numbers for a processor are defined by the processor attribute structure: see *Simulated Memory Management* elsewhere in this document for more information about this.

The new code dictionary will be used when translating the next instruction executed after the call to `vmirtSetMode`.

Example

The OVP RISC-V model uses this function when modeling mode changes.:

```
void riscvSetMode(riscvP riscv, riscvMode mode) {

    riscvDMode dMode = mode;

    // if executing in supervisor or user mode, include VM-enabled indication
    if((mode<=RISCV_MODE_SUPERVISOR) && (RD_CSR_FIELD(riscv, satp, MODE))) {
        dMode |= RISCV_DMODE_VM;
    } else {
        dMode &= ~RISCV_DMODE_VM;
    }

    // update mode if it has changed
    if(riscv->mode != dMode) {
        vmirtSetMode((vmiProcessorP)riscv, dMode);
        riscv->mode = dMode;
    }

    // refresh current data domain (may be modified by mstatus.MPRV, and may
    // have changed while taking an exception even if mode has not changed)
    riscvVMRefreshMPRVDomain(riscv);
}
```

Notes and Restrictions

1. The new dictionary will be used for the next simulated instruction after the current instruction. Deferring the mode change by more than one instruction is not supported.
2. If the mode switch succeeded, the function returns `True`. If the processor does not support the indexed mode, the function returns `False` and the processor state is not updated.
3. The mapping between *processor modes* as defined in the processor specification and *dictionary modes* used by the simulator is not necessarily one-to-one. For example, it could be the case that the memory map used in a single *processor-defined* mode may change greatly when configuration registers change: in the RISC-V processor, enabling or disabling address translation is a specific example. In such a case, a single processor-defined mode is best simulated using *two* dictionary modes (a translation-enabled mode and a translation-disabled mode).
4. To access processor-defined modes, see functions `vmirtGetCurrentMode` and `vmirtGetNextMode`.

QuantumLeap Semantics
Non-self-synchronizing

13.2 vmirtGetMode

Prototype

```
Uns32 vmirtGetMode(vmiProcessorP processor);
```

Description

Processors are often modal – for example, many have *user* and *kernel* modes in which instruction implementations are different. Register writes may succeed in kernel mode but cause an instruction privilege trap in user mode. In addition, virtual-to-physical memory maps or access permissions often differ between modes.

The simulator allows such modal processors to be simulated efficiently by supporting multiple *dictionary modes*. Each dictionary mode has an associated *dictionary* of JIT-translated code blocks and *memory domain* defining virtual-to-physical address translations.

This function returns the current processor *dictionary index*. The dictionary index can be changed using related function `vmirtSetMode`.

Example

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

Notes and Restrictions

1. The mapping between *processor modes* as defined in the processor specification and *dictionary modes* used by the simulator is not necessarily one-to-one (see notes associated with function `vmirtSetMode` for more information).
2. To access processor-defined modes, see functions `vmirtGetCurrentMode` and `vmirtGetNextMode`.

QuantumLeap Semantics

Non-self-synchronizing

13.3 vmirtSetBlockMask

Prototype

```
void vmirtSetBlockMask(vmiProcessorP processor, Uns32 blockMask);
```

Description

In cases where there are only minor differences in behavior between what would otherwise be different modes, and where it is very likely that translated code will always be executed with the processor in the *same* mode, block masks can be used to implement modal instructions without the cost of additional dictionaries and mode switches.

A processor has a notion of a *current block mask*, which is an `Uns32` value. The current block mask is set by `vmirtSetBlockMask`.

When creating JIT-translated code using the Imperas Morph Time Function interface, code can be emitted to validate the current block mask using function `vmimtValidateBlockMask`, which has the following prototype:

```
void vmimtValidateBlockMask(Uns32 modeMask);
```

This causes code to be inserted in the JIT compiled block with the following effect:

1. A subset of bits in the processor block mask is selected by bitwise-AND with the specified `modeMask`.
2. If the result of the bitwise-AND is *the same as* the same bits selected from the block mask that was in effect when the code block was created, the code block is executed and there is no further special action.
3. If the result *is different to* the same bits selected from the block mask that was in effect when the code block was created, the code block is deleted and retranslated.

In cases where code blocks are almost always executed with the block mask that was in effect when they were created, this results in very efficient simulation without the cost of an additional code dictionary that might otherwise be required.

Example

The standard ARM processor model uses block masks to control code generated for `LDM` and `STM` instructions that access user-mode registers from other modes. The model is designed so that, in each processor mode, registers r0 to r15 for that mode are always held in the current register bank, and registers not normally visible in that mode are held in subsidiary model registers. The model is written so that it accesses either the normal or subsidiary registers as applicable for the current processor mode. It uses a `blockMask` to ensure that the assumption made about the current mode when code was generated is still valid when the code is executed.

In `armMorph.c`, function `getBankedReg` returns a `vmiReg` structure for a register, using `vmimtValidateBlockMask` to check that the processor is in the correct mode (function `armEmitValidateBlockMask` is a wrapper round `vmimtValidateBlockMask`):

```
// Macro returning base register (and validating block mode)
#define GET_BASE_REG(_BM) \
    armEmitValidateBlockMask(_BM);  \
    return VMI_NOREG

// Macro returning banked register (and validating block mode)
#define GET_BANKED_REG(_BM, _N, _SET) \
    armEmitValidateBlockMask(_BM);  \
    return ARM_BANK_REG(_N, _SET)

// If 'userRegs' is True and the register index specifies a banked register,
// return a vmiReg structure for the banked register; otherwise, return
// VMI_NOREG. Also, validate the current block mode for registers r8 to r14.
static vmiReg getBankedReg(armMorphStateP state, Uns32 r, Bool userRegs) {

    if(!userRegs) {

        return VMI_NOREG;

    } else switch(state->arm->CPSR.fields.mode) {

        case ARM_CPSR_FIQ:
            switch(r) {
                case 8:  GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       8,  fiq);
                case 9:  GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       9,  fiq);
                case 10: GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       10, fiq);
                case 11: GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       11, fiq);
                case 12: GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       12, fiq);
                case 13: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_FIQ, 13, fiq);
                case 14: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_FIQ, 14, fiq);
                default: return VMI_NOREG;
            }
            break;

        case ARM_CPSR_IRQ:
            switch(r) {
                case 8:  GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 9:  GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 10: GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 11: GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 12: GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 13: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_IRQ, 13, irq);
                case 14: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_IRQ, 14, irq);
                default: return VMI_NOREG;
            }
            break;

        … code for remaining modes follows …
    }
}
```

Function `armSetBlockMask` in file `armUtils.c` is called to set the current block mask on a mode switch:

```
// Return current processor block mask
static armBlockMask getBlockMask(armP arm) {
    switch(arm->CPSR.fields.mode) {
        case ARM_CPSR_USER:       return ARM_BM_USER;
        case ARM_CPSR_FIQ:        return ARM_BM_FIQ;
        case ARM_CPSR_IRQ:        return ARM_BM_IRQ;
        case ARM_CPSR_SUPERVISOR: return ARM_BM_SUPERVISOR;
        case ARM_CPSR_ABORT:      return ARM_BM_ABORT;
        case ARM_CPSR_UNDEFINED:  return ARM_BM_UNDEFINED;
        case ARM_CPSR_SYSTEM:     return ARM_BM_SYSTEM;
        default:                  return ARM_BM_USER;
    }
}

// Update processor block mask
void armSetBlockMask(armP arm) {
    vmirtSetBlockMask((vmiProcessorP)arm, getBlockMask(arm));
}
```

The `blockMask` values are specified by a `typedef` in file `armMode.h`:

```
typedef enum armBlockMaskE {

    ARM_BM_R8_R12_BASE       = 0x01,
    ARM_BM_R8_R12_FIQ        = 0x02,
    ARM_BM_R13_R14_BASE      = 0x04,
    ARM_BM_R13_R14_SPSR_FIQ = 0x08,
    ARM_BM_R13_R14_SPSR_IRQ = 0x10,
    ARM_BM_R13_R14_SPSR_SVC = 0x20,
    ARM_BM_R13_R14_SPSR_ABT = 0x40,
    ARM_BM_R13_R14_SPSR_UND = 0x80,

    ARM_BM_USER       = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_BASE,
    ARM_BM_SYSTEM     = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_BASE,
    ARM_BM_FIQ        = ARM_BM_R8_R12_FIQ  | ARM_BM_R13_R14_SPSR_FIQ,
    ARM_BM_IRQ        = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_IRQ,
    ARM_BM_SUPERVISOR = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_SVC,
    ARM_BM_ABORT      = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_ABT,
    ARM_BM_UNDEFINED  = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_UND

} armBlockMask;
```

Notes and Restrictions

1. Since a block mask match failure causes a code block to be deleted and retranslated, block masks can make simulation much slower if this often happens. They should therefore only be used in cases where it can be inferred that mismatches are very unlikely to occur.
2. As of VMI version 6.41.0, functions `vmirtSetBlockMask32` and `vmirtSetBlockMask64` have been added. These functions allow block masks in general 32-bit and 64-bit processor fields to be updated.

QuantumLeap Semantics

Non-self-synchronizing

13.4 vmirtSetBlockMask32

Prototype

```
void vmirtSetBlockMask32(
    vmiProcessorP processor,
    Uns32         *blockMaskP,
    Uns32          blockMask
)
```

Description

This function allows a block mask in a general 32-bit processor field to be modified. It is in effect exactly equivalent to function `vmirtSetBlockMask` except that the block mask can be any processor field and not just the single built-in 32-bit block mask. See section 13.3 for a full description of the purpose and use of block masks.

When creating JIT-translated code using the Imperas Morph Time Function interface, code can be emitted to validate the block mask in a general 32-bit processor field using function `vmimtValidateBlockMaskR`, which has the following prototype:

```
void vmimtValidateBlockMaskR(Uns32 bits, vmiReg r, Uns64 modeMask);
```

This causes code to be inserted in the JIT compiled block with the following effect:

1. A subset of bits in the block mask contained in register `r` is selected by bitwise-AND with the specified `modeMask`. When operating on a 32-bit block mask, `bits` should be 32 and the most significant half of `modeMask` should be zero.
2. If the result of the bitwise-AND is *the same as* the same bits selected from the block mask that was in effect when the code block was created, the code block is executed and there is no further special action.
3. If the result *is different to* the same bits selected from the block mask that was in effect when the code block was created, the code block is deleted and retranslated.

Example

The ARM processor model example described in section 13.3 could equivalently have been written using `vmirtSetBlockMask32` and `vmimtValidateBlockMaskR` as follows:

```
typedef struct armS {
    . . . fields omitted . . .
    Uns32 blockMask;            // block mask in processor structure
} arm, *armP;

// morph-time macro to calculate offset to block mask in an arm structure
#define ARM_CPU_REG(_F)         VMI_CPU_REG(armP, _F)
#define ARM_BLOCK_MASK          ARM_CPU_REG(blockMask)

// Update processor block mask
void armSetBlockMask(armP arm) {
    vmirtSetBlockMask32((vmiProcessorP)arm, &arm->blockMask, getBlockMask(arm));
}

// Emit code to validate block mask bits selected by the given modeMask
static void validateBlockMaskR(armBlockMask modeMask) {
    vmimtValidateBlockMaskR(32, ARM_BLOCK_MASK, modeMask);
}
```

Notes and Restrictions

1. Since a block mask match failure causes a code block to be deleted and retranslated, block masks can make simulation much slower if this often happens. They should therefore only be used in cases where it can be inferred that mismatches are very unlikely to occur.

QuantumLeap Semantics

Non-self-synchronizing

13.5 vmirtSetBlockMask64

Prototype

```
void vmirtSetBlockMask64(
    vmiProcessorP processor,
    Uns64         *blockMaskP,
    Uns64          blockMask
)
```

Description

This function allows a block mask in a general 64-bit processor field to be modified. It is in effect exactly equivalent to function `vmirtSetBlockMask32` except that the block mask field is 64 bits instead of 32. See section 13.3 for a full description of the purpose and use of block masks.

When creating JIT-translated code using the Imperas Morph Time Function interface, code can be emitted to validate the block mask in a general 64-bit processor field using function `vmimtValidateBlockMaskR`, which has the following prototype:

```
void vmimtValidateBlockMaskR(Uns32 bits, vmiReg r, Uns64 modeMask);
```

This causes code to be inserted in the JIT compiled block with the following effect:

1. A subset of bits in the block mask contained in register `r` is selected by bitwise-AND with the specified `modeMask`. When operating on a 64-bit block mask, `bits` should be 64.
2. If the result of the bitwise-AND is *the same as* the same bits selected from the block mask that was in effect when the code block was created, the code block is executed and there is no further special action.
3. If the result *is different to* the same bits selected from the block mask that was in effect when the code block was created, the code block is deleted and retranslated.

Example

The ARM processor model example described in section 13.3 could equivalently have been written using `vmirtSetBlockMask64` and `vmimtValidateBlockMaskR` as follows:

```
typedef struct armS {
    . . . fields omitted . . .
    Uns64 blockMask;            // block mask in processor structure
} arm, *armP;

// morph-time macro to calculate offset to block mask in an arm structure
#define ARM_CPU_REG(_F)         VMI_CPU_REG(armP, _F)
#define ARM_BLOCK_MASK          ARM_CPU_REG(blockMask)

// Update processor block mask
void armSetBlockMask(armP arm) {
    vmirtSetBlockMask64((vmiProcessorP)arm, &arm->blockMask, getBlockMask(arm));
}

// Emit code to validate block mask bits selected by the given modeMask
static void validateBlockMaskR(armBlockMask modeMask) {
    vmimtValidateBlockMaskR(64, ARM_BLOCK_MASK, modeMask);
}
```

Notes and Restrictions

1. Since a block mask match failure causes a code block to be deleted and retranslated, block masks can make simulation much slower if this often happens. They should therefore only be used in cases where it can be inferred that mismatches are very unlikely to occur.

QuantumLeap Semantics

Non-self-synchronizing

13.6 vmirtGetBlockMask

Prototype

```
Uns32 vmirtGetBlockMask(vmiProcessorP processor);
```

Description

This function returns the current processor block mask. The block mask can be changed using related function `vmirtSetBlockMask`.

Example

The OVP MICROBLAZE processor model uses this function when updating the current mode and block mask:

```
void microblazeUpdateBlockMask(microblazeP microblaze) {

    microblazeUpdateFPControlWord(microblaze);

    vmirtSetMode((vmiProcessorP)microblaze, microblazeGetVMMode(microblaze));

    Uns32 BM = vmirtGetBlockMask((vmiProcessorP)microblaze);

    //
    // update simulated processor state for Exception Enable
    //
    if (microblaze->SPR_MSR.EE) {
        BM |= BM_MSR_EE;
    } else {
        BM &= ~BM_MSR_EE;
    }

    //
    // update simulated processor state for User Mode
    //
    if (microblaze->SPR_MSR.UM) {
        BM |= BM_MSR_UM;
    } else {
        BM &= ~BM_MSR_UM;
    }

    vmirtSetBlockMask((vmiProcessorP)microblaze, BM);

    microblazeCheckExceptions(microblaze);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

13.7 vmirtGetProcessorScope

Prototype

```
const char *vmirtGetProcessorScope(vmiProcessorP processor);
```

Description

This function returns the current processor scope as a string. The scope is a string of the form:

```
mode:ASID
```

where mode is a model-specific mode code and ASID is a model-specific ASID specifier. The ASID specifier may combine traditional ASID (address space identifier) with other information, for example a VMID (virtual machine identifier) in processor models that support Hypervisors.

This function is intended for use with OS support functions `vmiosUpdateScope`, `vmiosGetScope` and `vmiosMatchScope` to constrain the use of intercept libraries and other plugins – see the *VMI OS Support Function Reference* for more information.

Example

This example shows how scopes might be used to constrain an intercept library callback that monitors a function in a Linux user application so that it is activated only when one particular process runs. The example is somewhat contrived because it constrains the intercept library so that it is tied to the *first user process that executes at the intercepted function address*, whether or not this is in fact running the target application. In reality, the scope constraint would be specified by an OS-aware helper library that monitored the creation and deletion of user tasks.

```
#include “vmi/vmiRt.h”
#include “vmi/vmiOSLib.h”

static VMIOS_INTERCEPT_FN(doFib) {

    Uns32 index;

    getArg(processor, object, 0, &index);

    if(index>=30) {

        const char *context = vmirtGetProcessorScope(processor);

        // constrain intercept library to one scope
        if(!object->initialized) {
            object->initialized = True;
            vmiosUpdateScope(object, context);
            vmiMessage(
                "I", CPU_PREFIX "_SCOPE",
                "set intercept scope %s",
                vmiosGetScope(object)
            );
        }

        if(vmiosMatchScope(processor, object)) {
            // matching scope
            vmiMessage(
                "I", CPU_PREFIX "_MATCH",
                "(scope %s): index=%u",
                context,
                index
            );

        } else {

            // mismatched scope
            vmiMessage(
                "I", CPU_PREFIX "_IGNRE",
                "(scope %s ignored)",
                Context
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14 Floating Point Operations

VMI floating point operation support has been extensively revised with version 5.13.0. A detailed description of this enhanced support can be found in the *VMI Morph Time Function Reference*.

This section describes the functions that are used to configure a model floating point unit. Refer to the *VMI Morph Time Function Reference* for all other details of floating point instruction simulation.

14.1 vmirtConfigureFPU

Prototype

```
void vmirtConfigureFPU(
    vmiProcessorP processor,
    vmiFPConfigCP config
);
```

Description

This function is used to specify a default configuration for floating point operations for which no instruction-specific configuration has been specified. 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;
    Uns8                perElementFlags;
    vmiFPFlagForce      forceAHP   : 2;
    vmiFPFlagForce      forceFZ16  : 2;
    vmiFPFlagForce      forceDAZ16 : 2;
} vmiFPConfig;
```

The native floating point operations described in the *VMI Morph Time Function Reference* normally follow exact IEEE Standard 754 or Intel x87 semantics, depending on the floating point operand type. This function allows the FPU for a processor model to be configured to be non-compliant in some respects, so that the native floating point operation code may be used even when a processor model diverges from these standards in some respects.

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. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
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. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
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. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
7. `indeterminateUns8` specifies the bit pattern produced when a floating point operation generates a 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 a 8-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.

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. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
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. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
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. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
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). See the *VMI Morph Time Function Reference* for more information about tiny result handlers.
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). See the *VMI Morph Time Function Reference* for more information about tiny argument handlers.
17. `fp8ResultCB` is a callback function which, if given, is called whenever a 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`. See the *VMI Morph Time Function Reference* for more information about result handlers.
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`. See the *VMI Morph Time Function Reference* for more information about result handlers.
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`. See the *VMI Morph Time Function Reference* for more information about result handlers.
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`. See the *VMI Morph Time Function Reference* for more information about result handlers.
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. See the *VMI Morph Time Function Reference* for more information about the exception handler callback.

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`[11] 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` in the *VMI Morph Time Function Reference* 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.

[11] 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.

Example

The OVP RISC-V model uses this function to initialize its state. The processor implements a set of special configurations for min/max operations and a normal configuration that is used in all other cases.

```
const static vmiFPConfig fpConfigs[RVFP_LAST] = {
    [RVFP_NORMAL]   = FPU_CONFIG(
        handleQNaN32, handleQNaN64, 0,            0
    ),
    [RVFP_FMIN]     = FPU_CONFIG(
        0,            0,            doFMin32_2_2, doFMin64_2_2
    ),
    [RVFP_FMAX]     = FPU_CONFIG(
        0,            0,            doFMax32_2_2, doFMax64_2_2
     ),
    [RVFP_FMIN_2_3] = FPU_CONFIG(
        0,            0,            doFMin32_2_3, doFMin64_2_3
    ),
    [RVFP_FMAX_2_3] = FPU_CONFIG(
        0,            0,            doFMax32_2_3, doFMax64_2_3
    ),
};

void riscvConfigureFPU(riscvP riscv) {
    vmirtConfigureFPU((vmiProcessorP)riscv, &fpConfigs[RVFP_NORMAL]);
}
```

Notes and Restrictions

1. This function is intended to be called once only in the processor constructor. It flushes all processor code dictionaries to reflect the new FPU configuration. *Calling the function frequently during simulation will severely degrade performance.*
2. The default configuration can be overridden on an instruction-specific basis. See the *VMI Morph Time Function Reference* for more information.

QuantumLeap Semantics

Non-self-synchronizing

14.2 vmirtGetFPControlWord

Prototype

```
vmiFPControlWord vmirtGetFPControlWord(vmiProcessorP processor);
```

Description

This function returns the currently-active floating point control word for the processor. This is specified by a structure 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 standard 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.

The `RC` field specifies the *rounding control* to use when arithmetic results cannot be exactly represented. 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 most processors implement variations of these options.

Fields `AHP`, `FZ16` and `DAZ16` control the behavior of operations using 16-bit floating point types. If `AHP` is 1, then operations on 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.

Example

The OVP MIPS model uses this function to when emulating denormals-are-zero behavior:

```
static void emulateBinopDAZ32(vmiProcessorP processor, vmiFPArgP args) {

    vmiFPControlWord cw = vmirtGetFPControlWord(processor);

    // handle denormal inputs
    if(cw.DAZ && isDenorm32(args[0].u32)) {
        args[0].u32 &= MIPS_SIGN_32;
    }
    if(cw.DAZ && isDenorm32(args[1].u32)) {
        args[1].u32 &= MIPS_SIGN_32;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.3 vmirtSetFPControlWord

Prototype

```
void vmirtSetFPControlWord(
    vmiProcessorP    processor,
    vmiFPControlWord fpcw
);
```

Description

This function sets the currently-active floating point control word for the processor. This is specified by a structure in `vmiTypes.h`; see `vmirtGetFPControlWord` for details.

Example

The OVP RISC-V model uses this function when updating the floating point control word:

```
static void setFPCW(riscvP riscv, vmiFPRC rc) {

    . . . lines omitted for clarity . . .

    // use the specified rounding control (all exceptions disabled)
    vmiFPControlWord cw = {
        .IM = 1,
        .DM = 1,
        .ZM = 1,
        .OM = 1,
        .UM = 1,
        .PM = 1,
        .RC = rc
    };

    vmirtSetFPControlWord((vmiProcessorP)riscv, cw);
}
```

Notes and Restrictions

None

QuantumLeap Semantics

Non-self-synchronizing

14.4 vmirtRestoreFPState

Prototype

```
void vmirtRestoreFPState(vmiProcessorP _unused);
```

Description

The simulator core uses native floating point instructions to implement most floating point operations (see the *VMI Morph Time Function Reference* guide for more details on these operations). As these instructions execute, the native floating point state is modified to reflect exceptions and rounding modes required to implement that operation. As a consequence, the native floating point state seen in an embedded call is by default dependent on any floating point operations that the processor has executed prior to the embedded call.

This function can be used in an embedded call to restore the native floating point state to the state when the simulator was invoked. Use this function prior to any code in an embedded call that uses floating point instructions to avoid unexpected behavior.

Example

The OVP ARM model uses this function before executing code in an embedded call implementing reciprocal estimation:

```
static Flt64 recipEstimate(armP arm, Flt64 a) {

    Int32 q, s;
    Flt64 r;

    // reset floating point state before operating on floating-point values
    vmirtRestoreFPState(0);

    q = (Int32)(a * 512.0);
    r = 1.0 / (((Flt64) q + 0.5) / 512.0);
    s = (Int32)(256.0 * r + 0.5);

    return ((Flt64) s / 256.0);
}
```

Notes and Restrictions

1. The `_unused` argument is present for backwards-compatibility only and is not used.

QuantumLeap Semantics

Thread safe

14.5 vmirtSetSIMDMaxUnroll

Prototype

```
void vmirtSetSIMDMaxUnroll(vmiProcessorP processor, Uns32 maxUnroll);
```

Description

This function controls the native code that is generated for floating point SIMD operations (defined using functions such as `vmimtFUnopSimdRR`). By default, and for VMI versions prior to 7.4.0, such operations always generate code that *unrolls* the SIMD loop into separate operations. The code is usually fast, but can get large, especially for larger SIMD widths (with 4, 8 or more parallel operations).

This function allows an upper limit on the number of unrolled operations to be set. When the number of parallel operations is less than or equal to `maxUnroll`, the operation will be unrolled; when it exceeds `maxUnroll`, the operation will instead be implemented as a loop. The loop implementation is usually much smaller, especially for wider operations, meaning that for SIMD-intense applications the JIT code dictionary is used more efficiently.

Example

The OVP ARM model uses this function when configuring floating point operations:

```
//
// Call on initialization
//
void armFPInitialize(armP arm) {

    . . . lines omitted for clarity . . .

    vmiProcessorP processor = (vmiProcessorP)arm;

    // specify default floating point configuration
    arm->defaultFPConfig = getDefaultFPConfig(arm);
    vmirtConfigureFPU(processor, arm->defaultFPConfig);

    // specify SIMD unroll limit for VMI intrinsics
    vmirtSetSIMDMaxUnroll(processor, MAX_SIMD_UNROLL(arm));
}
```

Notes and Restrictions

1. This function affects performance and JIT code size only; simulation behavior is otherwise identical for unrolled and loop implementations. See chapter 28 of the *OVP Processor Modeling Guide* for information about obtaining translation efficiency statistics from a model in order to judge whether looped SIMD operations are beneficial.

QuantumLeap Semantics

Non-self-synchronizing

14.6 vmirtGetFConvertRRDesc

Prototype

```
vmiFPConvertDescCP vmirtGetFConvertRRDesc(
    vmiProcessorP processor,
    Uns32         num,
    vmiFType      destType,
    vmiFType      srcType,
    vmiFPConfigCP config,
    vmiFPRC       rc,
    Bool          inCompound
);
```

Description

This function returns a descriptor object that can be passed as an argument to a later call to `vmirtFConvertSimdRR`. The descriptor object describes the operation of `num` parallel floating point conversions from type `srcType` to type `destType`. Argument `rc` indicates the rounding mode to use. Argument `config` contains detailed information about the semantics of the operation itself: see the *VMI Morph Time Function Reference* for detailed description of this structure.

Argument `inCompound` indicates whether the operation is part of a *compound operation*. If the argument is `False`, then any enabled exceptions resulting from evaluation of the conversion will cause the processor to take a floating point exception on return from the embedded call. If the argument is `True`, then any exceptions resulting from the operation will not cause a floating point exception to be taken, but will instead update the flag register passed to `vmirtFConvertSimdRR` using bitwise-or. Compound operations thus allow several run-time floating point primitives to be combined in a single embedded call, at the end of which the composite state can be used to determine whether an exception should be signaled or not.

The descriptor returned by this function is persistent during a simulation session. This means that it can be cached for later use if required.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.7 vmirtGetFUnopRRDesc

Prototype

```
vmiFPUnopDescCP vmirtGetFUnopRRDesc(
    vmiProcessorP processor,
    vmiFType      type,
    Uns32         num,
    vmiFUnop      op,
    vmiFPConfigCP config,
    vmiFPRC       rc,
    Bool          inCompound
);
```

Description

This function returns a descriptor object that can be passed as an argument to a later call to `vmirtFUnopSimdRR`. The descriptor object describes the operation of `num` parallel floating point unary operations operating on arguments of type `type`. Argument `rc` indicates the rounding mode to use. Argument `config` contains detailed information about the semantics of the operation itself: see the *VMI Morph Time Function Reference* for detailed description of this structure.

Argument `inCompound` indicates whether the operation is part of a *compound operation*. If the argument is `False`, then any enabled exceptions resulting from evaluation of the conversion will cause the processor to take a floating point exception on return from the embedded call. If the argument is `True`, then any exceptions resulting from the operation will not cause a floating point exception to be taken, but will instead update the flag register passed to `vmirtFUnopSimdRR` using bitwise-or. Compound operations thus allow several run-time floating point primitives to be combined in a single embedded call, at the end of which the composite state can be used to determine whether an exception should be signaled or not.

The descriptor returned by this function is persistent during a simulation session. This means that it can be cached for later use if required.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.8 vmirtGetFBinopRRRDesc

Prototype

```
vmiFPBinopDescCP vmirtGetFBinopRRRDesc(
    vmiProcessorP processor,
    vmiFType      type,
    Uns32         num,
    vmiFBinop     op,
    vmiFPConfigCP config,
    vmiFPRC       rc,
    Bool          inCompound
);
```

Description

This function returns a descriptor object that can be passed as an argument to a later call to `vmirtFBinopSimdRRR`. The descriptor object describes the operation of `num` parallel floating point binary operations operating on arguments of type `type`. Argument `rc` indicates the rounding mode to use. Argument `config` contains detailed information about the semantics of the operation itself: see the *VMI Morph Time Function Reference* for detailed description of this structure.

Argument `inCompound` indicates whether the operation is part of a *compound operation*. If the argument is `False`, then any enabled exceptions resulting from evaluation of the conversion will cause the processor to take a floating point exception on return from the embedded call. If the argument is `True`, then any exceptions resulting from the operation will not cause a floating point exception to be taken, but will instead update the flag register passed to `vmirtFBinopSimdRRR` using bitwise-or. Compound operations thus allow several run-time floating point primitives to be combined in a single embedded call, at the end of which the composite state can be used to determine whether an exception should be signaled or not.

The descriptor returned by this function is persistent during a simulation session. This means that it can be cached for later use if required.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.9 vmirtGetFTernopRRRRDesc

Prototype

```
vmiFPTernopDescCP vmirtGetFTernopRRRRDesc(
    vmiProcessorP processor,
    vmiFType      type,
    Uns32         num,
    vmiFTernop    op,
    vmiFPConfigCP config,
    vmiFPRC       rc,
    Bool          roundInt,
    Bool          inCompound
);
```

Description

This function returns a descriptor object that can be passed as an argument to a later call to `vmirtFTernopSimdRRRR`. The descriptor object describes the operation of `num` parallel floating point trinary operations operating on arguments of type `type`. Argument `rc` indicates the rounding mode to use. Argument `config` contains detailed information about the semantics of the operation itself: see the *VMI Morph Time Function Reference* for detailed description of this structure. Argument `roundInt` indicates whether intermediates should be rounded (if `True`) or whether the operation is fused (if `False`).

Argument `inCompound` indicates whether the operation is part of a *compound operation*. If the argument is `False`, then any enabled exceptions resulting from evaluation of the conversion will cause the processor to take a floating point exception on return from the embedded call. If the argument is `True`, then any exceptions resulting from the operation will not cause a floating point exception to be taken, but will instead update the flag register passed to `vmirtFBinopSimdRRR` using bitwise-or. Compound operations thus allow several run-time floating point primitives to be combined in a single embedded call, at the end of which the composite state can be used to determine whether an exception should be signaled or not.

The descriptor returned by this function is persistent during a simulation session. This means that it can be cached for later use if required.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.10 vmirtGetFCompareRRDesc

Prototype

```
vmiFPCompareDescCP vmirtGetFCompareRRDesc(
    vmiProcessorP processor,
    Uns8          dBits,
    vmiFType      type,
    Uns32         num,
    Bool          allowQNaN,
    vmiFPConfigCP config,
    Bool          inCompound
);
```

Description

This function returns a descriptor object that can be passed as an argument to a later call to `vmirtFCompareSimdRR`. The descriptor object describes the operation of `num` parallel floating point compare operations operating on arguments of type `type`. Argument `config` contains detailed information about the semantics of the operation itself: see the *VMI Morph Time Function Reference* for detailed description of this structure. Argument `allowQNaN` indicates whether QNaN operands should be legal in the comparison (if `True`) or cause an Illegal Operation to be signaled (if `False`).

The result of the comparison is indicated by assignment of values of the enumeration type `vmiFPRelation`, which describes 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;
```

Argument `inCompound` indicates whether the operation is part of a *compound operation*. If the argument is `False`, then any enabled exceptions resulting from evaluation of the conversion will cause the processor to take a floating point exception on return from the embedded call. If the argument is `True`, then any exceptions resulting from the operation will not cause a floating point exception to be taken, but will instead update the flag register passed to `vmirtFBinopSimdRRR` using bitwise-or. Compound operations thus allow several run-time floating point primitives to be combined in a single embedded call, at the end of which the composite state can be used to determine whether an exception should be signaled or not.

The descriptor returned by this function is persistent during a simulation session. This means that it can be cached for later use if required.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.11 vmirtGetFCompareRRCDesc

Prototype

```
vmiFPCompareDescCP vmirtGetFCompareRRCDesc(
    vmiProcessorP processor,
    Uns8          dBits,
    vmiFType      type,
    Uns32         num,
    Bool          allowQNaN,
    Uns32         valueUN,
    Uns32         valueEQ,
    Uns32         valueLT,
    Uns32         valueGT,
    vmiFPConfigCP config,
    Bool          inCompound
);
```

Description

This function returns a descriptor object that can be passed as an argument to a later call to `vmirtFCompareSimdRR`. The descriptor object describes the operation of `num` parallel floating point compare operations operating on arguments of type `type`. Argument `config` contains detailed information about the semantics of the operation itself: see the *VMI Morph Time Function Reference* for detailed description of this structure. Argument `allowQNaN` indicates whether QNaN operands should be legal in the comparison (if `True`) or cause an Illegal Operation to be signaled (if `False`).

The result of the comparison is indicated by assignment of one of the four values specified using the `valueUN`, `valueEQ`, `valueLT` and `valueGT` arguments, which are used respectively for unordered, equal, less than and greater than conditions.

Argument `inCompound` indicates whether the operation is part of a *compound operation*. If the argument is `False`, then any enabled exceptions resulting from evaluation of the conversion will cause the processor to take a floating point exception on return from the embedded call. If the argument is `True`, then any exceptions resulting from the operation will not cause a floating point exception to be taken, but will instead update the flag register passed to `vmirtFBinopSimdRRR` using bitwise-or. Compound operations thus allow several run-time floating point primitives to be combined in a single embedded call, at the end of which the composite state can be used to determine whether an exception should be signaled or not.

The descriptor returned by this function is persistent during a simulation session. This means that it can be cached for later use if required.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.12 vmirtFConvertSimdRR

Prototype

```
Bool vmirtFConvertSimdRR(
    vmiProcessorP      processor,
    vmiFPConvertDescCP desc,
    void              *fd,
    const void        *fa,
    Uns8              *flags
);
```

Description

This function implements a floating point conversion operation using the passed descriptor. The source arguments are referenced by argument `fa`, and the results assigned to argument `fd`. Any exceptions raised by the operation are assigned to argument `flags`. Any of `fd`, `fa` or `flags` may be `NULL`, in which case results are discarded and inputs treated as zero. The return value is `True` if the operation resulted in a processor exception being taken, and `False` otherwise.

Depending on the details of the descriptor, the operation of this function is exactly equivalent to the morph-time primitive functions `vmimtFConvertRR` and `vmimtFConvertSimdRR`: refer to the *VMI Morph Time Function Reference* manual for a detailed description.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.13 vmirtFUnopSimdRR

Prototype

```
Bool vmirtFUnopSimdRR(
    vmiProcessorP   processor,
    vmiFPUnopDescCP desc,
    void            *fd,
    const void      *fa,
    Uns8            *flags
);
```

Description

This function implements a floating point unary operation using the passed descriptor. The source arguments are referenced by argument `fa`, and the results assigned to argument `fd`. Any exceptions raised by the operation are assigned to argument `flags`. Any of `fd`, `fa` or `flags` may be `NULL`, in which case results are discarded and inputs treated as zero. The return value is `True` if the operation resulted in a processor exception being taken, and `False` otherwise.

Depending on the details of the descriptor, the operation of this function is exactly equivalent to the morph-time primitive functions `vmimtFUnopRR` and `vmimtFUnopSimdRR`: refer to the *VMI Morph Time Function Reference* manual for a detailed description.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.14 vmirtFBinopSimdRRR

Prototype

```
Bool vmirtFBinopSimdRRR(
    vmiProcessorP    processor,
    vmiFPBinopDescCP desc,
    void            *fd,
    const void      *fa,
    const void      *fb,
    Uns8            *flags
);
```

Description

This function implements a floating point binary operation using the passed descriptor. The source arguments are referenced by arguments `fa` and `fb`, and the results assigned to argument `fd`. Any exceptions raised by the operation are assigned to argument `flags`. Any of `fd`, `fa`, `fb` or `flags` may be `NULL`, in which case results are discarded and inputs treated as zero. The return value is `True` if the operation resulted in a processor exception being taken, and `False` otherwise.

Depending on the details of the descriptor, the operation of this function is exactly equivalent to the morph-time primitive functions `vmimtFBinopRRR` and `vmimtFBinopSimdRRR`: refer to the *VMI Morph Time Function Reference* manual for a detailed description.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.15 vmirtFTernopSimdRRRR

Prototype

```
Bool vmirtFTernopSimdRRRR(
    vmiProcessorP      processor,
    vmiFPTernopDescCP desc,
    void              *fd,
    const void        *fa,
    const void        *fb,
    const void        *fc,
    Uns8              *flags
);
```

Description

This function implements a floating point ternary operation using the passed descriptor. The source arguments are referenced by arguments `fa`, `fb` and `fc`, and the results assigned to argument `fd`. Any exceptions raised by the operation are assigned to argument `flags`. Any of `fd`, `fa`, `fb`, `fc` or `flags` may be `NULL`, in which case results are discarded and inputs treated as zero. The return value is `True` if the operation resulted in a processor exception being taken, and `False` otherwise.

Depending on the details of the descriptor, the operation of this function is exactly equivalent to the morph-time primitive functions `vmimtFTernopRRRR` and `vmimtFTernopSimdRRRR`: refer to the *VMI Morph Time Function Reference* manual for a detailed description.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

14.16 vmirtFCompareSimdRR

Prototype

```
Bool vmirtFCompareSimdRR(
    vmiProcessorP      processor,
    vmiFPCompareDescCP desc,
    void              *rd,
    const void        *fa,
    const void        *fb,
    Uns8              *flags
);
```

Description

This function implements a floating point comparison operation using the passed descriptor. The source arguments are referenced by arguments `fa` and `fb`, and the results assigned to argument `rd`. Any exceptions raised by the operation are assigned to argument `flags`. Any of `rd`, `fa`, `fb` or `flags` may be `NULL`, in which case results are discarded and inputs treated as zero. The return value is `True` if the operation resulted in a processor exception being taken, and `False` otherwise.

Depending on the details of the descriptor, the operation of this function is exactly equivalent to the morph-time primitive functions `vmimtFCompareRR`, `vmimtFCompareRRC`, `vmimtFCompareSimdRR` and `vmimtFCompareSimdRRC`: refer to the *VMI Morph Time Function Reference* manual for a detailed description.

Example

This function is not currently used by any OVP models.

QuantumLeap Semantics

Non-self-synchronizing

15 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.

This section describes routines that are used to establish connections between processors at initialization time, and to send and receive data using connection objects using run time calls.

The *VMI Morph Time Function Reference* describes functions that are used to emit code to send and receive data, which may be more convenient to use in some circumstances. Those functions are mentioned below for comparative purposes.

15.1 vmirtConnGetInput

Prototype

```
vmiConnInputP vmirtConnGetInput(
    vmiProcessorP processor,
    const char   *connPortName,
    Uns32         width
);
```

Description

Deprecated. See section 4.4 for an example of how to create and initialize connection ports.

QuantumLeap Semantics

Non-self-synchronizing

15.2 vmirtConnGetOutput

Prototype

```
vmiConnOutputP vmirtConnGetOutput(
    vmiProcessorP processor,
    const char   *connPortName,
    Uns32         width
);
```

Description

Deprecated. See section 4.4 for an example of how to create and initialize connection ports.

QuantumLeap Semantics

Non-self-synchronizing

15.3 vmirtConnGetInputInfo

Prototype

```
void vmirtConnGetInputInfo(vmiConnInputP conn, vmiConnInfoP info);
```

Description

This function returns information about the current state of the input connection container object `conn` by filling a structure of type `vmiConnInfo`. The structure is defined as follows:

```
typedef struct vmiConnInfoS {
    Uns32 words;        // maximum number of entries the connection can hold
    Uns32 numFilled;    // number of filled entries
    Uns32 numEmpty;     // number of empty entries
    Uns32 bits;         // the width of each entry in bits
} vmiConnInfo, *vmiConnInfoP;
```

Example

```
#include “vmi/vmiMessage.h”
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    vmiConnInputP inputConn;        // input connection (see section 4.4)
} cpux, *cpuxP;

// Print debug information about the processor input connection
static void vmic_InputConnDebug(cpuxP cpux) {

    // fill a local structure with connection information
    vmiConnInfo info;
    vmirtConnGetInputInfo(cpux->inputConn, &info);

    vmiPrintf(“processor %s:\n”, vmirtProcessorName((vmiProcessorP)cpux));
    vmiPrintf(“  words     = %u\n”, info.words);
    vmiPrintf(“  numFilled = %u\n”, info.numFilled);
    vmiPrintf(“  numEmpty  = %u\n”, info.numEmpty);
    vmiPrintf(“  bits      = %u\n”, info.bits);
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

15.4 vmirtConnGetOutputInfo

Prototype

```
void vmirtConnGetOutputInfo(vmiConnOutputP conn, vmiConnInfoP info);
```

Description

This function returns information about the current state of the output connection container object `conn` by filling a structure of type `vmiConnInfo`. The structure is defined as follows:

```
typedef struct vmiConnInfoS {
    Uns32 words;        // maximum number of entries the connection can hold
    Uns32 numFilled;    // number of filled entries
    Uns32 numEmpty;     // number of empty entries
    Uns32 bits;         // the width of each entry in bits
} vmiConnInfo, *vmiConnInfoP;
```

Example

```
#include “vmi/vmiMessage.h”
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

// processor structure definition
typedef struct cpuxS {
    vmiConnOutputP outputConn;      // output connection (see section 4.4)
} cpux, *cpuxP;

// Print debug information about the processor input connection
static void vmic_OutputConnDebug(cpuxP cpux) {

    // fill a local structure with connection information
    vmiConnInfo info;
    vmirtConnGetOutputInfo(cpux->outputConn, &info);

    vmiPrintf(“processor %s:\n”, vmirtProcessorName((vmiProcessorP)cpux));
    vmiPrintf(“  words     = %u\n”, info.words);
    vmiPrintf(“  numFilled = %u\n”, info.numFilled);
    vmiPrintf(“  numEmpty  = %u\n”, info.numEmpty);
    vmiPrintf(“  bits      = %u\n”, info.bits);
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

15.5 vmirtConnGet

Prototype

```
Bool vmirtConnGet(vmiConnInputP conn, void *value, Bool peek);
```

Description

This function performs a read from a connection container object specified by `conn`. The value read is assigned to the address specified by `value`; the size of `value` (in bits) must have been specified previously when calling `vmirtConnGetInput`. If `peek` is `True`, 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 function returns `False` and `value` is unaffected; otherwise, if the read was successful, the function returns `True`.

This function may be used in combination with `vmirtConnNotifyPut` and `vmirtHalt` to implement a blocking read, as shown in the example below. Blocking reads may also be implemented more directly using the morph-time function `vmimtConnGetRB`; see the *VMI Morph Time Function Reference* for more details.

Example

The template OR1K model uses this function as follows:

```
static Uns32 vmic_ConnGetOrBlock(or1kP or1k, Bool peek) {

    or1k->connValue = 0;

    // initially assume connection is not blocked
    or1k->blockState = OR1K_BS_NONE;

    if(!vmirtConnGet(or1k->inputConn, &or1k->connValue, peek)) {

        // halt this processor and re-execute this instruction on restart
        vmirtBlock((vmiProcessorP)or1k);

        // install notifier to be called on connection event
        vmirtConnNotifyPut(or1k->inputConn, 0, 0);

        // indicate input connection is blocked
        or1k->blockState = OR1K_BS_INPUT;
    }

    return or1k->connValue;
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

15.6 vmirtConnPut

Prototype

```
Bool vmirtConnPut(vmiConnOutputP conn, const void *value);
```

Description

This function performs a write to a connection container object specified by `conn`. The value to write is specified by `value`; the size of `value` (in bits) must have been specified previously when calling `vmirtConnGetOutput`.

In the case that the output connection container is full prior to the attempted write, the function returns `False` and the write is not performed; otherwise, if the write was successful, the function returns `True`.

This function may be used in combination with `vmirtConnNotifyGet` and `vmirtHalt` to implement a blocking write, as shown in the example below. Blocking writes may also be implemented more directly using the morph-time function `vmimtConnPutRB`; see the *VMI Morph Time Function Reference* for more details.

Example

The template OR1K model uses this function as follows:

```
static void vmic_ConnPutOrBlock(or1kP or1k) {

    // initially assume connection is not blocked
    or1k->blockState = OR1K_BS_NONE;

    if(!vmirtConnPut(or1k->outputConn, &or1k->connValue)) {

        // halt this processor and re-execute this instruction on restart
        vmirtBlock((vmiProcessorP)or1k);

        // install notifier to be called on connection event
        vmirtConnNotifyGet(or1k->outputConn, 0, 0);

        // indicate output connection is blocked
        or1k->blockState = OR1K_BS_OUTPUT;
    }
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

15.7 vmirtConnNotifyGet

Prototype

```
Bool vmirtConnNotifyGet(
    vmiConnOutputP  conn,
    vmiConnUpdateFn updateCB,
    void           *userData
);
```

Description

This function registers an update function, `updateCB`, which is called the next time a value is removed from the output connection container (for example, by a call to `vmirtConnGet`). The callback is one-shot: that is, it is deregistered when it is called. The value of `updateCB` may be `NULL`, in which case a default notifier is used that restarts the blocked processor immediately (equivalent to `vmirtRestartNow`).

The `vmiConnUpdateFn` function type is defined as follows in file `vmiTypes.h`. It is is passed the blocking processor and `userData` supplied as an argument to `vmirtConnNotifyGet` as arguments:

```
#define VMI_CONN_UPDATE_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    void         *userData      \
)
typedef VMI_CONN_UPDATE_FN((*vmiConnUpdateFn));
```

This function is required to support blocking writes with `vmirtConnPut`, as shown in the following example. The registered callback will typically call either `vmirtRestartNow` or `vmirtRestartNext` to restart the blocked processor, and possibly perform other actions.

Example

The template OR1K model uses this function as follows:

```
static void vmic_ConnPutOrBlock(or1kP or1k) {

    // initially assume connection is not blocked
    or1k->blockState = OR1K_BS_NONE;

    if(!vmirtConnPut(or1k->outputConn, &or1k->connValue)) {

        // halt this processor and re-execute this instruction on restart
        vmirtBlock((vmiProcessorP)or1k);

        // install notifier to be called on connection event
        vmirtConnNotifyGet(or1k->outputConn, 0, 0);

        // indicate output connection is blocked
        or1k->blockState = OR1K_BS_OUTPUT;
    }
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

15.8 vmirtConnNotifyPut

Prototype

```
Bool vmirtConnNotifyPut(
    vmiConnInputP   conn,
    vmiConnUpdateFn updateCB,
    void           *userData
);
```

Description

This function registers an update function, `updateCB`, which is called the next time a value is written to the input connection container (for example, by a call to `vmirtConnPut`). The callback is one-shot: that is, it is deregistered when it is called. The value of `updateCB` may be `NULL`, in which case a default notifier is used that restarts the blocked processor immediately (equivalent to `vmirtRestartNow`).

The `vmiConnUpdateFn` function type is defined as follows in file `vmiTypes.h`. It is is passed the blocking processor and `userData` supplied as an argument to `vmirtConnNotifyPut` as arguments:

```
#define VMI_CONN_UPDATE_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    void         *userData      \
)
typedef VMI_CONN_UPDATE_FN((*vmiConnUpdateFn));
```

This function is required to support blocking reads with `vmirtConnGet`, as shown in the following example. The registered callback will typically call either `vmirtRestartNow` or `vmirtRestartNext` to restart the blocked processor, and possibly perform other actions.

Example

The template OR1K model uses this function as follows:

```
static Uns32 vmic_ConnGetOrBlock(or1kP or1k, Bool peek) {

    or1k->connValue = 0;

    // initially assume connection is not blocked
    or1k->blockState = OR1K_BS_NONE;

    if(!vmirtConnGet(or1k->inputConn, &or1k->connValue, peek)) {

        // halt this processor and re-execute this instruction on restart
        vmirtBlock((vmiProcessorP)or1k);

        // install notifier to be called on connection event
        vmirtConnNotifyPut(or1k->inputConn, 0, 0);

        // indicate input connection is blocked
        or1k->blockState = OR1K_BS_INPUT;
    }

    return or1k->connValue;
}
```

Notes and Restrictions
See section 4.4.

QuantumLeap Semantics
Synchronizing

16 Simulation Environment Access

This section describes support functions that allow access to the simulation environment. For example, it is possible to obtain the values of command line flags, or Imperas Simulator platform attributes, or CpuManager / OVPsim user attributes.

16.1 vmirtGetCurrentProcessor

Prototype

```
vmiProcessorP vmirtGetCurrentProcessor(void);
```

Description

This routine returns the currently-executing processor. If it is called in a context where there is no currently-executing processor (for example, in a processor constructor or destructor) it returns `NULL`.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

16.2 vmirtCPUId

Prototype

```
Uns32 vmirtCPUId(vmiProcessorP processor);
```

Description

This routine returns the *CPU id* of the passed processor. The CPU id is specified using the `cpuid` parameter on a processor when it is instantiated. The CPU id is a generic simulation artifact value and not related to any model-specific hardware registers.

This function is useful when simulating multiprocessor platforms where behavior differs between processors and the processor models have no model-specific hardware registers that can be used for identification purposes (in general, this implies a proof-of-concept virtual platform using processors not originally designed for use in a multiprocessor context). For example, processors might be required to jump to different handler addresses on interrupt.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

16.3 vmirtGetProcessorForCPUId

Prototype

```
vmiProcessorP vmirtGetProcessorForCPUId(Uns32 cpuId);
```

Description

Given a *CPU id*, this function returns any processor in the platform with a matching id. If there is no matching processor, it returns `NULL`. The CPU id is specified using the `cpuid` parameter on a processor when it is instantiated. The CPU id is a generic simulation artifact value and not related to any model-specific hardware registers.

This function enables direct inter-processor introspection. An example use is a situation where registers of one processor are directly readable by another (perhaps by memory mapping). This is often the case in platforms where processors have been specifically designed to operate in groups.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.4 vmirtProcessorFlags

Prototype

```
Uns32 vmirtProcessorFlags(vmiProcessorP processor);
```

Description

Every processor in a simulation can be supplied with a model-specific set of *flags* as a 32-bit value. These flags can be freely used by the model, perhaps to enable special operating modes or debug output. The flags are specified using the `debugflags` parameter on a processor when it is instantiated.

This function allows a model to obtain its model-specific flags. It is usually used in the processor constructor.

Example

The OVP ARC model uses this function in its constructor as follows:

```
VMI_CONSTRUCTOR_FN(arcConstructor) {

    arcP            arc    = (arcP)processor;
    arcParamValuesP params = parameterValues;

    // save flags on processor structure
    arc->flags = vmirtProcessorFlags(processor);

    . . . lines omitted for clarity . . .
}
```

The value of `arc->flags` is used throughout the model to enable specific debug messages.

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

16.5 vmirtProcessorName

Prototype

```
const char *vmirtProcessorName(vmiProcessorP processor);
```

Description

This function returns the instance name of the passed processor, typically used for output message context.

Example

The OVP ARC model implements a macro-based message system:

```
#define SRCREF_FMT  "CPU '%s' 0x%08x %s: "
#define SRCREF_ARGS(_P, _PC) \
    vmirtProcessorName(_P), _PC, arcDisassemble(_P, _PC, DSA_NORMAL)
```

These macros are used to generate informative messages. For example:

```
void arcAuxBreak(arcP arc) {

    if(inKernelMode(arc) || AUX_FIELD(arc, debug, UB)) {

        // set the H bit in the status register
        arcSetHBit(arc, 1);

        // the BH bit in the Debug register is set too
        AUX_FIELD(arc, debug, BH) = 1;

    } else {

        // instruction not enabled in user mode
        // report the invalid register in verbose mode
        if(arc->verbose) {
            vmiMessage("W", CPU_PREFIX"_BNE",
                SRCREF_FMT "break instruction not enabled in %s mode",
                SRCREF_ARGS((vmiProcessorP)arc, getPC(arc)),
                arcGetModeName(arc, arc->mode)
            );
        }

        // take InstructionError exception if required
        arcTakeException0(arc, EC_IllegalInstruction);
    }
}
```

Notes and Restrictions

1. When called for a processor that is a member of an SMP cluster that has not been explicitly named using `vmirtSetProcessorName`, the returned name is a constructed string using the format:
    `<baseName>_<index1>_<index2>_..._<indexN>`
   where `<baseName>` is the given name of the SMP root processor and `<index1>`, `<index2>`, … etc are the SMP indices of each processor level down the hierarchy.
2. If the name is constructed using SMP indices, it is valid only until the next call to `vmirtSetProcessorName`.

3. If the name is constructed using SMP indices, it will change if the processor is relocated in the SMP hierarchy by a call the `vmirtSetSMPParent`.

QuantumLeap Semantics
Non-self-synchronizing

16.6 vmirtSetProcessorName

Prototype

```
void vmirtSetProcessorName(vmiProcessorP processor, const char *name);
```

Description

This function sets the name of the processor. The new name will be shown in all trace output and verbose listings, and will be returned by `vmirtProcessorName`. This function should usually be called in the processor constructor.

Example

This example is taken from the OVP MIPS32 model. `vmirtSetProcessorName` is used in this model to specify the names of all CPU, VPE and TC objects in the SMP cluster. The generated names incorporate the parent name at each level to ensure uniqueness in the platform:

```
static void setName(vmiProcessorP processor, const char *type) {

    vmiProcessorP parent = vmirtGetSMPParent(processor);

    if(parent) {

        // TCs appear to be children of CPU rather than VPE
        if(((mipsP)parent)->objectType == MOT_VPE) {
            parent = vmirtGetSMPParent(parent);
        }

        const char *baseName = vmirtProcessorName(parent);
        char        name[strlen(baseName)+10];
        char        index[10];

        sprintf(index, "%u", vmirtGetSMPIndex(processor));
        strcpy(name, baseName);
        strcat(name, "_");
        strcat(name, type);
        strcat(name, index);

        vmirtSetProcessorName(processor, name);
    }
}
```

Notes and Restrictions

1. For Imperas tool compatibility, choose simple alphanumeric processor names. Names with special characters may prevent the use of Imperas tools.

QuantumLeap Semantics

Non-self-synchronizing

16.7 vmirtProcessorVariant

Prototype

```
const char *vmirtProcessorVariant(vmiProcessorP processor);
```

Description

This function returns the variant name of the passed processor, typically used for output message context.

Example

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

Notes and Restrictions

1. When called for a processor that is a member of an SMP cluster, the variant name of the enclosing cluster is returned by default. SMP cluster member variant names are typically modified in the processor constructor using `vmirtSetProcessorVariant` if a different variant name is required.

QuantumLeap Semantics

Non-self-synchronizing

16.8 vmirtSetProcessorVariant

Prototype

```
void vmirtSetProcessorVariant(
    vmiProcessorP processor,
    const char   *variant
);
```

Description

This function sets the variant name of the processor. The new name will be shown in verbose listings, and will be returned by `vmirtProcessorVariant`. This function should usually be called in the processor constructor.

Example

This example is based on the OVP ARM model. This model supports *multicluster* variants, where a single multicluster processor with a shared GIC can instance a number of separate clusters of different types (for example, a Cortex-A53MPx4 and a Cortex-A57MPx4). `vmirtSetProcessorVariant` is used in the model constructor to set the variant of clusters and their contained processing elements (PEs) to the cluster variant, instead of the generic name *MultiCluster* which is used for the top level.

```
VMI_CONSTRUCTOR_FN(armConstructor) {

    armP arm    = (armP)processor;
    armP parent = getParent(arm);

    if(parent && !armIsCluster(parent)) {

        // copy configuration from the parent
        arm->configInfo = parent->configInfo;

    } else {

        // get default variant information (this is specific to
        // a multicluster member variant type and includes the
        // multicluster variant name)
        arm->configInfo = *getConfigVariantArg(parent, arm);

    }

    // update variant name
    vmirtSetProcessorVariant(processor, arm->configInfo.name);

    . . .
}
```

Notes and Restrictions

1. This function affects the reported variant name only.

QuantumLeap Semantics

Non-self-synchronizing

16.9 vmirtProcessorInfo

Prototype

```
vmiProcessorInfoCP vmirtProcessorInfo(vmiProcessorP processor);
```

Description

This function returns a pointer to the `vmiProcessorInfo` structure of the passed processor. See the *OVP Processor Modeling Guide* for more information about this structure.

This function is typically used in intercept libraries that need to understand details of the processor model implementation (for example, what the model VLNV path is, or what debugger is used).

Example

The following example shows how this function is used in the standard Context-Aware Trace tool intercept library to obtain a file path for the default rules file used by that tool, which requires knowledge of the VLNV path of the processor:

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    const char *definitions_file = params->definitions_file;

    . . .

    if(definitions_file) {

        // file specified using definitions_file parameter on intercept

    } else if((info=vmirtProcessorInfo(processor))) {

        // construct default definitions file name
        definitions_file = getDefinitionsFileName(
            info->vlnv.vendor,
            info->vlnv.library,
            info->vlnv.name,
            info->vlnv.version,
            "register_info.lis"
        );
    }

    . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

16.10 vmirtProcessorType

Prototype

```
const char *vmirtProcessorType(vmiProcessorP processor);
```

Description

This function returns the *type* of the passed processor. The type is specified using the `vlnv.name` field in the `vmiProcessorInfo` structure for the processor. See the *OVP Processor Modeling Guide* for more information about this structure.

Example

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

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of function `vmirtProcessorInfo`, which returns a pointer to the `vmiProcessorInfo` structure allowing all information it contains to be accessed.

QuantumLeap Semantics

Non-self-synchronizing

16.11 vmirtProcessorStringAttribute

Prototype

```
const char *vmirtProcessorStringAttribute(
    vmiProcessorP processor,
    const char   *name
);
```

Description

This routine enables any string-valued platform attribute to be accessed within an intercept library. If the named attribute exists and is string-valued, the function returns the value; otherwise, it returns `NULL`.

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to check for a user specified attribute first on the extension library, then on the processor and use a default value if neither are found:

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    // get parameters for intercept library (passed to constructor)
    paramValuesP params = parameterValues;
    const char  *regName;

    if((regName=params->resultReg)){

        // Use value specified on extension library
        strcpy(object->regName, regName);

    } else if((regName=vmirtProcessorStringAttribute(processor,"resultReg"))){

        // Use value specified on processor
        strcpy(object->regName, regName);

    } else {

        // Use default value
        strcpy(object->regName, regNameDefault);
    }
}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3
2. In VMI versions prior to 6.8.0, this function was called `vmirtPlatformStringAttribute`. This name is deprecated.

QuantumLeap Semantics

Non-self-synchronizing

16.12 vmirtProcessorBoolAttribute

Prototype

```
Bool vmirtProcessorBoolAttribute(
    vmiProcessorP processor,
    const char    *name,
    Bool          *found
);
```

Description

This routine enables any `Bool`-valued processor attribute to be accessed. If the named attribute exists and is of `Bool` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get an enable flag for the processor to which the library is attached.

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiosObjectP kobject = object;

    //
    // Read Attributes
    //
    Bool found;
    Bool val;

    //
    // Are we invoking strace ?
    //
    object->strace = False;
    val = vmirtProcessorBoolAttribute(processor, "strace", &found);
    if (found && val) {
        object->strace = True;
    }

    ...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3.

QuantumLeap Semantics

Non-self-synchronizing

16.13 vmirtProcessorUns32Attribute

Prototype

```
Uns32 vmirtProcessorUns32Attribute(
    vmiProcessorP processor,
    const char    *name,
    Bool          *found
);
```

Description

This routine enables any `Uns32`-valued processor attribute to be accessed. If the named attribute exists and is of `Uns32` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get CPU number property defined for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiosObjectP kobject = object;

    //
    // Read Attributes
    //
    Bool found;
    Uns32 val;

    object->numcpu = NUMCPU;
    val = vmirtProcessorUns32Attribute(processor, "numcpu", &found);
    if (found) {
        object->numcpu = val;
    }

    ...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3.

QuantumLeap Semantics

Non-self-synchronizing

16.14 vmirtProcessorUns64Attribute

Prototype

```
Uns64 vmirtProcessorUns64Attribute(
    vmiProcessorP processor,
    const char    *name,
    Bool          *found
);
```

Description

This routine enables any `Uns64`-valued processor attribute to be accessed. If the named attribute exists and is of `Uns64` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get a page size defined for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiosObjectP kobject = object;

    //
    // Read Attributes
    //
    Bool found;
    Uns64 val;

    //
    // Default location for user program top of stack at initialization
    //
    object->pagesize = PAGE_SIZE;
    val = vmirtProcessorUns64Attribute(processor, "pagesize", &found);
    if (found) {
        object->pagesize = val;
    }

    ...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3
2. In VMI versions prior to 6.8.0, this function was called `vmirtPlatformUns64Attribute`. This name is deprecated.

QuantumLeap Semantics
Non-self-synchronizing

16.15 vmirtProcessorFlt64Attribute

Prototype

```
Flt64 vmirtProcessorFlt64Attribute(
    vmiProcessorP processor,
    const char    *name,
    Bool          *found
);
```

Description

This routine enables any `Flt64`-valued processor attribute to be accessed. If the named attribute exists and is of `Flt64` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0.0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get the nominal processor MIPS rate defined for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

//
// Library Constructor Function
//
static VMIOS_CONSTRUCTOR_FN(libraryConstructor)) {

...
    // get processor mips rate
    Bool found;
    Flt64 mipsInMhz =
        vmirtProcessorFlt64Attribute(processor, "mips", &found);
    if (!found) {
        mipsInMhz = DEFAULT_MIPS;
    }
...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3

QuantumLeap Semantics

Non-self-synchronizing

17 SMP Processor Functions

The VMI interfaces allow *Symmetric MultiProcessing* (SMP) processors to be modeled. This section describes the concepts involved and the functions available to support SMP processors. SMP function support has revised and improved from version 2.0.20 of the API.

17.1 SMP Processor Hierarchies

SMP processor hierarchies consist of an arbitrary number of levels of *container* processor objects that contain, at the lowest level, *leaf* processor objects. Only the leaf level objects are simulated – container objects do not have a program counter and are not simulated, but may contain registers that are shared by descendant leaf objects that *are* simulated. Leaf level processors may get their parents or higher level ancestors using `vmirtGetSMPParent` to access these shared registers. Leaf level objects may be either *self-contained* or *simulation group members*.

Self-contained objects are treated by the simulator exactly as if they had been instantiated as separate processors: each has its own program counter and will run for a complete quantum when simulated. This level of abstraction is suitable for modeling processors with multiple independent cores.

Simulation group members are treated by the simulator as a time-sliced group: when simulated for a quantum, the quantum time will be divided into a number of sub-quanta, and each of the group members will be run for a sub-quantum before moving on to the next. If any group member is halted, the unused simulation time will be used instead by other running members of the group. This level of abstraction is suitable for modeling processors that use microthreading to share some common resources between several thread contexts. The container of all the simulation group members is a *simulation group* object.

Schedulable processors are either *self-contained objects* or *simulation group objects*, using the definitions above.

SMP processor hierarchies are defined using the function `smpContext` argument to the processor model constructor. Other functions are available to traverse the SMP hierarchy at run time and to dynamically modify the SMP hierarchy and simulation group schedule order.

17.2 Specifying SMP Attributes in the Processor Constructor

The processor constructor has the following prototype:

```
#define VMI_CONSTRUCTOR_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,           \
    const char    *type,                \
    Bool           simulateExceptions,  \
    vmiSMPContextP smpContext           \
)
```

The `smpContext` argument is used to construct an SMP processor hierarchy beneath the root level processor object. The type `vmiSMPContext` is defined in `vmiTypes.h` as follows:

```
typedef struct vmiSMPContextS {
    Bool  isContainer;  // whether this processor is a container
    Uns32 numChildren;  // if a container, the initial number of children
    Uns32 numSubSlices; // if a container, the number of subslices
    Uns32 index;        // index number in schedule list
} vmiSMPContext, *vmiSMPContextP;
```

Fields in this structure should be updated by the constructor to specify the details of the SMP hierarchy, as follows:

1. If this processor is to be a container object with children beneath it, set the `isContainer` field to `True`. For leaf-level objects, `isContainer` should be `False` (the default value).
2. The initial number of children of the container should be set using the `numChildren` field. It is valid for a container to have no children initially, in which case `numChildren` should be zero (the default). If this is a container object, then on completion of the constructor `numChildren` processor objects will be allocated and attached as children of this processor, with indices `0`, `1`, … `numChildren -1`. The constructor will then be called again for each child processor, enabling further levels to be added to the hierarchy if required beneath each child.
3. The `numSubSlices` field only has effect for a parent of leaf-level processors. If `numSubSlices` is non-zero, then children of this processor comprise a *simulation group*: each quantum for which the processor is simulated will be split into `numSubSlices` sub-quanta, and the children will share these sub-quanta when simulated, as described in the introduction to this section. If `numSubSlices` is zero, then children of this processor will be considered as independent cores when simulated.
4. The `index` field specifies the initial SMP index number for this processor. This index number is returned by `vmirtGetSMPIndex`, and used to construct the processor name if no name has been explicitly specified using `vmirtSetProcessorName` (see `vmirtProcessorName` for a description of the default name syntax). If the processor is a member of a simulation group, the index determines the order in which members of that group are simulated (smallest indices are simulated first). It is legal for two members of a simulation

group to have the same index; in this case, the order in which they are simulated is indeterminate. By default, index numbers increase from zero for each child of a processor.

The `vmiSMPContext` field options are summarized in the following table:

Level	isContainer	numChildren	numSubSlices	Meaning
leaf	`False`	(ignored)	(ignored)	leaf level processor in SMP hierarchy (default)
parent of container	`True`	`N`	(ignored)	container processor with `N` children which are themselves containers
parent of leaf	`True`	`N`	`0`	container processor with `N` independently-scheduled children
parent of leaf	`True`	`N`	`M {!=0)`	container processor with `N` children in a simulation group with `M` sub-quanta

Example

This example is taken from the standard OVP MIPS processor model. In general, MIPS processors can have up to four hierarchy levels:

1. The *CMP* level – typically, four processors in a cluster (for example, a 1004K);
2. The *core* level – an individual core in a 1004K, or the root level in non-CMP processors (e.g. 34K);
3. The *VPE* level – a *virtual processing element* within a processor such as the 34K;
4. The *TC* level – a *thread context* within a VPE.

The TCs within a VPE are an SMP group that share timers and other hardware resources, but have separate GPRs.

The constructor for the MIPS processor uses the `smpContext` in four ways:

1. At the CMP level, the processor is specified to be a container for *cores* in the CMP;
2. At the core level, the processor is specified to be a container for *VPEs* in the core;
3. At the VPE level, the processor is specified to be a container for *TCs* in the VPE. It is also specified that the number of sub-slices should be the number of TCs in the CPU (so that when all TCs are running on one VPE, each will execute for one sub-quantum per simulated quantum).
4. At the TC level, index numbers are adjusted so that they increase uniquely across all TCs on the CPU. For example, VPE0 may initially own TCs 0, 1 and 2, and VPE1 may initially own TCs 4 and 5.

The `mips` processor object has an `objectType` field which is used to identify the type of object for which the constructor is being called.

```
#include “vmi/vmiRt.h”

VMI_CONSTRUCTOR_FN(mipsConstructor) {

    mipsP         mips        = (mipsP)processor;
    mipsP         parent      = (mipsP)vmirtGetSMPParent(processor);
    vmiViewObjectP baseObject = vmirtGetProcessorViewObject(processor);
    mipsConfigCP  config      = allocConfig(mips, parent);
    Bool           isMT       = config->Config3.MT;
    Uns32          CPUNum     = config->GCR_CONFIG.PCORES+1;

    . . .

    // determine the object type
    if(parent) {
        mips->objectType = isMT ? parent->objectType-1 : MOT_TC;
    } else if(CPUNum>1) {
        mips->objectType = MOT_CMP;
    } else if(isMT) {
        mips->objectType = MOT_CPU;
    } else {
        mips->objectType = MOT_TC;
    }

    . . .

    switch(mips->objectType) {

        case MOT_CMP: {

            // multi-processor cluster with 'CPUNum' children

            // supply SMP configuration properties
            smpContext->isContainer = True;
            smpContext->numChildren = CPUNum;

            // do CMP level initialization
            initializeCMP(processor);
            setName(processor, "CMP");

            break;
        }

        case MOT_CPU: {

            // CPU-level object with 'VPENum' children, scheduled independently
            Uns32 VPENum = mips->config->MVPConf0.PVPE+1;

            // supply SMP configuration properties
            smpContext->isContainer = True;
            smpContext->numChildren = VPENum;

            // do CPU level initialization
            initializeCore(processor);
            setName(processor, "CPU");

            break;
        }

        case MOT_VPE: {

            // VPE-level object with 'TCNum' children, scheduled together
            Uns32 TCNum     = mips->config->MVPConf0.PTC+1;
            Uns32 VPE0TCNum = mips->config->VPE0MaxTC+1;
            Uns32 VPETCNum;

            // limit VPE0TCNum to the available number of TCs
            if(VPE0TCNum>TCNum) {
                VPE0TCNum = TCNum;
            }

            // number of children depends on this VPE index
            if(smpContext->index==0) {
                VPETCNum = VPE0TCNum;
            } else if(smpContext->index==1) {
                VPETCNum = TCNum-VPE0TCNum;
            } else {
                VPETCNum = 0;
            }
            // supply SMP configuration properties
            smpContext->isContainer  = True;
            smpContext->numChildren  = VPETCNum;
            smpContext->numSubSlices = TCNum;

            // do VPE level initialization
            initializeVPE(processor);
            setName(processor, "VPE");

            break;
        }

        default: {

            // TC-level object, scheduled as a group member
            if(isMT) {

                // get parent VPE index
                Uns32 VPE0TCNum = mips->config->VPE0MaxTC+1;
                Uns32 vpeIndex  = vmirtGetSMPIndex((vmiProcessorP)parent);

                // correct indices for TCs on VPE 1
                if(vpeIndex==1) {
                    smpContext->index += VPE0TCNum;
                }

            } else {

                // do CPU/VPE register initialization
                initializeCore(processor);
                initializeVPE(processor);
            }

            // do TC level initialization
            initializeTC(processor, smpContext->index);
            setName(processor, "TC");

            break;
        }
    }
}
```

17.3 vmirtGetSMPParent

Prototype

```
vmiProcessorP vmirtGetSMPParent(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the parent processor object in the hierarchy. If the processor is not a member of an SMP hierarchy, or is at the root level, `NULL` is returned.

Example

The OVP MIPS model uses this function in the processor constructor to determine whether to generate documentation nodes:

```
VMI_CONSTRUCTOR_FN(mipsConstructor) {

    mipsP            mips   = (mipsP)processor;
    mipsParamValuesP params = parameterValues;

    // initialize configuration
    mipsAllocModelConfig(mips, params);

    // create data structures at this level
    mipsNew(mips, smpContext, params);

    // document the model (root processor only)
    if(!vmirtGetSMPParent(processor)) {
        mipsDoc(processor, params);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

17.4 vmirtSetSMPParent

Prototype

```
void vmirtSetSMPParent(vmiProcessorP processor, vmiProcessorP parent);
```

Description

This function relocates an SMP processor within the hierarchy so that its new parent is `parent`. It may be used to relocate both leaf level and container processors.

Example

This example is taken from the OVP MIPS model.

```
#include “vmi/vmiRt.h”

void mipsWriteTCBind(mipsP cxt, mipsP tc, Uns32 newValue) {

    mipsP oldVPE   = VPE_FOR_TC(tc);
    mipsP cpu      = CPU_FOR_VPE(oldVPE);
    Uns32 oldValue = COP0_REG(tc, TCBind);
    mipsP newVPE;

    // get the old value of the CurVPE field
    Uns8 oldCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    // update the field, preserving read-only bits
    COP0_REG(tc, TCBind) = (
        (oldValue & ~C0_MASK_TCBind) |
        (newValue &  C0_MASK_TCBind)
    );

    // get the new value of the CurVPE field
    Uns8 newCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    if(newCurVPE==oldCurVPE) {

        // no action if field CurVPE is unchanged

    } else if(!COP0_FIELD(cpu, MVPControl, VPC)) {

        // TCBind/CurVPE not writable unless MVPControl/VPC is clear
        COP0_FIELD(tc, TCBind, CurVPE) = oldCurVPE;

    } else if((newVPE=findVPE(cpu, newCurVPE))) {

        // reassign to the new parent
        vmirtSetSMPParent((vmiProcessorP)tc, (vmiProcessorP)newVPE);
        tc->context = newVPE;

        . . . .
    }
}
```

Notes and Restrictions

1. The new parent and the original parent must lie beneath the same root level container processor.
2. The new parent and the original parent must lie at the same depth in the tree beneath the root level container (but they do not have to be siblings).

QuantumLeap Semantics
Synchronizing

17.5 vmirtGetSMPChild

Prototype

```
vmiProcessorP vmirtGetSMPChild(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the first child processor object in the hierarchy. If the processor is not a member of an SMP hierarchy, or has no children, `NULL` is returned.

Example

The OVP ARM model uses this function to deliver event notifications to all processors in a cluster:

```
inline static armP getChild(armP arm) {
    return (armP)vmirtGetSMPChild((vmiProcessorP)arm);
}

inline static Bool isLeaf(armP arm) {
    return !getChild(arm);
}

static VMI_SMP_ITER_FN(doSEVCB) {
    armP arm = (armP)processor;
    if(isLeaf(arm)) {
        armDoEvent(arm);
    }
}

static void doSEVHier(armP root) {
    vmirtIterAllProcessors((vmiProcessorP)root, doSEVCB, 0);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

17.6 vmirtGetSMPPrevSibling

Prototype

```
vmiProcessorP vmirtGetSMPPrevSibling(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the previous sibling. If the processor is not a member of an SMP hierarchy, or has no previous sibling, `NULL` is returned.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

17.7 *vmirtGetSMPNextSibling*

Prototype

```
vmiProcessorP vmirtGetSMPNextSibling(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the previous sibling. If the processor is not a member of an SMP hierarchy, or has no previous sibling, `NULL` is returned.

Example

The OVP MIPS model uses this function to search for a VPE with a given index:

```
#define getChild(_P)         ((mipsP)vmirtGetSMPChild((vmiProcessorP)_P))
#define getNextSibling(_P)   ((mipsP)vmirtGetSMPNextSibling((vmiProcessorP)_P))

static mipsP findVPE(mipsP cpu, Uns32 vpeIndex) {

    mipsP vpe;

    for(vpe=getChild(cpu); vpe; vpe=getNextSibling(vpe)) {
        if(getSMPIndex(vpe)==vpeIndex) {
            return vpe;
        }
    }

    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

17.8 vmirtGetSMPActiveSibling

Prototype

```
vmiProcessorP vmirtGetSMPActiveSibling(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the *active* sibling. If the processor is not a member of an SMP hierarchy, the processor itself is returned

When QuantumLeap is disabled, there is only one leaf level processor in an SMP group that is active at any time (the active processor is the one currently being simulated). When the simulator switches from simulating one member of the SMP group to the next, the previously-running member becomes inactive and the new member becomes active.

When QuantumLeap is enabled, this function will return one member of the set of running processors.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

17.9 vmirtGetSMPIndex

Prototype

```
Uns32 vmirtGetSMPIndex(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the SMP index number associated with that object. If the processor is not a member of an SMP hierarchy, zero is returned.

By default, siblings are in numeric index order, with the first having an index of zero and the last an index of `coreNum-1`. This default behavior can be modified in the processor constructor or by `vmirtSetSMPIndex`.

Example

The OVP MIPS model uses this function to search for a VPE with a given index:

```
inline static Uns32 getSMPIndex(mipsP this) {
    return vmirtGetSMPIndex((vmiProcessorP)this);
}

static mipsP findVPE(mipsP cpu, Uns32 vpeIndex) {

    mipsP vpe;

    for(vpe=getChild(cpu); vpe; vpe=getNextSibling(vpe)) {
        if(getSMPIndex(vpe)==vpeIndex) {
            return vpe;
        }
    }

    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

17.10 vmirtSetSMPIndex

Prototype

```
void vmirtSetSMPIndex(vmiProcessorP processor, Uns32 index);
```

Description

Given a processor object at any level in an SMP hierarchy, this sets the SMP index number associated with that object. The index is initialized using the `index` field of the `vmiSMPContext` structure when the processor is created: see the detailed description at the start of this chapter for more information.

If processors are members of an SMP group, they are simulated in increasing index order in each time slice. This function can therefore be used to affect the scheduling order of processors in an SMP group.

Example

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

Notes and Restrictions

1. If SMP members are not explicitly named by `vmirtSetProcessorName`, the SMP index is used to construct the processor name returned by `vmirtProcesorName`, and used in trace output and reporting. Setting the index will have the effect of renaming the processor in this case.

QuantumLeap Semantics

Non-self-synchronizing

17.11 vmirtGetSMPCpuType

Prototype

```
vmiSMPCpuType vmirtGetSMPCpuType(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the `vmiSMPCpuType` enumeration member describing that object.

The `vmiSMPCpuType` enumeration is defined as:

```
typedef enum vmiSMPCpuTypeE {
    SMP_TYPE_CONTAINER = 0x0,   // Container Processor
    SMP_TYPE_LEAF      = 0x1,   // Leaf processor
    SMP_TYPE_GROUP     = 0x2,   // Part of an SMP simulation group

    // simple leaf processor (not a member of an SMP simulation group)
    SMP_SIMPLE_LEAF = SMP_TYPE_LEAF,
    // leaf processor that is a member of an SMP simulation group
    SMP_GROUP_LEAF  = SMP_TYPE_LEAF      | SMP_TYPE_GROUP,
    // simple container processor
    SMP_CONTAINER   = SMP_TYPE_CONTAINER,
    // container processor that contains a group of SMP_GROUP_LEAF processors
    SMP_GROUP       = SMP_TYPE_CONTAINER | SMP_TYPE_GROUP,
} vmiSMPCpuType;
```

This function may be used to determine if a processor is a leaf or container, and if it is a member of a simulation group.

Additional macros are defined in `vmiTypes.h` to help check specific characteristics:

```
// Is a processor of vmiSMPCpuType _T a leaf processor
#define SMP_IS_LEAF(_T) ((_T) & SMP_TYPE_LEAF)

// Is a processor of vmiSMPCpuType _T an SMP group processor
#define SMP_IS_GROUP(_T) ((_T) & SMP_TYPE_GROUP)
```

Example

The following example will report the type for each processor in an SMP hierarchy:

```
static vmiProcessorP getRoot(vmiProcessorP processor) {

    vmiProcessorP nextParent;
    while ((nextParent=vmirtGetSMPParent(processor)) != NULL) {
        processor = nextParent;
    }

    return processor;
}

static VMI_SMP_ITER_FN(reportType) {

    vmiSMPCpuType type = vmirtGetSMPCpuType(processor);
    vmiPrintf("SMP Type: \tProcessor %16s: type: %s%s\n",
            vmirtProcessorName(processor),
            SMP_IS_LEAF(type)  ? "Leaf " : "",
            SMP_IS_GROUP(type) ? "Group" : "");
}
```

```
reportSMPTypes(vmiProcessorP processor) {

    vmiProcessorP root = getRoot(processor);

    vmiPrintf("SMP Type: SMP Types for %s (root of %s):\n",
              vmirtProcessorName(root), vmirtProcessorName(processor));

    reportType(root, NULL);

    vmirtIterAllDescendants(root, reportType, NULL);
}
```

Notes and Restrictions

1. If the processor is not a member of an SMP hierarchy then the type `SMP_SIMPLE_LEAF` is returned.

QuantumLeap Semantics

Non-self-synchronizing

17.12 vmirtIterAllChildren

Prototype

```
void vmirtIterAllChildren(
    vmiProcessorP processor,
    vmiSMPIterFn  iterCB,
    void         *userData
);
```

Description

Given a processor object at any level in an SMP hierarchy, this iterates over the *children* of the processor, calling the passed `iterCB` for each one. The `iterCB` is passed the child processor pointer and a `userData` pointer. If the processor is not an SMP parent, the function has no effect.

The iterator callback should be defined using the `VMI_SMP_ITER_FN` macro.

Example

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

Notes and Restrictions

1. See also functions `vmirtIterAllDescendants` and `vmirtIterAllProcessors`.

QuantumLeap Semantics

Non-self-synchronizing

17.13 vmirtIterAllDescendants

Prototype

```
void vmirtIterAllDescendants(
    vmiProcessorP processor,
    vmiSMPIterFn  iterCB,
    void         *userData
);
```

Description

Given a processor object at any level in an SMP hierarchy, this iterates over all descendants of the processor, calling the passed `iterCB` for each one. The `iterCB` is passed the descendant processor pointer and a `userData` pointer. If the processor is not an SMP parent, the function has no effect.

The iterator callback should be defined using the `VMI_SMP_ITER_FN` macro.

Example

This example shows how this function is used in the standard OVP MIPS model destructor:

```
static VMI_SMP_ITER_FN(destructorCB) {

    mipsP mips = (mipsP)processor;

    // free UDI data structures
    mipsFreeUDI(mips);

    // free cache structures
    if(mips->cacheInfo) {
        mipsCacheFree(mips);
    }

    // free TLB structures
    if(mips->tlb) {
        mipsFreeTLB(mips);
    }

    // free debugger interface structures
    if(mips->regInfo) {
        mipsFreeRegInfo(mips);
    }

    // free shadow registers
    mipsFreeShadowRegisters(mips);

    // free single-step timer
    if(mips->sstTimer) {
        vmirtDeleteModelTimer(mips->sstTimer);
    }

    // free CMP structures
    mipsCMPFree(mips);

    // free trace state structures
    mipsFreeTraceState(mips);

    // free DV state structures
    mipsFreeDVState(mips);
    // free CDMM structures
    mipsFreeCDMM(mips);

    // free MPU structures
    mipsFreeMPU(mips);

    // free SPRAM structures
    mipsFreeSPRAM(mips);

    // free ITC structures
    mipsFreeITC(mips);
}

void mipsFree(mipsP mips) {

    vmiProcessorP processor = (vmiProcessorP)mips;

    // apply destructor to root processor
    destructorCB(processor, 0);

    // apply destructor to all descendants
    vmirtIterAllDescendants(processor, destructorCB, 0);

    // free configuration
    mipsFreeModelConfig(mips);

    // free net port specifications
    mipsFreeNetPorts(mips);

    // free bus port specifications
    mipsFreeBusPorts(mips);

    // free parameter definitions
    mipsFreeParameters(mips);
}
```

Notes and Restrictions

1. See also functions `vmirtIterAllChildren` and `vmirtIterAllProcessors`.

QuantumLeap Semantics

Non-self-synchronizing

17.14 vmirtIterAllProcessors

Prototype

```
void vmirtIterAllProcessors(
    vmiProcessorP processor,
    vmiSMPIterFn  iterCB,
    void         *userData
);
```

Description

Given a processor object at any level in an SMP hierarchy, this calls the passed `iterCB` for that processor and then iterates over all descendants of the processor, calling the passed `iterCB` for each one. The `iterCB` is passed each pointer and a `userData` pointer. If the processor is not an SMP parent, `iterCB` is called for the given processor only..

The iterator callback should be defined using the `VMI_SMP_ITER_FN` macro.

Example

The OVP ARM model uses this function to dispatch events to all leaf processors in a cluster:

```
inline static armP getChild(armP arm) {
    return (armP)vmirtGetSMPChild((vmiProcessorP)arm);
}

inline static Bool isLeaf(armP arm) {
    return !getChild(arm);
}

static VMI_SMP_ITER_FN(doSEVCB) {
    armP arm = (armP)processor;
    if(isLeaf(arm)) {
        armDoEvent(arm);
    }
}

static void doSEVHier(armP root) {
    vmirtIterAllProcessors((vmiProcessorP)root, doSEVCB, 0);
}
```

Notes and Restrictions

1. See also functions `vmirtIterAllChildren` and `vmirtIterAllDescendants`.

QuantumLeap Semantics

Non-self-synchronizing

18 Communication Between Objects

This section describes functions that enable communication between distinct objects in a simulation. The interface is typically used by *intercept objects*.

Intercept objects get all their services from the simulator. Calling functions in another intercept object, although possible, is undesirable because it introduces a requirement to load one object before the other and prevents the objects from being used independently; they would be equally useful if linked into one object. The *shared data* interface allows objects to share data and to call functions through a registry which maps symbolic names (*keys*) to shared entries.

Keys are C strings; they can contain any valid string.
Keys can either have *global* or *processor* scope. They persist until `opSessionTerminate` or the legacy `icmTerminate` function is called, or the simulator exits.

Shared data objects have *version control*, to ensure that the communicating intercept objects are using the same interpretation of any shared data. The version is a constant string which should be compiled into the two (or more) communicating intercept objects. It is the responsibility of the programmer to change the version if the protocol changes so that the programs are no longer compatible. The simulator will terminate if two intercept objects use the same key with incompatible versions.

18.1 vmirtFindAddSharedData

Prototype

```
vmiSharedDataHandleP vmirtFindAddSharedData(
     const char *version,
     const char *key,
     void       *value
);
```

Description

This function finds a shared entry by its key and returns its handle or, if the entry does not exist, creates it. This function can be used by multiple objects, regardless of the order in which they are constructed. The key has *global* scope.

The value field in the entry can be given an initial value, but it should be noted that another user of this entry could also supply an initial value, in which case the last call will win. A more useful method of sharing data is to use `vmirtSetSharedDataValue` and `vmirtGetSharedDataValue` when communication has been established.

This function is typically used in an object's constructor, but can be used at any time during a simulation run.

Example

```
// Define this in a common header file, together with any shared data types
#define API_VERSION "1.0.0"
#define FEATURE_KEY "myApplicationKey"

// Use like this in an intercept library
VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiSharedDataHandleP handle = vmirtFindAddSharedData(
        API_VERSION, FEATURE_KEY, 0
    );

    . . . use vmirtSetSharedDataValue/vmirtGetSharedDataValue etc here
}
```

Notes and Restrictions

1. See also function `vmirtFindAddProcessorSharedData` which allows creation of shared data with processor scope.

QuantumLeap Semantics

Synchronizing

18.2 vmirtFindAddProcessorSharedData

Prototype

```
vmiSharedDataHandleP vmirtFindAddProcessorSharedData(
     vmiProcessorP processor,
     const char    *version,
     const char    *key,
     void          *value
);
```

Description

This function finds a shared entry by its key and returns its handle or, if the entry does not exist, creates it. This function can be used by multiple objects, regardless of the order in which they are constructed. The key has *processor* scope: only keys defined with the same scope match.

The value field in the entry can be given an initial value, but it should be noted that another user of this entry could also supply an initial value, in which case the last call will win. A more useful method of sharing data is to use `vmirtSetSharedDataValue` and `vmirtGetSharedDataValue` when communication has been established.

This function is typically used in an object’s constructor, but can be used at any time during a simulation run.

Example

```
// Define this in a common header file, together with any shared data types
#define API_VERSION "1.0.0"
#define FEATURE_KEY "myApplicationKey"

// Use like this in an intercept library
VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiSharedDataHandleP handle = vmirtFindAddProcessorSharedData(
        processor, API_VERSION, FEATURE_KEY, 0
    );

    . . . use vmirtSetSharedDataValue/vmirtGetSharedDataValue etc here
}
```

Notes and Restrictions

1. See also function `vmirtFindAddSharedData` which allows creation of shared data with global scope.

QuantumLeap Semantics

Synchronizing

18.3 vmirtFindSharedData

Prototype

```
vmiSharedDataHandleP vmirtFindSharedData(
     const char *version,
     const char *key
);
```

Description

This function finds a shared entry by its key, and returns its handle. If no entry with that name exists, `NULL` is returned. The key has *global* scope.

Example

```
// Define this in a common header file, together with any shared data types
#define API_VERSION "1.0.0"
#define FEATURE_KEY "myApplicationKey"

// Use like this in an intercept library
VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiSharedDataHandleP handle = vmirtFindSharedData(API_VERSION, FEATURE_KEY);

    . . . use vmirtSetSharedDataValue/vmirtGetSharedDataValue etc here
}
```

Notes and Restrictions

1. See also function `vmirtFindProcessorSharedData` which allows access to shared data with processor scope.

QuantumLeap Semantics

Synchronizing

18.4 vmirtFindProcessorSharedData

Prototype

```
vmiSharedDataHandleP vmirtFindProcessorSharedData(
     vmiProcessorP processor,
     const char   *version,
     const char   *key
);
```

Description

This function finds a shared entry by its key, and returns its handle. If no entry with that name exists, `NULL` is returned. The key has *processor* scope: only keys defined with the same scope match.

Example

```
// Define this in a common header file, together with any shared data types
#define API_VERSION "1.0.0"
#define FEATURE_KEY "myApplicationKey"

// Use like this in an intercept library
VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiSharedDataHandleP handle = vmirtFindProcessorSharedData(
        processor, API_VERSION, FEATURE_KEY
    );

    . . . use vmirtSetSharedDataValue/vmirtGetSharedDataValue etc here
}
```

Notes and Restrictions

1. See also function `vmirtFindSharedData` which allows access to shared data with global scope.

QuantumLeap Semantics

Synchronizing

18.5 vmirtGetSharedDataValue

Prototype

```
 void *vmirtGetSharedDataValue(vmiSharedDataHandleP handle);
```

Description

This function returns the client-specific `data` from a shared entry.

Example

```
vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");

if(handle) {
    myType p = vmirtGetSharedDataValue(handle);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

18.6 vmirtSetSharedDataValue

Prototype

```
void vmirtSetSharedDataValue(vmiSharedDataHandleP handle, void *value);
```

Description

This function sets the client-specific `data` in a shared entry.

Example

```
vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");

if(handle) {
    myType p = ....
    vmirtSetSharedDataValue(handle, p);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

18.7 vmirtRemoveSharedData

Prototype

```
void vmirtRemoveSharedData(vmiSharedDataHandleP handle);
```

Description

This function removes a shared entry and all its listeners.

Example

```
#include “vmi/vmiRt.h”
...
    vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
...
    vmirtRemoveSharedData(handle);
...
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

18.8 vmirtWriteListeners

Prototype

```
Int32 vmirtWriteListeners(
    vmiSharedDataHandleP handle,
    Int32                 id,
    void                 *userData
);
```

Description

This function writes to all listeners registered to this entry (the listeners will usually be in another intercept object). The integer return status is set to zero then passed by address to each listener (which might modify it) before being returned by this function.

Example

```
#include “vmi/vmiRt.h”
#include “myShared.h"

VMIOS_CONSTRUCTOR(constructor) {
  ...
  vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0",  "myApplicationKey" );
  ...
}

...
  myStruct str = { ..... } ;
  // later on, during simulation
  int r = vmirtWriteListeners(handle, MY_CODE, &str);
...
```

Notes and Restrictions

1. Many listeners can be installed on one handle, allowing multiple objects to be notified by one or more shared objects.
2. It is advisable to share the definitions of `MY_CODE` and `myStruct` with other objects that communicate with this one.
3. Multiple objects can write to one shared data entry.

QuantumLeap Semantics

Synchronizing

18.9 vmirtRegisterListener

Prototype

```
void vmirtRegisterListener(
    vmiSharedDataHandleP      handle,
    vmirtSharedDataListenerFn listenerCB,
    void                     *object
);
```

Description

This function registers a *listener function* to be called when `vmirtWriteListeners` is called (by another object). The caller uses the `object` argument to specify a generic object that is passed when the listener is activated by a call to `vmirtWriteListeners`. When used in an intercept library context, this will typically be the `vmiosObjectP` of the current intercept object.

Type `vmirtSharedDataListenerFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_SHARED_DATA_LISTENER_FN(_NAME) void _NAME( \
    void  *userObject,  \
    Int32 *ret,         \
    Int32  id,          \
    void  *userData     \
)
typedef VMI_SHARED_DATA_LISTENER_FN((*vmirtSharedDataListenerFn));
```

The arguments are as follows:

1. **`userObject`**: the pointer passed as the `object` argument to `vmirtRegisterListener`.
2. **`ret`**: a pointer to the return status. This integer is set to zero by the simulator. Its address is passed to each listener in turn (who might choose to update it).
3. **`id`**: an integer passed to `vmirtWriteListeners`, typically used to specify a required service or action.
4. **`userData`**: the pointer passed as the `userData` argument to `vmirtWriteListeners`.

Function `vmirtRegisterListener` can be called multiple times for the same shared data object, usually in the context of different intercept libraries. When the listeners are activated by `vmirtWriteListeners`, each registered listener will be notified in turn.

Example

```
#include “vmi/vmiRt.h”
#include “myShared.h"

VMI_SHARED_DATA_LISTENER(myListener) {
    switch (id) {
        case MY_CODE: {
             myStruct *strP = userData;
             strP->member = ....
        }
    }

    // change the return status.
    (*ret)++;
}

VMIOS_CONSTRUCTOR(constructor) {
    vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
    vmirtRegisterListener(handle, myListener, object);
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

18.10 vmirtUnregisterListener

Prototype

```
Bool vmirtUnregisterListener(
    vmiSharedDataHandleP      handle,
    vmirtSharedDataListenerFn listenerCB,
    void                     *object
);
```

Description

This function removes a listener function with matching callback and object from a shared data handle. The function return value indicates whether a matching listener was found and removed.

Example

```
#include “vmi/vmiRt.h”
...
VMIOS_DESTRUCTOR(destructor) {
    vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
    vmirtUnregisterListener(handle, myListener, object);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

19 Address Range Hash Utilities

Modeling TLBs and caches often requires the ability to model structures with address ranges in an efficient manner. For example, a TLB may contain many entries mapping pages of different sizes, and an efficient model requires the ability to map from a virtual address to the corresponding TLB entry very quickly.

To facilitate implementation of models containing TLBs and caches, the VMI Run Time Function API implements a *range hash* object, documented in this section.

19.1 vmirtNewRangeTable

Prototype

```
void vmirtNewRangeTable(vmiRangeTablePP table);
```

Description

Create a new *range table* object, used to hold range hash entries.

Example

The OVP RISC-V model uses this function when modeling the TLB:

```
typedef struct riscvTLBS {
    vmiRangeTableP lut;     // range LUT entry (for fast lookup by address)
    tlbEntryP      free;    // list of free TLB entries available for reuse
} riscvTLB;

static riscvTLBP newTLB(riscvP riscv) {

    riscvTLBP tlb = STYPE_CALLOC(riscvTLB);

    // allocate range table for fast TLB entry search
    vmirtNewRangeTable(&tlb->lut);

    return tlb;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

19.2 vmirtFreeRangeTable

Prototype

```
void vmirtFreeRangeTable(vmiRangeTablePP table);
```

Description

Free a previously-allocated *range table* object and all its entries.

Example

The OVP RISC-V model uses this function when modeling the TLB:

```
typedef struct riscvTLBS {
    vmiRangeTableP lut;     // range LUT entry (for fast lookup by address)
    tlbEntryP      free;    // list of free TLB entries available for reuse
} riscvTLB;

static void freeTLB(riscvP riscv, riscvTLBP tlb) {

    if(tlb) {

        tlbEntryP entry;

        // delete all entries in the TLB (puts them in the free list)
        invalidateTLBEntriesRange(riscv, tlb, 0, RISCV_MAX_ADDR, MM_ANY, 0);

        // release entries in the free list
        while((entry=tlb->free)) {
            tlb->free = entry->nextFree;
            STYPE_FREE(entry);
        }

        // free the range table
        vmirtFreeRangeTable(&tlb->lut);

        // free the TLB structure
        STYPE_FREE(tlb);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

19.3 vmirtInsertRangeEntry

Prototype

```
vmiRangeEntryP vmirtInsertRangeEntry(
    vmiRangeTablePP table,
    Addr            low,
    Addr            high,
    Uns64           userData
);
```

Description

Create and return a new range table entry spanning the range `low:high` in the passed range table. The `userData` argument can hold any application-specific data required for the entry: for example, it might typically be a physical address, or a structure containing access permissions.

In VMI versions prior to 6.2.0, ranges were not permitted to overlap. From VMI version 6.2.0, range tables may hold multiple overlapping ranges without restriction.

Example

The OVP RISC-V model uses this function when modeling the TLB:

```
typedef struct riscvTLBS {
    vmiRangeTableP lut;     // range LUT entry (for fast lookup by address)
    tlbEntryP      free;    // list of free TLB entries available for reuse
} riscvTLB;

typedef struct tlbEntryS {

    Uns64 lowVA;        // entry low virtual address
    Uns64 highVA;       // entry high virtual address

    . . . fields omitted . . .

    // range LUT entry (for fast lookup by address)
    union {
        struct tlbEntryS *nextFree; // when in free list
        vmiRangeEntryP    lutEntry; // equivalent range entry when mapped
        Uns64             _size;    // for 32/64-bit host compatibility
    };
}

tlbEntry;inline static void insertTLBEntry(riscvTLBP tlb, tlbEntryP entry) {
    entry->lutEntry = vmirtInsertRangeEntry(
        &tlb->lut, entry->lowVA, entry->highVA, (UnsPS)entry
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

19.4 vmirtRemoveRangeEntry

Prototype

```
void vmirtRemoveRangeEntry(vmiRangeTablePP table, vmiRangeEntryP entry);
```

Description

Remove the passed entry for the range table and delete it.

Example

The OVP RISC-V model uses this function when modeling the TLB:

```
typedef struct riscvTLBS {
    vmiRangeTableP lut;      // range LUT entry (for fast lookup by address)
    tlbEntryP      free;     // list of free TLB entries available for reuse
} riscvTLB;

typedef struct tlbEntryS {

    . . . fields omitted . . .

    // range LUT entry (for fast lookup by address)
    union {
        struct tlbEntryS *nextFree; // when in free list
        vmiRangeEntryP    lutEntry; // equivalent range entry when mapped
        Uns64             _size;    // for 32/64-bit host compatibility
    };
}

static void deleteTLBEntry(riscvP riscv, riscvTLBP tlb, tlbEntryP entry) {

    // remove entry mappings if required
    if(entry->isMapped) {
        unmapTLBEntry(riscv, entry);
    }

    // emit debug if required
    reportDeleteTLBEntry(riscv, entry);

    // remove the TLB entry from the range LUT
    vmirtRemoveRangeEntry(&tlb->lut, entry->lutEntry);
    entry->lutEntry = 0;

    // add the TLB entry to the free list
    entry->nextFree = tlb->free;
    tlb->free       = entry;
}
```

Notes and Restrictions

1. The range entry must be present in the range table prior to removal.
2. The range entry must not be referenced once it has been removed and deleted.

QuantumLeap Semantics

Synchronizing

19.5 vmirtGetFirstRangeEntry

Prototype

```
vmiRangeEntryP vmirtGetFirstRangeEntry(
    vmiRangeTablePP table,
    Addr            low,
    Addr            high
);
```

Description

Find and return the *first* range entry in the range table that intersects the passed range `low:high`. The range entry may straddle either or both the lower or upper range bounds, or lie completely within the bounds. This function initiates a delete-safe iterator; use `vmirtGetNextRangeEntry` to find subsequent elements.

Example

The OVP RISC-V model uses this function when modeling the TLB:

```
typedef struct riscvTLBS {
    vmiRangeTableP lut;     // range LUT entry (for fast lookup by address)
    tlbEntryP      free;    // list of free TLB entries available for reuse
} riscvTLB;

static tlbEntryP firstTLBEntryRange(
    riscvP    riscv,
    riscvTLBP tlb,
    Uns64     lowVA,
    Uns64     highVA
) {
    vmiRangeEntryP lutEntry = vmirtGetFirstRangeEntry(&tlb->lut, lowVA, highVA);

    return getTLBEntryForRange(riscv, tlb, lowVA, highVA, lutEntry);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

19.6 vmirtGetNextRangeEntry

Prototype

```
vmiRangeEntryP vmirtGetNextRangeEntry(
    vmiRangeTablePP table,
    Addr            low,
    Addr            high
);
```

Description

Find and return the *next* range entry in the range table that intersects the passed range `low:high`, once the first entry has been found using `vmirtGetFirstRangeEntry`. The range entry may straddle either or both the lower or upper range bounds, or lie completely within the bounds. This iterator function is *delete-safe*; in other words, it may be used to iterate over entries in a table deleting them.

This function is available from VMI version 6.2.0.

Example

The OVP RISC-V model uses this function when modeling the TLB:

```
typedef struct riscvTLBS {
    vmiRangeTableP lut;     // range LUT entry (for fast lookup by address)
    tlbEntryP      free;    // list of free TLB entries available for reuse
} riscvTLB;

static tlbEntryP nextTLBEntryRange(
    riscvP    riscv,
    riscvTLBP tlb,
    Uns64     lowVA,
    Uns64     highVA
) {
    vmiRangeEntryP lutEntry = vmirtGetNextRangeEntry(&tlb->lut, lowVA, highVA);

    return getTLBEntryForRange(riscv, tlb, lowVA, highVA, lutEntry);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

19.7 vmirtGetRangeEntryLow

Prototype

```
Addr vmirtGetRangeEntryLow(vmiRangeEntryP entry);
```

Description

Return the low address of the passed range entry.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

19.8 vmirtGetRangeEntryHigh

Prototype

```
Addr vmirtGetRangeEntryLow(vmiRangeEntryP entry);
```

Description

Return the high address of the passed range entry.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

19.9 vmirtGetRangeEntryUserData

Prototype

```
Uns64 vmirtGetRangeEntryUserData(vmiRangeEntryP entry);
```

Description

Return any `userData` associated with the range entry.

Example

The OVP RISC-V model uses this function when modeling the TLB. The `userData` is used to hold a pointer to the model TLB entry structure:

```
static tlbEntryP getTLBEntryForRange(
    riscvP         riscv,
    riscvTLBP      tlb,
    Uns64          lowVA,
    Uns64          highVA,
    vmiRangeEntryP lutEntry
) {
    while(lutEntry) {

        union {Uns64 u64; tlbEntryP entry;} u = {
            vmirtGetRangeEntryUserData(lutEntry)
        };

        if(!u.entry->artifact) {
            return u.entry;
        }

        deleteTLBEntry(riscv, tlb, u.entry);

        lutEntry = vmirtGetNextRangeEntry(&tlb->lut, lowVA, highVA);
    }

    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

19.10 vmirtSetRangeEntryUserData

Prototype

```
void vmirtSetRangeEntryUserData(vmiRangeEntryP entry, Uns64 userData);
```

Description

Update any `userData` associated with the range entry.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

20 Save/Restore Support Functions

Processor models written using the VMI API can optionally include save/restore functionality. If this is included, then the harness layer can save processor state (using functions such as `opProcessorStateSaveFile` or the legacy `icmProcessorSaveStateFile`) and later, or in a different simulation run, restore the processor model (using functions such as `opProcessorStateRestoreFile` or the legacy `icmProcessorRestoreStateFile`). See the *Imperas Processor Modeling Guide* for a detailed description of save/restore.

This section describes support functions that are used when implementing processor state save/restore.

20.1 vmirtSave

Prototype

```
vmiSaveRestoreStatus vmirtSave(
    vmiSaveContextP cxt,
    const char     *name,
    const void     *value,
    Uns32           bytes
);
```

Description

This function saves an arbitrary `value` to the active save context `cxt`. The value is of size `bytes`. The `name` parameter associates a name with the saved value (checked for consistency when the value is later restored by `vmirtRestore`).

This function is most often used via macros `VMIRT_SAVE_REG` and `VMIRT_SAVE_FIELD`:

```
#define VMIRT_SAVE_REG(_CXT, _NAME, _REG) \
    vmirtSave(_CXT, _NAME, _REG, sizeof(*(_REG)))

#define VMIRT_SAVE_FIELD(_CXT, _PROCESSOR, _FIELD) \
    VMIRT_SAVE_REG(_CXT, #_FIELD, &(_PROCESSOR)->_FIELD)
```

Example

This example is from the template OR1K processor model to save fields in the processor structure not covered by the debug register interface:

```
VMI_SAVE_STATE_FN(or1kSaveStateCB) {

    or1kP or1k = (or1kP)processor;

    switch(phase) {

        case SRT_BEGIN:
            // start of save/restore process
            or1k->inSaveRestore = True;
            break;

        case SRT_BEGIN_CORE:
            // start of individual core
            break;

        case SRT_END_CORE:
            // end of individual core: save fields not covered by debug register
            // interface
            VMIRT_SAVE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_SAVE_FIELD(cxt, or1k, timerRunning);
            VMIRT_SAVE_FIELD(cxt, or1k, resetInput);
            VMIRT_SAVE_FIELD(cxt, or1k, reset);
            break;

        case SRT_END:
            // end of save/restore process
            or1k->inSaveRestore = False;
            break;

        default:
            // not reached
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }
}
```

Notes and Restrictions

None.

20.2 vmirtRestore

Prototype

```
vmiSaveRestoreStatus vmirtRestore(
    vmiRestoreContextP cxt,
    const char        *name,
    void              *value,
    Uns32              bytes
);
```

Description

This function restores an arbitrary `value` from the active restore context `cxt`. The value is of size `bytes`. The `name` must match the name given previously by `vmirtSave`).

This function is most often used via macros `VMIRT_RESTORE_REG` and `VMIRT_RESTORE_FIELD`:

```
#define VMIRT_RESTORE_REG(_CXT, _NAME, _REG) \
    vmirtRestore(_CXT, _NAME, _REG, sizeof(*(_REG)))

#define VMIRT_RESTORE_FIELD(_CXT, _PROCESSOR, _FIELD) \
    VMIRT_RESTORE_REG(_CXT, #_FIELD, &(_PROCESSOR)->_FIELD)
```

Example

This example is from the template OR1K processor model to restore fields in the processor structure not covered by the debug register interface:

```
VMI_RESTORE_STATE_FN(or1kRestoreStateCB) {

    or1kP or1k = (or1kP)processor;

    switch(phase) {

        case SRT_BEGIN:
            // start of save/restore process
            or1k->inSaveRestore = True;
            break;

        case SRT_BEGIN_CORE:
            // start of individual core
            break;

        case SRT_END_CORE:
            // end of individual core: save fields not covered by debug register
            // interface
            VMIRT_RESTORE_FIELD(cxt, or1k, TTCRSetCount);
            VMIRT_RESTORE_FIELD(cxt, or1k, timerRunning);
            VMIRT_RESTORE_FIELD(cxt, or1k, resetInput);
            VMIRT_RESTORE_FIELD(cxt, or1k, reset);
            // take any pending interrupt before the next instruction
            or1kInterruptNext(or1k);
            break;

        case SRT_END:
            // end of save/restore process
            or1k->inSaveRestore = False;
            break;

        default:
            // not reached
            VMI_ABORT("unimplemented case"); // LCOV_EXCL_LINE
            break;
    }
}
```

Notes and Restrictions

None.

20.3 vmirtSaveElement

Prototype

```
vmiSaveRestoreStatus vmirtSaveElement(
    vmiSaveContextP cxt,
    const char     *elementName,
    const char     *endName,
    const void     *value,
    Uns32           bytes
);
```

Description

This function saves a member of a list of values to the active save context `cxt`. Each value is of size `bytes`. If value is non-`NULL`, then the value is written, preceded by the `elementName` key. If the value is `NULL`, then this indicates the end of the list, and the key `endName` is written. Every list must consist of a series of zero or more non-`NULL` entries and a single `NULL` terminating entry. The `elementName` and `endName` strings must match the corresponding keys used by a later call to `vmirtRestoreElement`.

Example

This example is from the OVP ARM processor model to save TLB entries:

```
static void saveTLBEntry(
    armP            arm,
    vmiSaveContextP cxt,
    tlbEntryP       entry,
    Uns32           entryBytes
) {
    // save entry
    tlbEntry entryS = *entry;

    // clear down properties used to manage mapping
    entryS.isMapped = 0;
    entryS.lutEntry = 0;

    // copy least-significant 8 bits of VMID to legacy position if required
    // (simASIDMSB was 8-bit VMID originally)
    if(!saveTLBEntryVMID16State(arm)) {
        entryS.simASIDMSB = entryS.VMID;
    }

    vmirtSaveElement(cxt, ARM_TLB_ENTRY, ARM_TLB_END, &entryS, entryBytes);
}

static void saveTLB(armP arm, vmiSaveContextP cxt, armTLBP tlb) {

    armTLBLDP tlbld      = tlb->tlbLD;
    Uns32     entryBytes = getTLBEntrySRSize(arm, tlb->id);

    // save TLB name
    vmirtSave(cxt, tlb->name, 0, 0);

    // save TLB lockdown section
    if(tlbld) {

        Uns32 i;

        // restore lockdown entry indices
        VMIRT_SAVE_FIELD(cxt, arm, TLBLDRI);
        VMIRT_SAVE_FIELD(cxt, arm, TLBLDWI);
        // save all lockdown entries (even if invalid)
        for(i=0; i<tlbld->ldSize; i++) {
            tlbEntryP entry = getLockDownEntry(tlbld, i);
            saveTLBEntry(arm, cxt, entry, entryBytes);
        }
    }

    // save all normal TLB entries
    ITER_TLB_ENTRY_RANGE(
        arm, tlb, 0, ARM_MAX_ADDR, entry,
        if(!entry->LDV) {
            saveTLBEntry(arm, cxt, entry, entryBytes);
        }
    );

    // save terminator
    vmirtSaveElement(cxt, ARM_TLB_ENTRY, ARM_TLB_END, 0, 0);
}
```

Notes and Restrictions

None.

20.4 vmirtRestoreElement

Prototype

```
vmiSaveRestoreStatus vmirtRestoreElement(
    vmiRestoreContextP cxt,
    const char        *elementName,
    const char        *endName,
    void              *value,
    Uns32              bytes
);
```

Description

This function restores a member of a list of values from the active restore context `cxt`. Each value is of size `bytes`. The `elementName` and `endName` strings must match the corresponding keys used by an earlier call to `vmirtSaveElement`. The return value is of type `vmiSaveRestoreStatus`:

```
typedef enum vmiSaveRestoreStatusE {
    SRS_ERROR,      // error in save/restore restore
    SRS_OK,         // successful operation
    SRS_END_LIST    // end of list (vmirtSaveElement/vmirtRestoreElement only)
} vmiSaveRestoreStatus;
```

As each element is restored, the value `SRS_OK` is returned. At the end of the list, `SRS_END_LIST` is returned and `value` is not updated.

Example

This example is from the OVP ARM processor model to restore TLB entries:

```
static void restoreTLB(armP arm, vmiRestoreContextP cxt, armTLBP tlb) {

    armTLBLDP tlbld      = tlb->tlbLD;
    Uns32     entryBytes = getTLBEntrySRSize(arm, tlb->id);

    // temporary entry must be zeroed out because the top part may be unmodified
    // by restore (if not a stage 1 non-secure entry)
    tlbEntry new = {0};

    // set S2AP to default value indicating RWX access
    new.v.S2AP = getStage2APRWX();

    // restore TLB name
    vmirtRestore(cxt, tlb->name, 0, 0);

    // restore TLB lockdown section
    if(tlbld) {

        Uns32 i;

        // restore lockdown entry indices
        VMIRT_RESTORE_FIELD(cxt, arm, TLBLDRI);
        VMIRT_RESTORE_FIELD(cxt, arm, TLBLDWI);

        // restore all lockdown entries
        for(i=0; i<tlbld->ldSize; i++) {
            tlbEntryP entry = getLockDownEntry(tlbld, i);
            vmirtRestore(cxt, ARM_TLB_ENTRY, entry, entryBytes);
            restoreTLBEntryAttrs(arm, tlb, entry);
        }
    }
    // restore all normal TLB entries
    while(
        vmirtRestoreElement(cxt, ARM_TLB_ENTRY, ARM_TLB_END, &new, entryBytes) ==
        SRS_OK
    ) {
        tlbEntryP entry = newTLBEntry(tlb);
        restoreTLBEntry(arm, tlb, entry, &new);
    }
}
```

Notes and Restrictions

None.

20.5 vmirtSaveModelTimer

Prototype

```
vmiSaveRestoreStatus vmirtSaveModelTimer(
    vmiSaveContextP cxt,
    const char     *name,
    vmiModelTimerP  modelTimer
);
```

Description

This function saves a timer object to the active save context `cxt` with key `name`. The `name` string must match the corresponding name used by a later call to `vmirtRestoreModelTimer`.

Example

This example is from the OVP ARM processor model to save model timers:

```
static void saveTimers(armP arm, vmiSaveContextP cxt) {

    // save CCNT timer if required
    if(arm->timerCCNT) {
        vmirtSaveModelTimer(cxt, "timerCCNT", arm->timerCCNT);
    }

    // save PMN0 timer if required
    if(arm->timerPMN0) {
        vmirtSaveModelTimer(cxt, "timerPMN0", arm->timerPMN0);
    }

    // save PMN1 timer if required
    if(arm->timerPMN1) {
        vmirtSaveModelTimer(cxt, "timerPMN1", arm->timerPMN1);
    }

    // save VALDEBUGCNT timer if required
    if(arm->timerVALDEBUGCNT) {
        vmirtSaveModelTimer(cxt, "timerVALDEBUGCNT", arm->timerVALDEBUGCNT);
    }
}
```

Notes and Restrictions

None.

20.6 vmirtRestoreModelTimer

Prototype

```
vmiSaveRestoreStatus vmirtRestoreModelTimer(
    vmiRestoreContextP cxt,
    const char        *name,
    vmiModelTimerP     modelTimer
);
```

Description

This function restores a timer object from the active restore context `cxt`. The `name` string must match the corresponding name used by an earlier call to `vmirtSaveModelTimer`.

Example

This example is from the OVP ARM processor model to restore model timers:

```
static void restoreTimers(armP arm, vmiRestoreContextP cxt, Uns32 srVersion) {

    // restore CCNT timer if required (present for ARMv7 only from version 4 of
    // save/restore API)
    if(!arm->timerCCNT) {
        // no action
    } else if((srVersion>=4) || arm->configInfo.sysRegPresent.CCNT) {
        vmirtRestoreModelTimer(cxt, "timerCCNT", arm->timerCCNT);
    }

    // restore PMN0 timer if required
    if(arm->timerPMN0) {
        vmirtRestoreModelTimer(cxt, "timerPMN0", arm->timerPMN0);
    }

    // restore PMN1 timer if required
    if(arm->timerPMN1) {
        vmirtRestoreModelTimer(cxt, "timerPMN1", arm->timerPMN1);
    }

    // restore VALDEBUGCNT timer if required
    if(arm->timerVALDEBUGCNT) {
        vmirtRestoreModelTimer(cxt, "timerVALDEBUGCNT", arm->timerVALDEBUGCNT);
    }
}
```

Notes and Restrictions

None.

20.7 vmirtSaveDomain

Prototype

```
vmiSaveRestoreStatus vmirtSaveDomain(
    vmiSaveContextP cxt,
    memDomainP      domain
);
```

Description

This function saves the contents of any privately-allocated memory domain (`memDomainP`) objects to the active save context `cxt`.

Example

This example is from the OVP ARC processor model. It is used to save the contents of model-allocated closely-coupled memories (CCMs)

```
static void saveCCM(arcDomainSetP domainSet, vmiSaveContextP cxt) {

    Uns32 i;

    for(i=0; i<domainSet->ccmNum; i++) {

        arcCCMInfoP ccm = &domainSet->ccms[i];

        if(ccm && ccm->internal) {
            vmirtSaveDomain(cxt, ccm->port->vmiPort.domain);
        }
    }
}
```

Notes and Restrictions

None.

20.8 vmirtRestoreDomain

Prototype

```
vmiSaveRestoreStatus vmirtRestoreDomain(
    vmiRestoreContextP cxt,
    memDomainP         domain
);
```

Description

This function restores the contents of any privately-allocated memory domain (`memDomainP`) objects from the active restore context `cxt`.

Example

This example is from the OVP ARC processor model. It is used to restore the contents of model-allocated closely-coupled memories (CCMs)

```
static void restoreCCM(arcDomainSetP domainSet, vmiRestoreContextP cxt) {

    Uns32 i;

    for(i=0; i<domainSet->ccmNum; i++) {

        arcCCMInfoP ccm = &domainSet->ccms[i];

        if(ccm && ccm->internal) {
            vmirtRestoreDomain(cxt, ccm->port->vmiPort.domain);
        }
    }
}
```

Notes and Restrictions

None.

20.9 vmirtGetPostSlotCB

Prototype

```
vmiPostSlotFn vmirtGetPostSlotCB(vmiProcessorP processor);
```

Description

When a processor is currently executing a delay slot instruction, this function returns any active post-delay-slot callback function for that processor. This is required so that information about the callback can be saved.

Example

This example is from the OVP MIPS processor model. If there is an active callback, an identification code for it is saved to the active save context:

```
void mipsSavePostSlotCB(vmiProcessorP processor, vmiSaveContextP cxt) {

    vmiPostSlotFn slotCB   = vmirtGetPostSlotCB(processor);
    Uns32         slotCBId = 0;

    // convert from slotCB to slotCBId
    if(!slotCB) {
        // no action
    } else if(slotCB==postDelaySlotJALX_ISA0) {
        slotCBId = 1;
    } else if(slotCB==postDelaySlotJALX_ISA1) {
        slotCBId = 2;
    } else if(slotCB==postDelaySlotJR) {
        slotCBId = 3;
    } else {
        VMI_ABORT("unimplemented slotCB %p", slotCB); // LCOV_EXCL_LINE
    }

    // save slotCBId
    VMIRT_SAVE_REG(cxt, "slotCB", &slotCBId);
}
```

Notes and Restrictions

None.

20.10 vmirtSetPostSlotCB

Prototype

```
void vmirtSetPostSlotCB(vmiProcessorP processor, vmiPostSlotFn slotCB);
```

Description

This function allows the active post-delay-slot callback function for a processor to be updated. This is required when a processor is restored to a state in which it is executing a delay slot instruction.

Example

This example is from the OVP MIPS processor model:

```
void mipsRestorePostSlotCB(vmiProcessorP processor, vmiRestoreContextP cxt) {

    vmiPostSlotFn slotCB   = 0;
    Uns32         slotCBId = 0;

    // restore slotCBId
    VMIRT_RESTORE_REG(cxt, "slotCB", &slotCBId);

    // convert from slotCBId to slotCB
    if(!slotCBId) {
        // no action
    } else if(slotCBId==1) {
        slotCB = postDelaySlotJALX_ISA0;
    } else if(slotCBId==2) {
        slotCB = postDelaySlotJALX_ISA1;
    } else if(slotCBId==3) {
        slotCB = postDelaySlotJR;
    } else {
        VMI_ABORT("unimplemented slotCBId %u", slotCBId); // LCOV_EXCL_LINE
    }

    // restore slotCB
    vmirtSetPostSlotCB(processor, slotCB);
}
```

Notes and Restrictions

None.

21 Application Program Symbol Table Access

This section describes functions allowing symbol tables of applications program loaded by processor models to be accessed.

These functions can be very useful when writing debug routines. They are also of use when writing processor semihosting plugins. Most of these routines are available only with Imperas Professional Tools.

21.1 vmirtAddressLookup

Prototype

```
memDomainP vmirtAddressLookup(
    vmiProcessorP processor,
    const char    *name,
    Addr          *simAddr
);
```

Description

Given a processor and a symbol name, this function searches for the name in any object file or symbol file directly loaded onto the processor or its external code or data domains. If a symbol of the given name is found, the function sets byref argument `simAddr` to the symbol load address and returns the memory domain object into which the object file was originally loaded. If the name cannot be found, the function returns `NULL`.

Example

This example is from the OR1K Newlib semihost library.

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    Uns32 i;

    // first few argument registers (standard ABI)
    object->args[0] = vmiosGetRegDesc(processor, "R3");
    object->args[1] = vmiosGetRegDesc(processor, "R4");
    object->args[2] = vmiosGetRegDesc(processor, "R5");

    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "R11");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "R1");

    // __impure_ptr address
    object->impurePtrDomain = vmirtAddressLookup(
        processor, ERRNO_REF, &object->impurePtrAddr
    );

    // initialize stdin, stderr and stdout
    object->fileDescriptors[0] = vmiosGetStdin(processor);
    object->fileDescriptors[1] = vmiosGetStdout(processor);
    object->fileDescriptors[2] = vmiosGetStderr(processor);

    // initialize remaining file descriptors
    for(i=3; i<FILE_DES_NUM; i++) {
        object->fileDescriptors[i] = -1;
    }
}
```

Notes and Restrictions

1. See also `vmirtGetSymbolByAddr` in this section, which provides more detailed information about the symbol at a specified address.
2. This routine is the only one in this section that is available *without* an Imperas Professional Tools license.

QuantumLeap Semantics
Synchronizing

21.2 vmirtSymbolLookup

Prototype

```
const char *vmirtSymbolLookup(
    vmiProcessorP processor,
    Addr          simAddr,
    Offset       *offset
);
```

Description

Given a processor and an address, this function searches for the *nearest* symbol to the address in any object file loaded onto the processor. If a nearby symbol is found, the function returns the name of the symbol, and byref argument `offset` is updated to give the offset of the address from the symbol address. If the address is not found in any object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

Example

This example shows a function returning the name of the function in which a processor is executing:

```
static const char *getAddrFunction(vmiosObjectP object, Addr address) {
    Offset offset;
    return vmirtSymbolLookup(object->processor, address, &offset);
}

static const char *getCurrentFunction(vmiosObjectP object) {
    Addr   simPC = vmirtGetPC(object->processor);
    return getAddrFunction(object, simPC);
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

21.3 *vmirtAddSymbolFile*

Prototype

```
vmiSymbolFileCP vmirtAddSymbolFile(
    vmiProcessorP processor,
    const char   *filename,
    Addr          address,
    const char   *sections,
    Bool          privateTable
);
```

Description

An application program file is used by the simulator to load the program, to obtain the symbols for symbolic interception and by the product's debugger to allow symbolic debug.
The latter functionality is available only if the program is loaded explicitly by the simulator. If the program is loaded by other means, its symbols must be loaded explicitly.

This function attempts to load the symbol table from the given file into a debugger attached to the given processor and also into the simulator's symbol tables. It allows the product's multi-processor debugger to debug a program that was not loaded by the simulator (i.e. the program is unknown to the debugger). It returns a handle to the symbol file if it was successful. It relies on the gdb command:

```
    add-symbol-file <file> <address> [sections]
```

Where `<file> <address>` and `[sections]` are the arguments to `vmirtAddSymbolFile`. The optional [sections] string should be of the form:

```
    -s <section> <section-address>  -s <section> <section-address> …
```

If the `privateTable` argument is true then the symbols will not be available to the simulator, only to functions that use the handle returned by this function.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiMessage.h”

{
    vmiSymbolFileCP file = vmirtAddSymbolFile(
        processor,
        “myProg.elf”,
        0x8000,
        “-s text 0x9000 –s init 0x9800”,
        True
    );

    if(file == NULL) {
        vmiPrintf(“Unable to load\n”);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The gdb documentation explains the purpose of the `–s` argument to `add-symbol-file`.

QuantumLeap Semantics

Synchronizing

21.4 *vmirtNextSymbolFile*

Prototype

```
vmiSymbolFileCP vmirtNextSymbolFile(
    vmiProcessorP   processor,
    vmiSymbolFileCP prev
);
```

Description

This function is an iterator that, given a symbol file associated with a processor, returns the *next* symbol file associated with that processor. If `NULL` is passed as the `prev` parameter, the function returns the first symbol file associated with the processor.

Symbol files added directly to the passed processor are returned first. After this, symbol files of any parent are returned, and so on up the processor hierarchy.

The `vmiSymbolFileCP` object may be used by functions `vmirtGetSymbolFileName`, `vmirtNextSymbolByNameFile`, `vmirtNextSymbolByAddrFile`, `vmirtPrevSymbolByAddrFile`, `vmirtNextFLByAddrFile` and `vmirtPrevFLByAddrFile`.

Example

```
// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByAddrFile(file, symbol))) {
            vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
            vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
            vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
            vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
            vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolFileCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

21.5 *vmirtGetSymbolFileName*

Prototype

```
const char *vmirtGetSymbolFileName(vmiSymbolFileCP symbolFile);
```

Description

This function returns the name of a symbol file found by `vmirtNextSymbolFile`.

Example

```
// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByAddrFile(file, symbol))) {
            vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
            vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
            vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
            vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
            vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

21.6 vmirtGetSymbolByName

Prototype

```
vmiSymbolCP vmirtGetSymbolByName(
    vmiProcessorP processor,
    const char    *name
);
```

Description

Given a processor and a symbol name, this function searches for the name in all object files loaded by the processor. If the symbol is found in an object file, the function returns an object of type `vmiSymbolCP` describing the symbol. If the symbol is not found in the object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write a debug message giving address and details of given symbol
static void vmic_PrintSymbol(cpuxP cpux, const char *name) {

    vmiSymbolCP symbol = vmirtGetSymbolByName((vmiProcessorP)cpux, name);

    if(symbol) {
        vmiPrintf(“Symbol ‘%s’:\n”, name);
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    } else {
        vmiPrintf(“Symbol ‘%s’ is unknown\n”, name);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

21.7 vmirtGetSymbolByNameFile

Prototype

```
vmiSymbolCP vmirtGetSymbolByNameFile(
    vmiSymbolFileCP file,
    const char     *name
);
```

Description

Given a symbol file and a symbol name, this function searches for the name in the symbols in that file. If the symbol is found, the function returns an object of type `vmiSymbolCP` describing the symbol. If the symbol is not found in the file the function returns `NULL`.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

The file descriptor `vmiSymbolFileCP` is returned by `vmirtAddSymbolFile` and `vmirtNextSymbolFile.`

Example

```
// write a debug message giving address and details of given symbol

static void vmic_PrintSymbol(vmiSymbolFileCP file, const char *name) {

    vmiSymbolCP symbol = vmirtGetSymbolByNameFile(file, name);

    if(symbol) {
        vmiPrintf("Symbol '%s':\n", name);
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    } else {
        vmiPrintf("Symbol '%s' is unknown\n", name);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

21.8 vmirtGetSymbolByAddr

Prototype

```
vmiSymbolCP vmirtGetSymbolByAddr(vmiProcessorP processor, Addr simAddr);
```

Description

Given a processor and an address, this function searches for the address in any object file loaded by the processor. If the address is defined within the object file, the function returns an object of type `vmiSymbolCP` describing the object at that address. If the address is not found in the object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write debug message giving current execution scope
static void vmic_PrintScope(cpuxP cpux) {

    // get current program counter
    Addr        simPC  = vmirtGetPC((vmiProcessorP)cpux);
    vmiSymbolCP symbol = vmirtGetSymbolByAddr((vmiProcessorP)cpux, simPC);

    if(symbol) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    } else {
        vmiPrintf("Scope of 0x%llx is unknown\n", simPC);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

21.9 vmirtGetSymbolByAddrFile

Prototype

```
vmiSymbolCP vmirtGetSymbolByAddrFile(vmiSymbolFileCP file, Addr simAddr);
```

Description

Given a symbol file and an address, this function searches for the address in the symbols in that file. If the address is defined within the object file, the function returns an object of type `vmiSymbolCP` describing the object at that address. If the address is not found in the object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

The file descriptor `vmiSymbolFileCP` is returned by `vmirtAddSymbolFile` and `vmirtNextSymbolFile.`

Example

```
// write debug message giving current execution scope
static void vmic_PrintSymbol (vmiSymbolFileCP file, Addr address) {

    // get current program counter
    vmiSymbolCP symbol = vmirtGetSymbolByAddrFile(file, address);

    if(symbol) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    } else {
        vmiPrintf("No symbol found\n");
    }
}
```

Notes and Restrictions

3. This routine is only available for products with an Imperas Professional Tools license.
4. The `vmiSymbolCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

21.10 vmirtNextSymbolByName

Prototype

```
vmiSymbolCP vmirtNextSymbolByName(
    vmiProcessorP processor,
    vmiSymbolCP   prev
);
```

Description

This function is an iterator that, given a symbol in an object file associated with a processor, returns the *next* symbol in that object file in alphabetic order. If `NULL` is passed as the `prev` parameter, the function returns the alphabetically first symbol found in the first object file associated with the processor.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByName(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByName((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.
4. To get symbols for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextSymbolByNameFile` functions instead.

QuantumLeap Semantics

Synchronizing

21.11 vmirtNextSymbolByAddr

Prototype

```
vmiSymbolCP vmirtNextSymbolByAddr(
    vmiProcessorP processor,
    vmiSymbolCP   prev
);
```

Description

This function is an iterator that, given a symbol in an object file associated with a processor, returns the *next* symbol in that object file in increasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the first symbol found in the first object file associated with the processor.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.
4. To get symbols for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextSymbolByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

21.12 vmirtPrevSymbolByAddr

Prototype

```
vmiSymbolCP vmirtPrevSymbolByAddr(
    vmiProcessorP processor,
    vmiSymbolCP   prev
);
```

Description

This function is an iterator that, given a symbol in an object file associated with a processor, returns the *previous* symbol in that object file in decreasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the last symbol found in the first object file associated with the processor.
The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByReverseAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtPrevSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.
4. To get symbols for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtPrevSymbolByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

21.13 vmirtNextSymbolByNameFile

Prototype

```
vmiSymbolCP vmirtNextSymbolByNameFile(
    vmiSymbolFileCP symbolFile,
    vmiSymbolCP     prev
);
```

Description

This function is an iterator that, given a symbol in an object file, returns the *next* symbol in that object file in alphabetic order. If `NULL` is passed as the `prev` parameter, the function returns the alphabetically first symbol found in the first object file.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByName(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByNameFile(file, symbol))) {
            vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
            vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
            vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
            vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
            vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.

QuantumLeap Semantics

Synchronizing

21.14 vmirtNextSymbolByAddrFile

Prototype

```
vmiSymbolCP vmirtNextSymbolByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiSymbolCP     prev
);
```

Description

This function is an iterator that, given a symbol in an object file, returns the *next* symbol in that object file in increasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the first symbol found in the first object file.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByAddrFile(file, symbol))) {
            vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
            vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
            vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
            vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
            vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.

QuantumLeap Semantics

Synchronizing

21.15 vmirtPrevSymbolByAddrFile

Prototype

```
vmiSymbolCP vmirtPrevSymbolByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiSymbolCP     prev
);
```

Description

This function is an iterator that, given a symbol in an object file, returns the *previous* symbol in that object file in decreasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the last symbol found in the object file.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

```
// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByReverseAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((symbol=vmirtPrevSymbolByAddrFile(file, symbol))) {
            vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
            vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
            vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
            vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
            vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.

QuantumLeap Semantics

Synchronizing

21.16 vmirtGetSymbolName

Prototype

```
const char *vmirtGetSymbolName(vmiSymbolCP symbol);
```

Description

This function returns the symbol name of a `vmiSymbolCP` object.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The returned string will persist until the simulation finishes.

QuantumLeap Semantics

Synchronizing

21.17 vmirtGetSymbolAddr

Prototype

```
Addr vmirtGetSymbolAddr(vmiSymbolCP symbol);
```

Description

This function returns the *virtual* address of a `vmiSymbolCP` object, i.e. the address at which of that symbol is expected to be found in an executing program. Note that this can be distinct from the *load* address of the symbol in some cases.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. See also function `vmirtGetSymbolLoadAddr`, which returns the symbol *load* address.

QuantumLeap Semantics

Synchronizing

21.18 vmirtGetSymbolLoadAddr

Prototype

```
Addr vmirtGetSymbolLoadAddr(vmiSymbolCP symbol);
```

Description

This function returns the *load* address of a `vmiSymbolCP` object, i.e. the address at which that symbol was *loaded into memory*. Note that this can be distinct from the *virtual* address of the symbol in some cases.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. See also function `vmirtGetSymbolAddr`, which returns the symbol *virtual* address.

QuantumLeap Semantics

Synchronizing

21.19 vmirtGetSymbolType

Prototype

```
ordSymbolType vmirtGetSymbolType(vmiSymbolCP symbol);
```

Description

This function returns the type of a `vmiSymbolCP` object. The type is a member of an enumeration defined in `ImpPublic/include/host/ord/ordTypes.h`:

```
typedef enum ordSymbolTypeE {
    ORD_SYMBOL_TYPE_SECTION,
    ORD_SYMBOL_TYPE_NONE,
    ORD_SYMBOL_TYPE_OBJECT,
    ORD_SYMBOL_TYPE_FUNC,
    ORD_SYMBOL_TYPE_FILE
} ordSymbolType;
```

Symbol type information is typically used when writing introspection or profiling utilities.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. Symbol types are not available in all file formats. The type will default to `ORD_SYMBOL_TYPE_NONE`.

QuantumLeap Semantics

Synchronizing

21.20 vmirtGetSymbolBinding

Prototype

```
ordSymbolBinding vmirtGetSymbolBinding(vmiSymbolCP symbol);
```

Description

This function returns the binding of a `vmiSymbolCP` object. The type is a member of an enumeration defined in `ImpPublic/include/host/ord/ordTypes.h`:

```
typedef enum ordSymbolBindingE {
    ORD_SYMBOL_BIND_LOCAL,
    ORD_SYMBOL_BIND_WEAK,
    ORD_SYMBOL_BIND_GLOBAL
} ordSymbolBinding;
```

Symbol binding information is typically used when writing introspection or profiling utilities.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. Binding is not reported by all file formats. The result defaults to `ORD_SYMBOL_BIND_LOCAL`.

QuantumLeap Semantics

Synchronizing

21.21 *vmirtGetSymbolSize*

Prototype

```
Addr vmirtGetSymbolSize(vmiSymbolCP symbol);
```

Description

This function returns the size in bytes of the symbol specified by a `vmiSymbolCP` object.

Example

```
// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. Depending on the symbol type, the size might correspond to the size of a represented data object, the length of a function or the total size of a section. Not all symbol types have a specified size, depending on the file format. The size will default to zero.

QuantumLeap Semantics

Synchronizing

22 Application Dwarf Line Number Information Access

This section describes functions allowing Dwarf-format line number information from an of the application program loaded by a processor model to be accessed. This information is only present if the application program has been compiled for debug.

These functions can be very useful when writing debug routines. They are also of use when writing processor semihosting plugins. These routines are available only with Imperas Professional Tools.

22.1 vmirtGetFLByAddr

Prototype

```
vmiFileLineCP vmirtGetFLByAddr(vmiProcessorP processor, Addr simAddr);
```

Description

Given a processor and an address, this function searches for the address in the Dwarf information for any object file loaded by the processor. If the address is found, the function returns an object of type `vmiFileLineCP` describing the object at that address. If the address is not found in the object file, or the processor had no object file specified when it was loaded, or the application was not compiled for debug, the function returns `NULL`.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

```
// write debug message giving current line
static void vmic_CurrentLine(cpuxP cpux) {

    // get current program counter
    Addr          simPC = vmirtGetPC((vmiProcessorP)cpux);
    vmiFileLineCP fl    = vmirtGetFLByAddr(((vmiProcessorP)cpux, simPC);

    if(fl) {
        vmiPrintf("Scope of 0x%llx is:\n", simPC);
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    } else {
        vmiPrintf("Scope of 0x%llx is unknown\n", simPC);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

22.2 vmirtGetFLByAddrFile

Prototype

```
vmiFileLineCP vmirtGetFLByAddrFile(
    vmiSymbolFileCP symbolFile,
    Addr simAddr
);
```

Description

Given a loaded symbol file and an address, this function searches for the address in the Dwarf information. If the address is found, the function returns an object of type `vmiFileLineCP` describing the object at that address. If the address is not found in the object file, or the application was not compiled for debug, the function returns `NULL`.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

```
// write debug message giving current line
static void lookup (vmiProcessorP processor) {

    vmiSymbolFileCP file = vmirtAddSymbolFile(
        processor,
        “myProg.elf”,
        0x8000,
        0,
        True
    );

    // get current program counter
    Addr          simPC = vmirtGetPC(processor);
    vmiFileLineCP fl    = vmirtGetFLByAddrFile(file, simPC);

    if(fl) {
        vmiPrintf(“Scope of 0x%llx is:\n”, simPC);
        vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
        vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
        vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
    } else {
        vmiPrintf(“Scope of 0x%llx is unknown\n”, simPC);
    }
}
```

Notes and Restrictions

3. This routine is only available for products with an Imperas Professional Tools license.
4. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

22.3 vmirtNextFLByAddr

Prototype

```
vmiFileLineCP vmirtNextFLByAddr(
    vmiProcessorP processor,
    vmiFileLineCP prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file associated with the processor to be processed in increasing simulated address order.

The first file/line record in the first object file associated with the processor is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

To get file/line records for all object files associated with the processor use the `vmirtNextSymbolFile` and `vmirtNextFLByAddrFile` functions instead.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf("New FL record:\n");
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.
3. To get file/line records for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextFLByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

22.4 vmirtPrevFLByAddr

Prototype

```
vmiFileLineCP vmirtNextFLByAddr(
    vmiProcessorP processor,
    vmiFileLineCP prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file associated with the processor to be processed in decreasing simulated address order.

The last file/line record in the first object file associated with the processor is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByReverseAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtPrevFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf("New FL record:\n");
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.
3. To get file/line records for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextFLByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

22.5 vmirtNextFLByAddrFile

Prototype

```
vmiFileLineCP vmirtNextFLByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiFileLineCP   prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file to be processed in increasing simulated address order. The first file/line record is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiFileLineCP fl = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((fl=vmirtNextFLByAddrFile(file, fl))) {
            vmiPrintf("New FL record:\n");
            vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
            vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
            vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

22.6 vmirtPrevFLByAddrFile

Prototype

```
vmiFileLineCP vmirtNextFLByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiFileLineCP   prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file to be processed in decreasing simulated address order. The last file/line record in the first object file associated with the processor is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByReverseAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiFileLineCP fl = 0;

        vmiPrintf("new symbol file %s\n", vmirtGetSymbolFileName(file));

        while((fl=vmirtPrevFLByAddrFile(file, fl))) {
            vmiPrintf("New FL record:\n");
            vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
            vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
            vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

22.7 vmirtGetFLFileName

Prototype

```
const char *vmirtGetFLFileName(vmiFileLineCP fl);
```

Description

This function returns the file name of a `vmiFileLineCP` object. This is the file name of the file in which the item corresponding to the file/line record was defined.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf("New FL record:\n");
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

22.8 vmirtGetFLLineNumber

Prototype

```
Uns32 vmirtGetFLLineNumber(vmiFileLineCP fl);
```

Description

This function returns the line number of a `vmiFileLineCP` object. This is the line number in the file at which the item corresponding to the file/line record was defined.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf("New FL record:\n");
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

22.9 vmirtGetFLAddr

Prototype

```
Addr vmirtGetFLAddr(vmiFileLineCP fl);
```

Description

This function returns the simulated address of a `vmiFileLineCP` object. This is the address in the application executable file at which the item corresponding to the file/line record is located.

Example

```
// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf("New FL record:\n");
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

23 Licensing

This section describes functions allowing models to be licensed using the Imperas license manager daemon. In order to use these functions, a license key for the model needs to be created for use with the Imperas daemon – contact Imperas for more information.

23.1 vmirtGetLicense

Prototype

```
Bool vmirtGetLicense(const char *name);
```

Description

This routine attempts to check out a license for the current model. The `Bool` return code indicates whether the license was successfully checked out. This function ***must*** be called from within a model constructor. If the `name` argument is `NULL`, then the license feature is derived from the simulator product name and the variant or name of the model, otherwise the `name` string is checked out from the license server.

Example

```
//
// Processor constructor
//
VMI_CONSTRUCTOR_FN(cpuxConstructor) {

    // Check for valid CPU license
    const char *name = "CPUX_LICENSE_FEATURE"
    if (!vmirtGetLicense(name)) {
        vmiMessage("F", "LIC_NA2", "%s", vmirtGetLicenseErrString(name));
    }

    . . . etc . . .
}
```

Notes and Restrictions

1. Must be called from the model constructor.

QuantumLeap Semantics

Synchronizing

23.2 vmirtGetLicenseErrString

Prototype

```
const char *vmirtGetLicenseErrString(const char *name);
```

Description

When function `vmirtGetLicense` fails, this function can be called to get an error string indicating why the checkout failed. Typically, the result should be used in a call to `vmiMessage` with the fatal message identifier `"F"`: this will cause a *fatal* message to be printed and terminate simulation.

The `name` argument must be the same as what was passed on the failing `vmirtGetLicense` call - either `NULL` or the name of the license to check out.

Example

```
//
// Processor constructor
//
VMI_CONSTRUCTOR_FN(cpuxConstructor) {

    // Check for valid CPU license
    const char *name = "CPUX_LICENSE_FEATURE"
    if (!vmirtGetLicense(name)) {
        vmiMessage("F", "LIC_NA2", "%s", vmirtGetLicenseErrString(name));

    }
    . . . etc . . .
}
```

Notes and Restrictions

1. Must be called from the model constructor.
2. Should only be called after `vmirtGetLicense` has failed.
3. Must be passed the same argument that was used on the failed `vmirtGetLicense` call.

QuantumLeap Semantics

Synchronizing

24 View Provider Interface

This section describes functions to create and provide *view objects*. These objects are used to communicate abstract information about processor state and state changes to debuggers and other clients.

24.1 vmirtGetProcessorViewObject

Prototype

```
vmiViewObjectP vmirtGetProcessorViewObject(vmiProcessorP processor);
```

Description

Every processor has a *root view object* with which a hierarchy of model-specific view objects and events can be associated. This function returns the root view object for a processor.

Example

The OVP OR1K processor training examples use this function as follows:

```
void or1kCreateView(or1kP or1k) {

    // get the base processor view object
    vmiProcessorP  processor       = (vmiProcessorP)or1k;
    vmiViewObjectP processorObject = vmirtGetProcessorViewObject(processor);

    // add new view object
    or1k->exObject  = vmirtAddViewObject(
        processorObject, "addressException", "Address exception"
    );

    // Create an event to be generated on an address exception
    or1k->addrExEvent = vmirtAddViewEvent(
        or1k->exObject, "address", "Address exception event trigger"
    );

    // Create an object to access the EEAR
    vmiViewObjectP eearObject = vmirtAddViewObject(or1k->exObject, "eear", "");
    vmirtSetViewObjectRefValue(eearObject, VMI_VVT_UNS32, &or1k->EEAR);

    // Create an object to access the EPC
    vmiViewObjectP epcObject = vmirtAddViewObject(or1k->exObject, "epc", "");
    vmirtSetViewObjectRefValue(epcObject, VMI_VVT_UNS32, &or1k->EPC);

    // Create an object to access the ESR
    vmiViewObjectP esrObject = vmirtAddViewObject(or1k->exObject, "esr", "");
    vmirtSetViewObjectRefValue(esrObject, VMI_VVT_UNS32, &or1k->ESR);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.2 vmirtAddViewObject

Prototype

```
vmiViewObjectP vmirtAddViewObject(
    vmiViewObjectP parent,
    const char     *name,
    const char     *description
);
```

Description

This function creates a new view object which is a child of the `parent` object. The description argument is used to describe the view object (and can be `NULL`).

Example

The OVP OR1K processor training examples use this function as follows:

```
void or1kCreateView(or1kP or1k) {

    // get the base processor view object
    vmiProcessorP  processor       = (vmiProcessorP)or1k;
    vmiViewObjectP processorObject = vmirtGetProcessorViewObject(processor);

    // add new view object
    or1k->exObject  = vmirtAddViewObject(
        processorObject, "addressException", "Address exception"
    );

    // Create an event to be generated on an address exception
    or1k->addrExEvent = vmirtAddViewEvent(
        or1k->exObject, "address", "Address exception event trigger"
    );

    // Create an object to access the EEAR
    vmiViewObjectP eearObject = vmirtAddViewObject(or1k->exObject, "eear", "");
    vmirtSetViewObjectRefValue(eearObject, VMI_VVT_UNS32, &or1k->EEAR);

    // Create an object to access the EPC
    vmiViewObjectP epcObject = vmirtAddViewObject(or1k->exObject, "epc", "");
    vmirtSetViewObjectRefValue(epcObject, VMI_VVT_UNS32, &or1k->EPC);

    // Create an object to access the ESR
    vmiViewObjectP esrObject = vmirtAddViewObject(or1k->exObject, "esr", "");
    vmirtSetViewObjectRefValue(esrObject, VMI_VVT_UNS32, &or1k->ESR);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.3 vmirtSetViewObjectConstValue

Prototype

```
void vmirtSetViewObjectConstValue(
    vmiViewObjectP   object,
    vmiViewValueType type,
    void             *pValue
);
```

Description

This function assigns a *constant value* to the view object, copied from the `pValue` argument. The `type` argument, indicates the value type, using one of the highlighted members of the `vmiViewValueType` enumeration:

```
typedef enum vmiViewValueTypeE {

    VMI_VVT_NOSPACE   = -1, // No value returned. More buffer required.
    VMI_VVT_ERROR     = 0,  // No value returned

    VMI_VVT_BOOL      = 1,  // Primitive types
    VMI_VVT_SCHAR     = 2,
    VMI_VVT_UCHAR     = 3,
    VMI_VVT_INT8      = 4,
    VMI_VVT_UNS8      = 5,
    VMI_VVT_INT16     = 6,
    VMI_VVT_UNS16     = 7,
    VMI_VVT_INT32     = 8,
    VMI_VVT_UNS32     = 9,
    VMI_VVT_INT64     = 10,
    VMI_VVT_UNS64     = 11,
    VMI_VVT_ADDR      = 12,
    VMI_VVT_FLT64     = 13,

    VMI_VVT_STRING    = 14,  // Single line string
    VMI_VVT_MULTILINE = 15,  // Multi line string

} vmiViewValueType;
```

Example

The OVP MIPS processor model uses this function to expose the name of the processor configuration:

```
void mipsNew(
    mipsP            mips,
    vmiSMPContextP   smpContext,
    mipsParamValuesP params
) {
    vmiProcessorP  processor  = (vmiProcessorP)mips;
    mipsP          parent     = (mipsP)vmirtGetSMPParent(processor);
    vmiViewObjectP baseObject = vmirtGetProcessorViewObject(processor);
    mipsConfigCP   config     = mips->config;

    . . . lines omitted for clarity . . .

    // add the config name to the processor's view
    vmiViewObjectP configName = vmirtAddViewObject(baseObject, "config", 0);
    vmirtSetViewObjectConstValue(configName, VMI_VVT_STRING, (void*)config->name);

    . . . lines omitted for clarity . . .
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

24.4 vmirtSetViewObjectRefValue

Prototype

```
void vmirtSetViewObjectRefValue(
    vmiViewObjectP   object,
    vmiViewValueType type,
    void             *pValue
);
```

Description

This function assigns a *referenced value* to the view object, given by the `pValue` argument. Every time the view object value is queried, the given pointer is dereferenced, so it *must remain valid for the duration of the simulation*. The `type` argument, indicates the value type, using one of the highlighted members of the `vmiViewValueType` enumeration:

```
typedef enum vmiViewValueTypeE {

    VMI_VVT_NOSPACE   = -1, // No value returned. More buffer required.
    VMI_VVT_ERROR     = 0,  // No value returned

    VMI_VVT_BOOL      = 1,  // Primitive types
    VMI_VVT_SCHAR     = 2,
    VMI_VVT_UCHAR     = 3,
    VMI_VVT_INT8      = 4,
    VMI_VVT_UNS8      = 5,
    VMI_VVT_INT16     = 6,
    VMI_VVT_UNS16     = 7,
    VMI_VVT_INT32     = 8,
    VMI_VVT_UNS32     = 9,
    VMI_VVT_INT64     = 10,
    VMI_VVT_UNS64     = 11,
    VMI_VVT_ADDR      = 12,
    VMI_VVT_FLT64     = 13,

    VMI_VVT_STRING    = 14,  // Single line string
    VMI_VVT_MULTILINE = 15,  // Multi line string

} vmiViewValueType;
```

Example

The OVP MIPS processor model uses this function to expose information about TLB accesses:

```
static void createTLBEventView(mipsP vpe, mipsUnsArch tlbSize) {

    Uns32 i;

    // create programmer's view for TLB
    vmiViewObjectP processorObject = vmirtGetProcessorViewObject(
        (vmiProcessorP)vpe
    );
    vpe->tlbView = vmirtAddViewObject(processorObject, "tlb", NULL);
    vpe->tlbMissEvent = vmirtAddViewEvent(vpe->tlbView, "miss", NULL);
    vpe->tlbWriteEvents[MIPS_WET_INDEX] = vmirtAddViewEvent(
        vpe->tlbView, "index", NULL
    );
    vpe->tlbWriteEvents[MIPS_WET_RANDOM] = vmirtAddViewEvent(
        vpe->tlbView, "random", NULL
    );
    vmiViewObjectP missObject = vmirtAddViewObject(vpe->tlbView, "va", NULL);
    vmirtSetViewObjectRefValue(missObject, VMI_VVT_UNS32, &vpe->tlbMissEventVA);

    vmiViewObjectP indexObject = vmirtAddViewObject(vpe->tlbView, "tlbindex", NULL);
    vmirtSetViewObjectRefValue(indexObject, VMI_VVT_UNS32, &vpe->tlbEventIndex);

    vmiViewObjectP oldVPNObject = vmirtAddViewObject(
        vpe->tlbView, "entryoldvpn", NULL
    );
    vmirtSetViewObjectRefValue(oldVPNObject, VMI_VVT_UNS32, &vpe->tlbEventOldVPN);

    vmiViewObjectP newVPNObject = vmirtAddViewObject(
        vpe->tlbView, "entrynewvpn", NULL
    );
    vmirtSetViewObjectRefValue(newVPNObject, VMI_VVT_UNS32, &vpe->tlbEventNewVPN);

    . . . lines omitted for clarity . . .
}
```

Notes and Restrictions

1. The saved `pValue` on a view object can be accessed and modified using functions `vmirtGetViewObjectUserData` and `vmirtSetViewObjectUserData`.

QuantumLeap Semantics

Synchronizing

24.5 vmirtSetViewObjectValueCallback

Prototype

```
void vmirtSetViewObjectValueCallback(
    vmiViewObjectP object,
    vmiViewValueFn valueCB,
    void          *userData
);
```

Description

This function defines a function to return the value of a view object. Every time the view object value is queried, the given function is called, and must fill a passed buffer with the object value. The callback function, of type `vmiViewValueFn`, must be declared using the `VMI_VIEW_VALUE_FN` macro:

```
#define VMI_VIEW_VALUE_FN(_NAME) vmiViewValueType _NAME( \
    vmiViewObjectP object,      \
    void          *clientData,  \
    void          *buffer,      \
    Uns32         *bufferSize   \
)
typedef VMI_VIEW_VALUE_FN((*vmiViewValueFn));
```

The arguments to this function are as follows:

1. **`object`**: the view object passed as the `object` argument to `vmirtSetViewObjectValueCallback`.
2. **`clientData`**: the value passed as `userData` to `vmirtSetViewObjectValueCallback`.
3. **`buffer`**: a buffer that must be filled with the view object result value.
4. **`bufferSize`**: the size of the buffer, passed as a by-ref argument. If the buffer is too small to hold the result, the callback should update `*bufferSize` to be the required size and return `VMI_VVT_NOSPACE`.

The function returns a value of type `vmiViewValueType`:

```
typedef enum vmiViewValueTypeE {

    VMI_VVT_NOSPACE   = -1, // No value returned. More buffer required.
    VMI_VVT_ERROR     = 0,  // No value returned

    VMI_VVT_BOOL      = 1,  // Primitive types
    VMI_VVT_SCHAR     = 2,
    VMI_VVT_UCHAR     = 3,
    VMI_VVT_INT8      = 4,
    VMI_VVT_UNS8      = 5,
    VMI_VVT_INT16     = 6,
    VMI_VVT_UNS16     = 7,
    VMI_VVT_INT32     = 8,
    VMI_VVT_UNS32     = 9,
    VMI_VVT_INT64     = 10,
    VMI_VVT_UNS64     = 11,
    VMI_VVT_ADDR      = 12,
    VMI_VVT_FLT64     = 13,

    VMI_VVT_STRING    = 14,  // Single line string
    VMI_VVT_MULTILINE = 15,  // Multi line string
} vmiViewValueType;
```

A result of `VMI_VVT_NOSPACE` indicates that the result buffer was too small to hold the required result (in which case `*bufferSize` should be set to the required size). A result of `VMI_VVT_ERROR` indicates that a result could not be returned for some unspecified reason. All other results indicate a value was successfully returned.

Example

This example shows a modification to the OVP MIPS processor model allowing view events to be used to return COP0 system register values:

```
typedef struct COP0RegViewInfoS {
    mipsP tc;
    Uns32 cs;
    Uns32 sel;
} *COP0RegViewInfoP;

static VMI_VIEW_VALUE_FN(getCOP0RegViewValue) {

    COP0RegViewInfoP info = (COP0RegViewInfoP)clientData;

    if(*bufferSize < sizeof(mipsUnsArch)) {

        *bufferSize = sizeof(mipsUnsArch);
        return VMI_VVT_NOSPACE;

    } else {

        mipsUnsArch result = doMFC0(0, info->tc, info->cs, info->sel);
        *(mipsUnsArchP)buffer = result;

        return (sizeof(result)==32) ? VMI_VVT_UNS32 : VMI_VVT_UNS64;
    }
}

void mipsAddCOP0RegisterView(mipsP tc) {

    Uns32 cs, sel;

    vmiViewObjectP processorObject = vmirtGetProcessorViewObject((vmiProcessorP)tc);
    vmiViewObjectP baseObject      = vmirtAddViewObject(processorObject, "COP0", 0);

    for(cs=0; cs<MIPS_COP_REGS; cs++) {

        for(sel=0; sel<MIPS_COP_SEL; sel++) {

            mipsCop0RegId id   = mapCOP0Index(tc, cs, sel);
            const char   *name = mipsMapCOP0RegName(cs, sel, False);

            char description[32];
            sprintf(description, "COP0(%u,%u)", cs, sel);

            vmiViewObjectP regObject = vmirtAddViewObject(
                baseObject, name, description
            );

            // create a dynamic view.
            COP0RegViewInfoP info = calloc(1, sizeof(*info));
            if(info) {

                info->tc  = tc;
                info->cs  = cs;
                info->sel = sel;

                vmirtSetViewObjectValueCallback(
                    regObject, getCOP0RegViewValue, info
                );
            }
        }
    }
}
```

Notes and Restrictions

1. The saved `userData` on a view object can be accessed and modified using functions `vmirtGetViewObjectUserData` and `vmirtSetViewObjectUserData`.

QuantumLeap Semantics

Synchronizing

24.6 vmirtGetViewObjectUserData

Prototype

```
void *vmirtGetViewObjectUserData(vmiViewObjectP object);
```

Description

This function returns to the generic `userData` pointer for a view object.

For view objects configured using `vmirtSetViewObjectValueCallback`, this returns the `userData` specified with that call.

For view objects configured using `vmirtSetViewObjectRefValue`, this returns the `pValue` specified with that call.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.7 vmirtSetViewObjectUserData

Prototype

```
void vmirtSetViewObjectUserData(vmiViewObjectP object, void *userData);
```

Description

This function assigns to the generic `userData` pointer for a view object.

For view objects configured using `vmirtSetViewObjectValueCallback`, this replaces the `userData` specified with that call.

For view objects configured using `vmirtSetViewObjectRefValue`, this replaces the `pValue` specified with that call.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.8 vmirtAddViewAction

Prototype

```
void vmirtAddViewAction(
    vmiViewObjectP  object,
    const char     *name,
    const char     *description,
    vmiViewActionFn actionCB,
    void           *userData
);
```

Description

This function adds an action callback to a view object. Invocation of these callbacks is not currently supported, so this function should not be used.

Example

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

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.9 vmirtAddViewEvent

Prototype

```
vmiViewEventP vmirtAddViewEvent(
    vmiViewObjectP object,
    const char    *name,
    const char    *description
);
```

Description

This function adds an *event* to a view object. Clients can register listener functions on such events so that they are notified when the event triggers. Models can trigger view events using function `vmirtTriggerViewEvent` . The description argument is used to describe the view event (and can be `NULL`).

Example

The OVP MIPS processor model uses this function to convey information about TLB accesses:

```
static void createTLBEventView(mipsP vpe, mipsUnsArch tlbSize) {

    Uns32 i;

    // create programmer's view for TLB
    vmiViewObjectP processorObject = vmirtGetProcessorViewObject(
        (vmiProcessorP)vpe
    );
    vpe->tlbView = vmirtAddViewObject(processorObject, "tlb", NULL);
    vpe->tlbMissEvent = vmirtAddViewEvent(vpe->tlbView, "miss", NULL);
    vpe->tlbWriteEvents[MIPS_WET_INDEX] = vmirtAddViewEvent(
        vpe->tlbView, "index", NULL
    );
    vpe->tlbWriteEvents[MIPS_WET_RANDOM] = vmirtAddViewEvent(
        vpe->tlbView, "random", NULL
    );

    . . . lines omitted for clarity . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.10 vmirtNextViewEvent

Prototype

```
vmiViewEventP vmirtNextViewEvent(
    vmiViewObjectP object,
    vmiViewEventP  old
);
```

Description

This function iterates through events on a view object. `old` should be set to `NULL` for the first call, then the returned value used for each subsequent call until `NULL` is returned. This function is typically used by intercept libraries.

Example

This example is extracted from a helper intercept library that monitors *exception* and *mode switch* events on a processor-:

```
static void addEvents(vmiosObjectP object, vmiViewObjectP procObject) {

    vmiViewEventP viewEvt;

    for (viewEvt = vmirtNextViewEvent(procObject, NULL);
         viewEvt != NULL;
         viewEvt = vmirtNextViewEvent(procObject, viewEvt)
    ) {
        const char *evtName = vmiviewGetViewEventName(viewEvt);

        if (strcmp(evtName, "exception") == 0) {

            vmiviewAddViewEventListener(viewEvt, arcExceptionEventTrigger, object);
            if (DIAG_DBG) vmiPrintf("exception event added\n");

        } else if (strcmp(evtName, "modeswitch") == 0) {

            vmiviewAddViewEventListener(viewEvt, arcModeswitchEventTrigger, object);
            if (DIAG_DBG) vmiPrintf("modeswitch event added\n");

        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.11 vmirtTriggerViewEvent

Prototype

```
void vmirtTriggerViewEvent(vmiViewEventP event);
```

Description

This function triggers a previously-created view event.

Example

The OVP ARM processor model uses this function to notify clients when the MMU is enabled:

```
static void updateMMUEnable(armP arm) {

    // switch to MMU-enabled mode
    armSwitchMode(arm);

    // trigger the programmer's view MMU enable event
    vmirtTriggerViewEvent(arm->mmuEnableEvent);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

24.12 vmirtDeleteViewObject

Prototype

```
void vmirtDeleteViewObject(vmiViewObjectP object);
```

Description

This function deletes a previously-created view object.

Example

The OVP MIPS processor model uses this function when TLB data structures are being deleted:

```
void mipsFreeTLB(mipsP vpe) {

    // free TLB structure
    tlbEntryP tlb = vpe->tlb - TLB_PSEUDO(vpe);
    STYPE_FREE(tlb);
    vpe->tlb = 0;

    // free FTLB random ways
    if(vpe->tlbFTLBRandomWays) {
        STYPE_FREE(vpe->tlbFTLBRandomWays);
    }

    // delete the programmer's views
    vmirtDeleteViewObject(vpe->tlbView);
    vpe->tlbView                        = 0;
    vpe->tlbMissEvent                   = 0;
    vpe->tlbWriteEvents[MIPS_WET_INDEX]  = 0;
    vpe->tlbWriteEvents[MIPS_WET_RANDOM] = 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

25 Runtime Commands

A runtime command is part of a model or plugin which can be executed by the simulator. Its typical use is to change the mode of operation of the model or to print information from inside it.

Creating a command installs a C callback function in the model that will be called when the command is executed. A command can be given arguments, which are either passed to the callback function exactly as given, or parsed inside the simulator by a standard parser and the resulting converted values passed to the callback function. The latter strategy is preferable because it results in standardized argument handling, allows the simulator to generate help messages and lets other tools use the command in a standard way.

To receive raw arguments exactly as supplied to the command, use `vmirtAddCommand` to define the command.

To receive arguments that have been validated and converted from strings to the desired types, use `vmirtAddCommandParse` to define the command, and use `vmirtAddArg` to define the argument names, types and usage.

Commands can be called in several ways:

- From the platform, using OP
    - See the function `opCommandCall` in the *Advanced Simulation Control of Platforms and Modules User Guide*.
- From a simulator control file or simulator command line, using `–callcommand`:
    - See the *Simulation Control of Platforms and Modules User Guide* for more information.
- From the Imperas multicore debugger. Commands appear in the TCL interpreter.
- From a graphical interface.

Arguments to `vmirtAddCommand` and `vmirtAddCommandParse` control how the command will appear in a graphical environment (if at all).

25.1 vmirtAddCommand

Prototype

```
void vmirtAddCommand(
    vmiProcessorP    processor,
    const char      *name,
    const char      *help,
    vmirtCommandFn   commandCB,
    vmiCommandAttrs  attrs
);
```

Description

Add a command to the given processor. The arguments are as follows:

`processor`	The current processor.
`name`	The name used to call the command.
`help`	Describe the arguments required by the command.
`commandCB`	The function to be called when the command is executed.
`attrs`	Graphical interface categorization (see section 25.6).

When the command is called, the function `commandCB` will be called in the model. It is passed the processor context, the number of arguments supplied to the command (`argc`) and arguments as an array of strings called `argv[]`. `argv[0]` is the command name.

Example

```
static VMIRT_COMMAND_FN(myCommand) {
    vmiPrintf("command %s was called, with args:\n", argv[0]);
    int i;
    for(i= 1; i < argc) {
        vmiPrintf("   arg %d=%s\n", i, argv[i]);
    }
}

VMI_CONSTRUCTOR_FN(constructor) {
...
    vmirtAddCommand(
        processor,
        "myCommand",
        "<filename> <duration> [option]",
        myCommand,
        VMI_CT_DEFAULT
    );
}
```

Notes and Restrictions

Strings passed to the callback will not persist after the callback is complete. Commands created using this function can be made visible in a graphical environment, but their arguments and usage will not be visible.

QuantumLeap Semantics

Synchronizing

25.2 vmirtAddCommandParse

Prototype

```
vmiCommandP vmirtAddCommandParse(
    vmiProcessorP       processor,
    const char         *name,
    const char         *help,
    vmirtCommandParseFn commandCB,
    vmiCommandAttrs     attrs
);
```

Description

Add a command to the given processor. The arguments are as follows:

`processor`	The current processor.
`name`	The name used to call the command.
`help`	Describe the arguments required by the command.
`commandCB`	The function to be called when the command is executed.
`attrs`	Graphical interface categorization (see section 25.6).

When the command is called, the arguments will be parsed according to the argument specifications provided by `vmirtAddArg`, then the function `commandCB` will be called in the model. The called function is passed the processor, the number of arguments specified with `vmirtAddArg` and an array of argument value structures filled with the parsed values, in the order they were originally specified. Type `vmirtCommandParseFn` is defined as:

```
#define VMIRT_COMMAND_PARSE_FN(_NAME) const char *_NAME( \
    vmiProcessorP processor,    \
    Int32         argc,         \
    vmiArgValue   argv[]        \
)
typedef VMIRT_COMMAND_PARSE_FN((*vmirtCommandParseFn));
```

The arguments values are defined by structures of type `vmiArgValue`:

```
typedef struct argValueS {
    const char   *name;
    vmiArgType    type;
    Bool          isSet;
    Bool          changed;
    argValueUnion u;
} vmiArgValue, *vmiArgValueP;
```

The argument value structure array contains the following information:

Field name	**Contains**
`name`	argument name (without the -)
`type`	argument type
`isSet`	true if the parser found this argument in the call
`changed`	true if the argument has a static value which was modified by this call.
`u`	a union of possible value types. Must be accessed as the right type.

If the arguments to the command have persistent values which are of interest to the user, `vmirtAddArg` can be used to inform the simulator how to retrieve them (see `vmirtAddArg`).

Example

```
// This is the place that stores permanent values of the flags (if required)
typedef struct staticValuesS {
    Addr   arg1Value;
    Bool   arg2Value;
    Flt64  arg3Value;
    Int32  arg4Value;
    Char  *arg5Value;
} staticValues;

staticValues *sv = malloc(sizeof(staticValues));

Bool myStaticBoolValue;

//
// Called by simulator to retrieve the value of this argument
// (for display by a GUI for example)
//
static VMIRT_ARG_VALUE_FN(arg6ValueFn) {
    *valuePtr = myStaticBoolValue;
}

static VMIRT_COMMAND_PARSE_FN(myCommand) {
    vmiPrintf("myCommand was called, with parsed values:\n");
    int i;
    for(i= 1; i < argc) {
        vmiPrintf("   arg %d is %s\n", i, arg[i].name);
        switch() {
            case VMI_CA_ADDRESS:
                 vmiPrintf("   address = 0x%llx\n", arg[i].u.addr);
                 break;
            case VMI_CA_FLAG:
                 vmiPrintf("   bool    = %s\n", arg[i].u.flag ? “True” : “False”);
                 break;
            case VMI_CA_FLOAT:
                 vmiPrintf("   float   = %f\n", arg[i].u.flt);
                 break;
            case VMI_CA_INTEGER:
                 vmiPrintf("   integer = %d\n", arg[i].u.integer);
                 break;
            case VMI_CA_STRING:
                 vmiPrintf("   string = %s\n", arg[i].u.string);
                 break;
            default:
                 break;
       }
       vmiPrintf(“       %s\n”,  argv[i].isSet     ? “Set”     : “Unset”     );
       vmiPrintf(“       %s\n\n”,  argv[i].changed ? “Changed” : “Unchanged” );
    }
}

VMI_CONSTRUCTOR_FN(constructor) {
...
vmiCommandP cmd = vmirtAddCommandParse(
    processor,
    "myCommand",
    "Do the operation ‘myCommand’",
    myCommand,
    VMI_CT_DEFAULT
);

vmirtAddArg(
    cmd,                    // command handle
    “arg1”,                  // name of argument
    “arg1 is the address”,  // help
    VMI_CA_ADDRESS,          // type
    VMI_CAA_MENU,            // will appear in a GUI
    False,                   // mandatory
    &sv->arg1Value
);

vmirtAddArg(
    cmd,                     // command handle
    “arg2”,                  // name of argument
    “arg2 is a boolean”,     // help
    VMI_CA_FLAG,             // type
    VMI_CAA_MENU | VMI_CAA_ENABLE,  // will appear in a GUI, turns the feature on
    False,                   // mandatory
    &sv->arg2Value
);

vmirtAddArg(
    cmd,                     // command handle
    “arg3”,                  // name of argument
    “arg3 is a float”,       // help
    VMI_CA_FLOAT,            // type
    VMI_CAA_MENU,            // will appear in a GUI
    False,                   // mandatory
    &sv->arg3Value
);

vmirtAddArg(
    cmd,                     // command handle
    “arg4”,                  // name of argument
    “arg4 is an integer”,    // help
    VMI_CA_INTEGER,          // type
    VMI_CAA_MENU,            // will appear in a GUI
    False,                   // mandatory
    &sv->arg4Value
);

vmirtAddArg(
    cmd,                     // command handle
    “arg5”,                  // name of argument
    “arg5 is a string”,      // help
    VMI_CA_STRING,           // type
    VMI_CAA_MENU,            // will appear in a GUI
    False,                   // mandatory
    &sv->arg5Value
);

vmirtAddArg(
    cmd,                     // command handle
    “arg6”,                  // name of argument
    “arg6 is a boolean”,     // help
    VMI_CA_INTEGER,          // type
    VMI_CAA_MENU
   |VMI_CAA_VALUE_CALLBACK, // will appear in a GUI. Can be retrieved by callback
    False,                   // mandatory
    arg6ValueFn              // function to be used
);
```

Example usage

```
-callcommand “procA/mycommand  –arg3 1.5 –arg2 Y –arg4 44 –arg5 maybe –arg6 –arg1
0xFF000000”

[output]
myCommand was called, with parsed values:
   arg[0] is arg1
    address = 0xFF000000
    Set
    Changed
   arg[1] is arg2
    boolean =  True
    Set
    Changed

   arg[2] is arg3
    float   =  1.500000
    Set
    Changed

   arg[3] is arg4
    integer =  44
    Set
    Changed

   arg[4] is arg5
    string   =  maybe
    Set
    Changed

   arg[5] is arg6
    boolean = True
    Set
    Changed
```

Notes and Restrictions

1. The array of argument values passed to the callback will not persist after the callback is complete.
2. When a command is called, its arguments need not be provided in the order they were specified.

QuantumLeap Semantics

Synchronizing

25.3 vmirtAddArg

Prototype

```
vmiArgP vmirtAddArg(
    vmiCommandP     command,
    const char     *name,
    const char     *help,
    vmiArgType      type,
    vmiArgAttrs     attrs,
    Bool            mandatory,
    void           *ptr
);
```

Description

Add an argument specification to the passed command. When the command is called the command parser will recognize an argument of this name (preceded by '-') and take the next item as its value, converting it according to type and putting it in the value array passed to the command callback.

`name`	The name of the argument to be recognized by the parser (preceded by '-')
`help`	Description of the argument.
`type`	The type of the argument, defined using type `vmiArgType`:

```
typedef enum vmiArgTypeE {
    VMI_CA_NONE  = 0, // do not use
    VMI_CA_BOOL     , //
    VMI_CA_INT32    , // signed 32-bit integer
    VMI_CA_INT64    , // signed 64-bit integer
    VMI_CA_UNS32    , // unsigned 32
    VMI_CA_UNS64    , // unsigned 64
    VMI_CA_DOUBLE   , // C float
    VMI_CA_STRING   , // string (see vmirtAddArg)
    VMI_CA_ENUM     , // types are defined using vmirtAddArgEnum
    VMI_CA_ENDIAN   , // types are defined using vmirtAddArgEnum
    VMI_CA_PTR      , // types are defined using vmirtAddArgEnum
} vmiArgType;
```

Name	**Meaning**
VMI_CA_BOOL	Boolean
VMI_CA_INT32	32-bit signed integer
VMI_CA_INT64	64-bit signed integer
VMI_CA_UNS32	32-bit unsigned integer
VMI_CA_UNS64	64-bit unsigned integer
VMI_CA_DOUBLE	Double-precision floating point
VMI_CA_STRING	Null-terminated C string.
VMI_CA_ENUM	Enumeration (members defined by `vmirtAddArgEnum`)
VMI_CA_ENDIAN	Endian (one of `big`, `little` or `either`)
VMI_CA_PTR	Generic pointer

`attrs` A bit-mask used to modify the behavior of the command when used with a graphical user interface. The 32-bit value should be constructed with the following macros from `vmiCommand.h`:

Name	**Meaning**

VMI_CAA_DEFAULT	Not used in a graphical interface
VMI_CAA_MENU	Is used in a graphical interface
VMI_CAA_ENABLE	This argument is used to enable the feature controlled by the command
VMI_CAA_VALUE_CALLBACK	The data pointer refers to a callback function rather than to the data.

`mandatory`   Whether the argument is mandatory or optional.
`ptr`   Refers to the location in the model that permanently stores the argument value, or to a function that can be called to retrieve the value, depending on the value of the `attrs` bit defined by `VMI_CAA_VALUE_CALLBACK`. If `ptr` is set to null then a graphical interface will be unable to display the current value of this argument.

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

25.4 vmirtAddArgEnum

Prototype

```
void vmirtAddArgEnum(
    vmiArgP          argument,
    const char      *name,
    const char      *help,
    Uns32            value
);
```

Description

Add an enumerated member `name`, with the specified `value`, to the previously-created argument of type `VMI_CA_ENUM`. A value of zero as interpreted as *unspecified*; in this case, a value is derived by adding 1 to the value of the previously-created member.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

25.5 vmirtFindArgValue

Prototype

```
vmiArgValueP vmirtFindArgValue(
    Uns32       argc,
    vmiArgValue argv[],
    const char *name
)
```

Description

This function returns a pointer to the argument value structure for the named argument in the list of argument values presented to a command. If there is no argument with the given name, the function returns `NULL`.

Example

The OVP MIPS model uses this function when handling arguments to the `mipsWriteTLBEntry` command:

```
static VMIRT_COMMAND_PARSE_FN(writeTLBEntryCmd) {

    vmiArgValueP argIndex = vmirtFindArgValue(argc, argv, "index");
    vmiArgValueP argLo0   = vmirtFindArgValue(argc, argv, "lo0");
    vmiArgValueP argLo1   = vmirtFindArgValue(argc, argv, "lo1");
    vmiArgValueP argHi0   = vmirtFindArgValue(argc, argv, "hi0");
    vmiArgValueP argMask  = vmirtFindArgValue(argc, argv, "mask");

    if(!argIndex->isSet ||
       !argLo0->isSet || !argLo1->isSet ||
       !argHi0->isSet || !argMask->isSet) {

        return "Argument Error, Must set all arguments";
    }

    mipsP tc    = (mipsP)processor;
    mipsP vpe   = GET_VPE(tc);
    Uns32 index = argIndex->u.uns64;

    mipsTLBRegs cxt = {
        EntryLo0 : { .uArch = argLo0->u.uns64},
        EntryLo1 : { .uArch = argLo1->u.uns64},
        EntryHi  : { .uArch = argHi0->u.uns64},
        PageMask : { .uArch = argMask->u.uns64}
    };

    if(mipsWriteTLBEntryRegs(vpe, index, &cxt)) {
        return "1";
    } else {
        return "0";
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

25.6 Command attributes

The last argument `to` `vmirtAddCommand` and `vmirtAddCommandParse` is a bitmask used to modify the behavior of the command when used with a graphical user interface. The 32-bit value should be constructed with the following macros from `vmiCommand.h:`

macro	**meaning**
VMI_CT_DEFAULT	Do not use this command in graphical interfaces
VMI_CT_MASK	mask for the command type:
VMI_CT_QUERY	command used to query the model, causing an immediate response.
VMI_CT_STATUS	command used to query the model. Should be called each time the simulator stops.
VMI_CT_MODE	command changes the mode of the model
VMI_CO_MASK	Command object type mask. Puts the command in a category depending on the object the command operates on.
VMI_CO_CACHE	Cache models
VMI_CO_CPU	Processor models
VMI_CO_CONTEXT	Application code stack trace
VMI_CO_DIAG	Model diagnostics
VMI_CO_FUNCTION	Application source code functions
VMI_CO_GIC	Interrupt controller
VMI_CO_LINE	Application code source lines
VMI_CO_LKM	Loadable kernel modules
VMI_CO_MEMORY	Memory
VMI_CO_OS	Operating system, RTOS or scheduler
VMI_CO_TASK	Operating system tasks or processes
VMI_CO_TLB	Translation buffer
VMI_CO_SYMBOL	Application code source symbols
VMI_CA_MASK	Command action type mask. Puts the command in a category according to its action.
VMI_CA_CONTROL	Command controls how the model operates (e.g. on or off)
VMI_CA_COVER	Command associated with code coverage
VMI_CA_PROFILE	Command associated with code profiling
VMI_CA_QUERY	Command to query the model
VMI_CA_REPORT	Command to print a report
VMI_CA_TRACE	Command to control tracing
VMI_CT_HELP	If set, the simulator will add the –help argument and supply its response

26 Processor Register, Mode, Exception and Port Access Utilities

A set of functions is available that allows processor registers to be read and written using the register interface presented using `vmiRegInfo` objects. Similarly, functions are available allowing processor mode and exception state to be queried using the interface presented using `vmiModeInfo` and `vmiExceptionInfo` objects. It is also possible to query the port connections of a processor using the interface presented by `vmiBusPort`, `vmiNetPort` and `vmiFifoPort` objects.

This interface is typically used in intercept libraries to query and update processor state without requiring access to processor private data structures.

26.1 vmirtGetRegGroupByName

Prototype

```
vmiRegGroupCP vmirtGetRegGroupByName(
    vmiProcessorP processor,
    const char   *name
);
```

Description

This function returns a pointer to a `vmiRegGroupCP` structure implementing the named register group. The returned pointer can subsequently be used with `vmirtGetNextRegInGroup`, allowing iteration of all registers in a group.

If a register group with the specified name is not found, a null pointer is returned.

Example

The OVP MIPS CPU Helper intercept library uses this function to obtain `vmiRegInfoCP` handles to all implemented COP0 registers:

```
static VMIOS_CONSTRUCTOR_FN(mipsCpuHelperConstructor) {

    . . . lines omitted for clarity . . .

    vmiRegGroupCP cop0Group = vmirtGetRegGroupByName(processor, "COP0");

    if(cop0Group) {

        vmiRegInfoCP cop0Reg = 0;

        while((cop0Reg=vmirtGetNextRegInGroup(processor, cop0Group, cop0Reg))) {

            Uns32 reg, sel;

            if(sscanf(cop0Reg->description, "CP0 register %u/%u", &reg, &sel)) {
                object->cop0Registers[reg][sel] = cop0Reg;
            }
        }
    }
}
```

Notes and Restrictions

1. Register groups can only be found for processors that define the `regGroupCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetRegGroupByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.2 vmirtGetNextRegGroup

Prototype

```
vmiRegGroupCP vmirtGetNextRegGroup(
    vmiProcessorP processor,
    vmiRegGroupCP previous
);
```

Description

Given a processor and a `vmiRegGroupCP` structure implementing a register group in that processor, this function returns the *next* defined register group for that processor. The returned pointer can subsequently be used with `vmirtGetNextRegInGroup`, allowing iteration of all registers in a group. The function returns `NULL` if there are no more register group definitions.

Typically, `vmirtGetNextRegGroup` will be used in the constructor of a semi-hosting library to initialize pointers used to access registers in that group.

Example

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

Notes and Restrictions

1. Registers groups can only be iterated for processors that define the `regGroupCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextRegGroup` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.3 vmirtGetRegByName

Prototype

```
vmiRegInfoCP vmirtGetRegByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiRegInfoCP` structure implementing the named register. The returned pointer can subsequently be used with `vmirtRegRead` and `vmirtRegWrite`. This allows the registers within a processor model to be accessed without requiring any knowledge of the internal structure of that processor model, other than the names of the registers.

If a register with the specified name is not found, a null pointer is returned.

Typically, `vmirtGetRegByName` will be used in the constructor of a semi-hosting library to initialize pointers used to access the registers used in the calling conventions of that processor. It is also useful when specifying correspondence between `vmiRegInfoCP` register structures and processor fields implementing those registers when defining processor instruction attributes.

Example

The OVP RISC-V processor model uses this function to specify that the `fpFlags` field in the processor structure is used to implement the `fflags` register:

```
#define RISCV_REG_IMPL_RAW(_REG, _FIELD, _BITS) \
    vmirtRegImplRaw(processor, _REG, _FIELD, _BITS)

VMI_REG_IMPL_FN(riscvRegImpl) {

    // specify that fpFlags are in fflags
    vmiRegInfoCP fflags = vmirtGetRegByName(processor, "fflags");
    RISCV_FIELD_IMPL_RAW(fflags, fpFlags);

    // exclude artifact registers
    RISCV_FIELD_IMPL_IGNORE(fpActiveRM);
}
```

Notes and Restrictions

1. In VMI versions prior to 5.0.0, this function was called `vmiosGetRegDesc` and prototyped in file `vmiOSLib.h`. It has been moved to `vmiRt.h` because it is of more general applicability than semihosting libraries alone.

QuantumLeap Semantics

Non-self-synchronizing

26.4 vmirtGetNextReg

Prototype

```
vmiRegInfoCP vmirtGetNextReg(
    vmiProcessorP  processor,
    vmiRegInfoCP   previous
);
```

Description

Given a processor and a `vmiRegInfoCP` structure implementing a register in that processor, this function returns the *next* defined register for that processor. The returned pointer can subsequently be used with `vmirtRegRead` and `vmirtRegWrite`. If `NULL` is passed as the `previous` argument, the *first* register definition in the processor is returned. The function returns `NULL` if there are no more register definitions.

Typically, `vmirtGetNextReg` will be used in the constructor of a semi-hosting library to initialize pointers used to access those registers.

Example

The OVP ARMM CPU Helper intercept library uses this function to install memory watch callbacks on memory-mapped SCS registers:

```
static void addSCSRegs(vmiosObjectP object) {

    Uns32          numRegs = countSCSRegs(object);
    armmEventViewP events  = object->armmEventViewObjects;
    scsRegDataP    regData = initRegData(events, numRegs);
    memDomainP     domainP = vmirtGetProcessorDataDomain(object->processor);
    vmiRegInfoCP   reg     = NULL;

    while((reg=vmirtGetNextReg(object->processor, reg))) {

        if(isSCSReg(reg)) {

            Uns32 address = getSCSRegAddress(reg);

            regData->object = object;
            regData->reg    = reg;

            vmirtAddReadCallback(
                domainP, 0, address, address+3, armmSCSRead,  regData
            );
            vmirtAddWriteCallback(
                domainP, 0, address, address+3, armmSCSWrite, regData
            );

            regData++;
        }
    }
}
```

Notes and Restrictions

1. Registers can only be iterated for processors that define the `regInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextReg` returns `NULL`.

QuantumLeap Semantics
Non-self-synchronizing

26.5 vmirtGetNextRegInGroup

Prototype

```
vmiRegInfoCP vmirtGetNextRegInGroup(
    vmiProcessorP processor,
    vmiRegGroupCP group,
    vmiRegInfoCP  previous
);
```

Description

Given a processor and a `vmiRegInfoCP` structure implementing a register in that processor, this function returns the *next* defined register for that processor in the given group. The returned pointer can subsequently be used with `vmirtRegRead` and `vmirtRegWrite`. If `NULL` is passed as the `previous` argument, the *first* register definition in the processor group is returned. The function returns `NULL` if there are no more register definitions.

Typically, `vmirtGetNextRegInGroup` will be used in the constructor of a semi-hosting library to initialize pointers used to access those registers.

Example

The OVP MIPS CPU Helper intercept library uses this function to obtain `vmiRegInfoCP` handles to all implemented COP0 registers:

```
static VMIOS_CONSTRUCTOR_FN(mipsCpuHelperConstructor) {

    . . . lines omitted for clarity . . .

    vmiRegGroupCP cop0Group = vmirtGetRegGroupByName(processor, "COP0");

    if(cop0Group) {

        vmiRegInfoCP cop0Reg = 0;

        while((cop0Reg=vmirtGetNextRegInGroup(processor, cop0Group, cop0Reg))) {

            Uns32 reg, sel;

            if(sscanf(cop0Reg->description, "CP0 register %u/%u", &reg, &sel)) {
                object->cop0Registers[reg][sel] = cop0Reg;
            }
        }
    }
}
```

Notes and Restrictions

1. Registers can only be iterated for processors that define the `regInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextRegInGroup` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.6 vmirtRegRead

Prototype

```
Bool vmirtRegRead(
    vmiProcessorP processor,
    vmiRegInfoCP  regDesc,
    void         *result
);
```

Description

Given a processor and a register description, this function copies the current value of the register to the result buffer. This is typically used in an intercepted function call to obtain the value of a function argument using the standard processor ABI.

Example

The OVP ARMM CPU Helper intercept library uses this function to obtain the processor flags from the `cpsr` register:

```
#define WIDTH(_W, _ARG)         ((_ARG) & ((1<<(_W))-1))
#define GETFIELD(_I, _W, _P)    (WIDTH(_W, (_I >> _P)))

static Uns32 condFlags(vmiosObjectP object) {

    Uns32 cpsr;

    vmiosRegRead(object->processor, object->cpsr, &cpsr);

    return GETFIELD(cpsr, 4, 28);
}
```

Notes and Restrictions

1. It is the caller’s responsibility to ensure that the buffer is large enough to hold the register value.
2. In VMI versions prior to 5.0.0, this function was called `vmiosRegRead` and prototyped in file `vmiOSLib.h`. It has been moved to `vmiRt.h` because it is of more general applicability than semihosting libraries alone.

QuantumLeap Semantics

Non-self-synchronizing

26.7 vmirtRegWrite

Prototype

```
Bool vmirtRegWrite(
    vmiProcessorP processor,
    vmiRegInfoCP  regDesc,
    const void    *value
);
```

Description

Given a processor and a register description, this function sets the current value of the register using the passed value. This is typically used in an intercepted function call to return a function result using the standard processor ABI.

Example

This example shows how an intercept library can use this function to return a value for an intercepted `time` system call:

```
static VMIOS_INTERCEPT_FN(timeCB) {

    Uns32 tvAddr;

    // obtain function arguments
    getArg(processor, object, 0, &tvAddr);

    // implement gettimeofday
    vmiosTimeBuf timeBuf = {0};
    vmiosGetTimeOfDay(processor, &timeBuf);

    // write back results
    transcribeSeconds(processor, tvAddr, timeBuf.sec);
    vmiosRegWrite(processor, object->result, &timeBuf.sec);
}
```

Notes and Restrictions

1. It is the caller's responsibility to ensure that the buffer is large enough to hold the register value.
2. In VMI versions prior to 5.0.0, this function was called `vmiosRegWrite` and prototyped in file `vmiOSLib.h`. It has been moved to `vmiRt.h` because it is of more general applicability than semihosting libraries alone.

QuantumLeap Semantics

Non-self-synchronizing

26.8 vmirtGetCurrentMode

Prototype

```
vmiModeInfoCP vmirtGetCurrentMode(vmiProcessorP processor);
```

Description

Given a processor, this function returns the *current* defined mode for that processor. The mode is described using a structure defined in `vmiDbg.h` as:

```
typedef struct vmiModeInfoS {
    const char *name;          // mode name
    Uns32       code;          // model-specific mode code
    const char *description;   // description string
} vmiModeInfo;
```

This function is typically used in intercept libraries.

Example

The OVP ARC CPU Helper intercept library uses this function to obtain the processor mode in a mode-switch event callback:

```
static VMI_VIEW_EVENT_LISTENER_FN(arcModeswitchEventTrigger) {

    vmiosObjectP  object = (vmiosObjectP) userData;
    vmiModeInfoCP mode   = vmirtGetCurrentMode(object->processor);

    // Get current mode of processor
    arcModeInfoCodeE newMode  = mode->code;
    arcModeInfoCodeE lastMode = object->eventData->lastMode;
    cpuGetProcMode   procMode;

    if (lastMode == newMode) {
        return;    // No change to mode so ignore
    }

    // Are we tracing mode switches?
    if (object->traceFlags & TRACE_MODE) {
        vmiPrintf ("TRC (MODE) "FMT_64u": '%s': mode switched to %s\n",
                   vmirtGetICount(object->processor), object->name,
                   mode->name);
    }

    . . . lines omitted for clarity  . . .

}
```

Notes and Restrictions

1. Current mode will only be reported for processors that define the `getModeCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetCurrentMode` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.9 *vmirtGetNextMode*

Prototype

```
vmiModeInfoCP vmirtGetNextMode(
    vmiProcessorP processor,
    vmiModeInfoCP previous
);
```

Description

Given a processor and a `vmiModeInfoCP` structure implementing a mode in that processor, this function returns the *next* defined mode for that processor. The function returns `NULL` if there are no more mode definitions.

Typically, `vmirtGetMode` will be used in the constructor of a semi-hosting library to initialize pointers used to access those modes.

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiModeInfoCP mode = NULL;
    Uns32         i    = 0;

    // count all defined modes
    while((mode=vmirtGetNextMode(processor, mode))) {
        object->modeNum++;
    }

    // allocate array for mode handles
    object->modes = STYPE_CALLOC_N(vmiModeInfoCP, object->modeNum);

    // fill array of modes for later use
    while((mode=vmirtGetNextMode(processor, mode))) {
        object->modes[i++] = mode;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Modes can only be iterated for processors that define the `modeInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextMode` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.10 vmirtGetCurrentException

Prototype

```
vmiExceptionInfoCP vmirtGetCurrentException(vmiProcessorP processor);
```

Description

Given a processor, this function returns the *current* defined exception for that processor. The exception is described using a structure defined in `vmiDbg.h` as:

```
typedef struct vmiExceptionInfoS {
    const char *name;          // exception name
    Uns32       code;          // model-specific exception code
    const char *description;   // description string
} vmiExceptionInfo;
```

This function is typically used in intercept libraries.

Example

The OVP ARC CPU Helper intercept library uses this function to obtain the exception type in an exception event callback:

```
static VMI_VIEW_EVENT_LISTENER_FN(arcExceptionEventTrigger) {

    vmiosObjectP       object     = (vmiosObjectP) userData;
    vmiProcessorP      processor  = object->processor;
    vmiExceptionInfoCP exInfo     = vmirtGetCurrentException(processor);
    arcExcModesP       excType    = &arcExcModeTable[exInfo->code];
    Uns32              returnAddr = 0;

    // Build description of exception
    char descr[128];
    snprintf (descr, sizeof(descr), "%s exception", exInfo->name);

    if (excType->isInterrupt) {
        returnAddr = buildInterruptDescr(
            object, processor, excType, descr, sizeof(descr)
        );
    } else {
        returnAddr = buildExceptionDescr(
            object, processor, excType, descr, sizeof(descr)
        );
    }

    // See if we are tracing exceptions/interrupts
    if (
        (!excType->isInterrupt && (object->traceFlags & TRACE_EXCEPT))    ||
        ( excType->isInterrupt && (object->traceFlags & TRACE_INTERRUPT))
       ) {
        vmiPrintf (
            "TRC (EXCP) "FMT_64u": '%s': %s\n",
            vmirtGetICount(processor),
            object->name,
            descr
        );
    }

    . . . lines omitted for clarity . . .
}
```

Notes and Restrictions

1. Current exception will only be reported for processors that define the `getExceptionCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetCurrentException` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.11 vmirtGetNextException

Prototype

```
vmiExceptionInfoCP vmirtGetNextException(
    vmiProcessorP      processor,
    vmiExceptionInfoCP previous
);
```

Description

Given a processor and a `vmiExceptionInfoCP` structure implementing an exception in that processor, this function returns the *next* defined exception for that processor. The function returns `NULL` if there are no more exception definitions.

Typically, `vmirtGetException` will be used in the constructor of a semi-hosting library to initialize pointers used to access those exceptions.

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiExceptionInfoCP exception = NULL;
    Uns32              i         = 0;

    // count all defined exceptions
    while((exception=vmirtGetNextException(processor, exception))) {
        object->exceptionNum++;
    }

    // allocate array for mode handles
    object->exceptions = STYPE_CALLOC_N(vmiExceptionInfoCP, object-> exceptionNum);

    // fill array of exceptions for later use
    while((exception=vmirtGetNextException(processor, exception))) {
        object->exceptions[i++] = exception;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Exceptions can only be iterated for processors that define the `exceptionInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextException` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.12 vmirtGetBusPortByName

Prototype

```
vmiBusPortP vmirtGetBusPortByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiBusPort` structure implementing the named port. See section 4.1 for more information about bus port structures.

If a port with the specified name is not found, a `NULL` pointer is returned.

Typically, `vmirtGetBusPortByName` will be used in the constructor of a semi-hosting library to enable access to information associated with that port (for example, the memory domain).

Example

The OVP ARM CPU Helper intercept library uses this function to obtain bus ports exposing GICR and GICD register values externally:

```
static vmiBusPortP getMPCorePort(vmiProcessorP processor, const char *name) {

    vmiProcessorP parent;

    // locate cluster root
    while((parent=vmirtGetSMPParent(processor))) {
        processor = parent;
    }

    // look for the named GIC block port on the cluster root
    return vmirtGetBusPortByName(processor, name);
}
```

Notes and Restrictions

1. Bus ports can only be found for processors that define the `busPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetBusPortByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.13 vmirtGetNextBusPort

Prototype

```
vmiBusPortP vmirtGetNextBusPort(
    vmiProcessorP  processor,
    vmiBusPortP    previous
);
```

Description

Given a processor and a `vmiBusPortP` structure defining a bus port in that processor, this function returns the *next* defined bus port for that processor. The function returns `NULL` if there are no more bus port definitions. See section 4.1 for more information about bus port structures.

Typically, `vmirtGetNextBusPort` will be used in the constructor of a semi-hosting library to initialize pointers used to access those bus ports.

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiBusPortP port = NULL;
    Uns32       i    = 0;

    // count all defined bus ports
    while((port=vmirtGetNextBusPort(processor, port))) {
        object->portNum++;
    }

    // allocate array for port handles
    object->ports = STYPE_CALLOC_N(vmiBusPortP, object->portNum);

    // fill array of ports for later use
    while((port=vmirtGetNextBusPort(processor, port))) {
        object->ports[i++] = port;
    }

    . . . etc . . .
}
```

Notes and Restrictions

1. Bus ports can only be iterated for processors that define the `busPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextBusPort` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.14 vmirtGetNetPortByName

Prototype

```
vmiNetPortP vmirtGetNetPortByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiNetPort` structure implementing the named port. See section 4.3 for more information about net port structures.

If a port with the specified name is not found, a `NULL` pointer is returned.

Typically, `vmirtGetNetPortByName` will be used in the constructor of a semi-hosting library to enable access to information associated with that port (for example, the description).

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiNetPortP port = vmirtGetNetPortByName(processor, “int0”);

    if(port) {
        vmiPrintf(“found net port %s (%s)\n”, port->name, port->description);
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Net ports can only be found for processors that define the `netPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNetPortByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.15 vmirtGetNextNetPort

Prototype

```
vmiNetPortP vmirtGetNextBusPort(
    vmiProcessorP  processor,
    vmiNetPortP    previous
);
```

Description

Given a processor and a `vmiNetPortP` structure defining a net port in that processor, this function returns the *next* defined net port for that processor. The function returns `NULL` if there are no more net port definitions. See section 4.3 for more information about net port structures.

Typically, `vmirtGetNextNetPort` will be used in the constructor of a semi-hosting library to initialize pointers used to access those net ports.

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiNetPortP port = NULL;
    Uns32       i    = 0;

    // count all defined net ports
    while((port=vmirtGetNextNetPort(processor, port))) {
        object->portNum++;
    }

    // allocate array for port handles
    object->ports = STYPE_CALLOC_N(vmiNetPortP, object->portNum);

    // fill array of ports for later use
    while((port=vmirtGetNextNetPort(processor, port))) {
        object->ports[i++] = port;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Net ports can only be iterated for processors that define the `netPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextNetPort` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.16 vmirtGetFifoPortByName

Prototype

```
vmiFifoPortP vmirtGetFifoPortByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiFifoPort` structure implementing the named port. See section 4.4 for more information about FIFO port structures.

If a port with the specified name is not found, a `NULL` pointer is returned.

Typically, `vmirtGetFifoPortByName` will be used in the constructor of a semi-hosting library to enable access to information associated with that port (for example, the description).

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiFifoPortP port = vmirtGetFifoPortByName(processor, “fifoOut”);

    if(port) {
        vmiPrintf(“found FIFO port %s (%s)\n”, port->name, port->description);
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. FIFO ports can only be found for processors that define the `fifoPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetFifoPortByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

26.17 vmirtGetNextFifoPort

Prototype

```
vmiFifoPortP vmirtGetNextFifoPort(
    vmiProcessorP  processor,
    vmiFifoPortP   previous
);
```

Description

Given a processor and a `vmiFifoPortP` structure defining a FIFO port in that processor, this function returns the *next* defined FIFO port for that processor. The function returns `NULL` if there are no more FIFO port definitions. See section 4.4 for more information about FIFO port structures.

Typically, `vmirtGetNextFifoPort` will be used in the constructor of a semi-hosting library to initialize pointers used to access those FIFO ports.

Example

```
static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiFifoPortP port = NULL;
    Uns32        i    = 0;

    // count all defined FIFO ports
    while((port=vmirtGetNextFifoPort(processor, port))) {
        object->portNum++;
    }

    // allocate array for port handles
    object->ports = STYPE_CALLOC_N(vmiFifoPortP, object->portNum);

    // fill array of ports for later use
    while((port=vmirtGetNextFifoPort(processor, port))) {
        object->ports[i++] = port;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. FIFO ports can only be iterated for processors that define the `fifoPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextFifoPort` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

27 Documentation Support Functions

This section describes functions that are used to associate documentation with a processor or intercept library model. These functions are typically called once in the model constructor. They are defined in header file `vmiDoc.h`.

27.1 Defining the constructor

The `vmiIASAttrs` structure has an entry `docCB` which should be set to a function defined using the `VMI_DOC_FN` prototype. This function should use the following VMI functions to create documentation for the model.

The following examples are taken from the OVP ARC model.

```
// Define the documentation constructor

VMI_DOC_FN(arcDoc) {

    arcP        arc  = (arcP)processor;
    vmiDocNodeP root = vmidocAddSection(0, "Root");

    // emit architecture description
    vmiDocNodeP desc = vmidocAddSection(root, "Description");
    if(isARC600(arc)) {
        vmidocAddText(desc, "ARC 600 processor model (ARCv1 architecture)");
    } else if(isARC700(arc)) {
        vmidocAddText(desc, "ARC 700 processor model (ARCv1 architecture)");
    } else if(isARCv2(arc)) {
        vmidocAddText(desc, "ARCv2 architecture processor model");
    }

    . . . lines omitted for clarity . . .

    vmidocProcessor(processor, root);
}


const vmiIASAttr modelAttrs = {

    // version, model type, status etc are set here
    . . . omitted for clarity . . .

    // Set the documentation callback to the model’s
    // documentation constructor
    .docCB         = arcDoc,

    . . . lines omitted for clarity . . .
};
```

27.2 vmidocAddSection

Prototype

```
vmiDocNodeP vmidocAddSection(vmiDocNodeP parent, const char *name);
```

Description

This function is called to create a new section node. The name of the section is given by the `name` argument. The section could be a root section (in which case the `parent` is `NULL`) or a child of a previously-defined section (given as the `parent` argument).

Once created, a section node will typically be used as a parent of other section nodes or text nodes.

Example

The following example is extracted from the OVP ARC processor model:

```
VMI_DOC_FN(arcDoc) {

    arcP        arc  = (arcP)processor;
    vmiDocNodeP root = vmidocAddSection(0, "Root");

    // emit architecture description
    vmiDocNodeP desc = vmidocAddSection(root, "Description");
    if(isARC600(arc)) {
        vmidocAddText(desc, "ARC 600 processor model (ARCv1 architecture)");
    } else if(isARC700(arc)) {
        vmidocAddText(desc, "ARC 700 processor model (ARCv1 architecture)");
    } else if(isARCv2(arc)) {
        vmidocAddText(desc, "ARCv2 architecture processor model");
    }

    . . . lines omitted for clarity . . .

    vmidocProcessor(processor, root);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

27.3 vmidocAddText

Prototype

```
vmiDocNodeP vmidocAddText(vmiDocNodeP node, const char *text);
```

Description

This function is called to create a new text node, defining a paragraph with the given text. Leaf-level sections will typically contain one or more text nodes.

Example

The following example is extracted OPV ARC processor model:

```
VMI_DOC_FN(arcDoc) {

    arcP        arc  = (arcP)processor;
    vmiDocNodeP root = vmidocAddSection(0, "Root");

    // emit architecture description
    vmiDocNodeP desc = vmidocAddSection(root, "Description");
    if(isARC600(arc)) {
        vmidocAddText(desc, "ARC 600 processor model (ARCv1 architecture)");
    } else if(isARC700(arc)) {
        vmidocAddText(desc, "ARC 700 processor model (ARCv1 architecture)");
    } else if(isARCv2(arc)) {
        vmidocAddText(desc, "ARCv2 architecture processor model");
    }

    . . . lines omitted for clarity . . .

    vmidocProcessor(processor, root);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

27.4 vmidocProcessor

Prototype

```
void vmidocProcessor(vmiProcessorP processor, vmiDocNodeP document);
```

Description

This function is called to associate a root documentation node with a processor. It should be called once the tree of nodes for a processor or intercept library has been created.

Example

The following example is extracted OPV ARC processor model:

```
VMI_DOC_FN(arcDoc) {

    arcP        arc  = (arcP)processor;
    vmiDocNodeP root = vmidocAddSection(0, "Root");

    // emit architecture description
    vmiDocNodeP desc = vmidocAddSection(root, "Description");
    if(isARC600(arc)) {
        vmidocAddText(desc, "ARC 600 processor model (ARCv1 architecture)");
    } else if(isARC700(arc)) {
        vmidocAddText(desc, "ARC 700 processor model (ARCv1 architecture)");
    } else if(isARCv2(arc)) {
        vmidocAddText(desc, "ARCv2 architecture processor model");
    }

    . . . lines omitted for clarity . . .

    vmidocProcessor(processor, root);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

27.5 vmidocAddFields

Prototype

```
vmiDocNodeP vmidocAddFields(
    vmiDocNodeP parent,
    const char *name,
    Uns32       width
);
```

Description

This function is called to create a new documentation node, defining a collection of fields that make up a register or instruction format. The `width` parameter should specify the overall width of the register or instruction, usually in bits.

Example

An example at the end of this section shows how to document an instruction.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

27.6 vmidocAddField

Prototype

```
vmiDocNodeP vmidocAddField(
    vmiDocNodeP parent,
    const char *name,
    Uns32       offset,
    Uns32       width
);
```

Description

This function is called to create a new documentation node, defining a field within a collection of fields that make up a register or instruction format. The `width` parameter should specify the width of the field, usually in bits. The `offset` is specified in bits from the least significant end. The parent node must be created with `vmidocAddFields`.

Example

An example at the end of this section shows how to document an instruction.

Notes and Restrictions

Each field can be added to a fields collection in any order but must not overlap with other fields. The most significant bit of the most significant field must not exceed the width of the fields collection.

QuantumLeap Semantics

Synchronizing

27.7 vmidocAddConstField

Prototype

```
vmiDocNodeP vmidocAddConstField(
    vmiDocNodeP parent,
    Uns64       value,
    Uns32       offset,
    Uns32       width
);
```

Description

This function is called to create a new documentation node, defining a field within a collection of fields that has a constant value. The `width` parameter should specify the width of the field, usually in bits. The parent node must be created with `vmidocAddFields`.

Example

An example at the end of this section shows how to document an instruction.

Notes and Restrictions

Each field can be added to a fields collection in any order but must not overlap with other fields. The most significant bit of the most significant field must be less than the width of the fields collection. The `value` should not be larger than the specified number of bits can represent.

QuantumLeap Semantics

Synchronizing

27.8 Describing a field

Note that the node returned by `vmidocAddField` (and `vmidocAddConstField`) can have further text nodes attached to describe it.

Example

```
VMI_DOC_FN(arcDoc) {

    // Create a new chapter
    vmiDocNodeP chapter = vmidocAddSection(0, "Added instructions");

    // Document an instruction
    vmiDocNodeP INST1 = vmidocAddFields(chapter, "INST1", 16);

        // Add fields to the instruction
        vmidocAddField(INST1, "FIELD1",   0, 4);
        vmidocAddField(INST1, "FIELD2",   4, 4);
        vmiDocNodeP FIELD3 = vmidocAddField(INST1, “FIELD3",   8, 4);
        vmidocAddConstField(INST1,    1,  12,4);

        vmidocAddText(FIELD3, “This is field3.”);

      . . . more instructions, omitted for clarity . . .

    // Add the chapter to the model’s documentation.
    vmidocProcessor(processor, chapter);
}
```

Used with the Imperas documentation generator, this will produce this documentation:

Chapter 1. Added instructions

1.1 INST1

15 *12*	*11* *8*	*7* *4*	*3* *0*
0x1	*FIELD3*	*FIELD2*	*FIELD1*

1.1.1 FIELD3
This is field3.

27.9 vmidocAddFieldMSLS

Prototype

```
vmidocAddFieldMSLS(
    parent,
    name,
    msb,
    lsb
)
```

Description

A macro to document a field specifying `MSB` and `LSB` instead of `offset` and `width`.

27.10 vmidocAddConstFieldMSLS

Prototype

```
vmidocAddConstFieldMSLS(
    parent,
    value,
    msb,
    lsb
)
```

Description A macro to create a constant field specifying `MSB` and `LSB` instead of `offset` and `width`.

27.11 vmidocAddTable

Prototype

```
vmiDocNodeP vmidocAddTable(
    vmiDocNodeP      parent,
    const char      *title,
   vmidocTableAttrs  attrs
    uns32            columns,
    const char      *alignments,
    const char      *columnNames
)
```

Description

Create a table and specify the columns, alignment, column headings and special attributes. This function is used with `vmidocAddTableRow` which adds each row to the table.

`parent`	specifies an existing paragraph doc.
`title`	appears below the table.
`attrs`	add features to the table.
`columns`	in the number of columns in the heading and every row of the table.
`alignments`	a list of strings, one per column, containing “l” (left aligned), “c” (centered) or “r” (right aligned).
`columnNames`	the contents of the top row of the table.

Example

```
    // Create a new table
    vmiDocNodeP myTable = vmidocAddTable(
        paragraphNode,          // already existing paragraph
        “My Table”,             // the title, which appears below the table
        VMIDOC_TABLE_DEFAULT,   // no special attributes
        3,                      // the number of columns
        VMI_STRING_LIST(“l”, “c”, “r”),
        VMI_STRING_LIST(“COLUMN 1”, “COLUMN 2”, “COLUMN 3”),
    );
    vmidocAddTable(
        myTable,                // add the row to this table
        3,                      // the number of columns.
                                // Must be the same as the table (at present)
        VMI_STRING_LIST(“item 1”, “item 2”, “item 3”)
    );
```

Notes

`alignments` and `columnNames` are lists of constant strings which, in Gnu C, must be cast to that type. The macro `VMI_STRING_LIST()` takes care of this.

The values available for the `attrs` parameter are `VMIDOC_TABLE_DEFAULT` and `VMIDOC_TABLE_SMALL_FONT`, useful when the table has many columns.

At present, the number of columns in each table row must match the number of columns in the table. The lists in alignments, columnNames and row must contain the same number of entries as columns.

Empty columns may be specified using an empty string “”.

27.12 vmidocAddTableRow

Prototype

```
vmiDocNodeP vmidocAddTableRow(
    vmiDocNodeP      table,
    uns32            columns,
    const char      *row
)
```

Description

Add a row of entries to a table. This function is used with `vmidocAddTable` which creates the table.

`Table`	specifies an existing table doc node.
`columns`	in the number of columns in the heading and every row of the table.
`row`	a list of strings, one per column, the contents of one row.

28 Semihost and Debugger Integration Support Functions

This section describes functions that are used when integrating OVP processor models with client debug or analysis environments, and also when implementing semihost libraries.

28.1 vmirtDisassemble

Prototype

```
const char *vmirtDisassemble(
    vmiProcessorP  processor,
    Addr           simPC,
    vmiDisassAttrs attrs
);
```

Description

Given a processor and a simulated address, this function returns the disassembled instruction at that address, as interpreted by the processor in its current state. The result is valid only until the next call to `vmirtDisassemble` or `vmirtDisassembleInstr`.

Note that for processors with distinct instruction set states (for example ARM/Thumb or MIPS/microMIPS) the disassembly is performed in the *current processor state*. There is no guarantee that the instruction at the given address is a valid instruction in that state.

The `attrs` bitfield argument is used to indicate the type of disassembly required, defined by type `vmiDisassAttrs` in file `vmiTypes.h` as follows:

```
typedef enum vmiDisassAttrsE {
    DSA_NORMAL   = 0x00000000,  // normal disassembly
    DSA_UNCOOKED = 0x00000001,  // model-specific uncooked format
    DSA_BASE     = 0x00000002,  // use base model disassembly (not intercept)
    DSA_MODEL    = 0x80000000,  // model-specific mask
} vmiDisassAttrs;
```

A value of `DSA_NORMAL` indicates that disassembly in the normal display format is required.

A value of `DSA_UNCOOKED` indicates that disassembly in *uncooked* format is required. Uncooked format can be used to obtain information about the instruction that is easier to parse by downstream tools. The exact format of uncooked disassembly is model-specific.

A value of `DSA_BASE` indicates that disassembly should be performed using the disassembly callback associated with the processor model only. If this bit is absent, then disassembly functions associated with intercept libraries will be used in preference, and the processor model disassembly function will be used only if no intercept library disassembly function generates a non-`NULL` result.

A value of `DSA_MODEL` combined with any other numeric value by bitwise-or indicates that some processor-specific disassembly format is required.

Example

The following example is extracted from the standard ARM AngelTrap semihost library. It shows how the result of `vmirtDisassemble` is used to detect `SVC` and `HLT` instructions that must be handled by the semihost library.

```
static Bool matchOpcode(const char *disass, const char *opcode) {

    // get opcode length
    Uns32 opBytes = strlen(opcode);

    // does opcode match value in disassembly?
    return !strncmp(disass, opcode, opBytes) && (disass[opBytes]==' ');
}

static VMIOS_MORPH_FN(armOSOperation) {

    // disassemble this instruction
    const char *disass = vmirtDisassemble(processor, thisPC, DSA_UNCOOKED);

    if(matchOpcode(disass, "svc") || matchOpcode(disass, "bkpt")) {

        // svc/bkpt instruction: get condition and constant
        armCondition cond      = parseCond(disass);
        Uns32        arg       = parseConst(disass);
        Bool         thumbMode = parseThumbMode(disass);

        // ARM Angel semi-hosting calls are indicated by:
        //  SVC  0x123456 on Cortex-A and -R in ARM mode or by
        //  SVC  0xab     on Cortex-A and -R in Thumb mode or by
        //  BKPT 0xab     on Cortex-M (which only supports Thumb mode)
        Uns32 angelCode = thumbMode ? AngelSVC_THUMB : AngelSVC_ARM;

        // only intercept if code is angel code
        if(arg == angelCode) {
            *context  = "armAngelSyncTrap";
            *opaque   = True;
            *userData = (void *)cond;
            return armAngelSyncTrap;
        }

    } else if(matchOpcode(disass, "hlt")) {

        // hlt instruction: get constant
        Uns32 arg = parseConst(disass);

        // only intercept if code is angel code
        if(arg == AngelHLT) {
            *context  = "armAngelSyncTrap";
            *opaque   = True;
            *userData = (void *)ARM_C_AL;
            return armAngelSyncTrap;
        }

    } else if(matchOpcode(disass, "b")) {

        // get condition and target
        armCondition cond = parseCond(disass);
        Uns32        arg  = parseTarget(disass);

        // intercept branch-to-self
        if(arg == thisPC) {
            *context  = "armAngelSyncTrap";
            *opaque   = True;
            *userData = (void *)cond;
            return armAngelBTS;
        }
    }

    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics
Synchronizing

28.2 vmirtDisassembleInstr

Prototype

```
const char *vmirtDisassembleInstr(
    vmiProcessorP  processor,
    Addr           simPC,
    void          *instrBytes,
    vmiDisassAttrs attrs
);
```

Description

Given a processor and a pointer to an instruction byte pattern, this function returns the disassembled instruction, assuming it is located at the specified address, as interpreted by the processor in its current state. The result is valid only until the next call to `vmirtDisassemble` or `vmirtDisassembleInstr`.

Note that for processors with distinct instruction set states (for example ARM/Thumb or MIPS/microMIPS) the disassembly is performed in the *current processor state*. There is no guarantee that the instruction at the given address is a valid instruction in that state.

The `attrs` bitfield argument is used to indicate the type of disassembly required, defined by type `vmiDisassAttrs` in file `vmiTypes.h` as follows:

```
typedef enum vmiDisassAttrsE {
    DSA_NORMAL   = 0x00000000,  // normal disassembly
    DSA_UNCOOKED = 0x00000001,  // model-specific uncooked format
    DSA_BASE     = 0x00000002,  // use base model disassembly (not intercept)
    DSA_MODEL    = 0x80000000,  // model-specific mask
} vmiDisassAttrs;
```

A value of `DSA_NORMAL` indicates that disassembly in the normal display format is required.

A value of `DSA_UNCOOKED` indicates that disassembly in *uncooked* format is required. Uncooked format can be used to obtain information about the instruction that is easier to parse by downstream tools. The exact format of uncooked disassembly is model-specific.

A value of `DSA_BASE` indicates that disassembly should be performed using the disassembly callback associated with the processor model only. If this bit is absent, then disassembly functions associated with intercept libraries will be used in preference, and the processor model disassembly function will be used only if no intercept library disassembly function generates a non-`NULL` result.

A value of `DSA_MODEL` combined with any other numeric value by bitwise-or indicates that some processor-specific disassembly format is required.

Example

The following example shows how this function might be used in an intercept library to disassemble a block of `numInst` instructions at a passed address. The code assumes that only 2-byte and 4-byte instruction encodings are used.

```
static void disassembleBlock(
    vmiProcessorP processor,
    Addr          simPC,
    Uns32         numInst
) {
    Uns32 i;

    for(i=0; i<numInst; i++) {

        // get instruction size
        Uns32 bytes = vmirtInstructionBytes(processor, simPC);
        Uns32 instr = 0;

        // get instruction bytes
        vmirtReadNByteDomain(
            vmirtGetProcessorCodeDomain(processor),
            simPC,
            &instr,
            bytes,
            0,
            MEM_AA_FALSE
        );

        // disassemble instruction
        const char *disass = vmirtDisassembleInstr(
            processor, simPC, &instr, DSA_NORMAL
        );

        // show disassembly
        if(bytes==4) {
            vmiPrintf(FMT_Ax": %08x : %s\n", simPC, instr, disass);
        } else {
            vmiPrintf(FMT_Ax": %04x     : %s\n", simPC, instr, disass);
        }

        // step to nest instruction
        simPC += bytes;
    }
}
```

Notes and Restrictions

1. The caller is responsible for validating that the instructions bytes referenced by `instrBytes` comprise a complete instruction.

QuantumLeap Semantics

Synchronizing

28.3 vmirtInstructionBytes

Prototype

```
Uns32 vmirtInstructionBytes(vmiProcessorP processor, Addr simPC);
```

Description

Given a processor and a simulated address, this function returns the size in bytes of the instruction at that address, as interpreted by the processor in its current state.

Note that for processors with distinct instruction set states (for example ARM/Thumb or MIPS/microMIPS) the result is determined in the *current processor state*. There is no guarantee that the instruction at the given address is a valid instruction in that state.

Example

This example is taken from an extension library that adds extra 4-byte instructions to a RISC-V processor. This processor has both 2-byte and 4-byte instructions; before attempting to decode the instruction at a given address, the extension library first checks that the address indeed contains a 4-byte instruction (if not, it cannot be one of the added instructions). Failure to do this would mean that the extension library would cause every instruction fetch to be 4 bytes, even when the instruction is in fact only two bytes in size, possibly perturbing model behavior (for example, by causing unexpected page table walks).

```
static riscvEnhancedInstrType getInstrType(
    vmiosObjectP  object,
    vmiProcessorP processor,
    Addr          thisPC
) {

    riscvEnhancedInstrType type  = RISCV_EIT_LAST;
    Uns32                  bytes = vmirtInstructionBytes(processor, thisPC);

    if (bytes == 4) {
        Uns32 instruction = vmicxtFetch4Byte(processor, thisPC);
        type = vmidDecode(object->table, instruction);
    }

    return type;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

29 Debugger Integration Support Functions

This section describes functions that are used when integrating OVP processor models with client debug or analysis environments. These functions are available only with the Imperas Professional Tools.

29.1 vmirtEvaluateGDBExpression

Prototype

```
const char *vmirtEvaluateGDBExpression(
    vmiProcessorP processor,
    const char   *expression,
    Bool          usePhysicalDomain,
    Bool          quiet
);
```

Description

This function evaluates an expression using a GDB attached to the given processor. The result is returned as a string, which remains valid only until the next call to `vmirtEvaluateGDBExpression`. The argument `usePhysicalDomain` specifies whether any memory accesses implied by the expression use physical addresses (`True`) or virtual addresses (`False`). The quiet argument causes GDB errors resulting from evaluation of the expression to be suppressed.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

2. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

29.2 vmirtEvaluateCodeLocation

Prototype

```
Bool vmirtEvaluateCodeLocation(
    vmiProcessorP processor,
    const char   *expression,
    Addr         *address,
    char        **sourceFile,
    char        **fullFile,
    unsigned     *sourceLine
);
```

Description

Given a processor and an expression, this function evaluates the expression using a GDB attached to the processor to yield an address, source file, full file and source line result. The expression should be of a form suitable for use with the GDB `break` command. The purpose of this function is to enable client debuggers to place breakpoints appropriately.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

29.3 vmirtAddRegisterWatchCallback

Prototype

```
vmiWatchHandleP vmirtAddRegisterWatchCallback(
    vmiProcessorP processor,
    vmiRegInfoCP  reg,
    vmiRegValueFn valueCB,
    void         *userData
);
```

Description

Given a processor and a `vmiRegInfoCP` descriptor for a register in that processor, this function inserts a watch point on that register. After any instruction in which the register changes, the notifier function `valueCB` is called. The notifier is prototyped in `vmiTypes.h` as:

```
 #define VMI_REG_VALUE_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    vmiRegInfoCP  reg,          \
    Addr          simPC,        \
    void         *oldValue,     \
    void         *newValue,     \
    void         *userData      \
)
typedef VMI_REG_VALUE_FN((*vmiRegValueFn));
```

As well as the processor and register, the notifier is passed the following arguments:

1. The simulated address of the instruction that changed the register.
2. A pointer to the previous register value.
3. A pointer to the new register value.
4. A `userData` pointer, passed to `vmirtAddRegisterCallback` when the watchpoint was created.

Typically, this function is used to create customized trace output.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

29.4 vmirtDeleteRegisterWatchCallback

Prototype

```
void vmirtDeleteRegisterWatchCallback(vmiWatchHandleP handle);
```

Description

This function deletes a watch point previously created by `vmirtAddRegisterCallback`.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

30 Shared Object Access

It is sometimes useful to be able to load shared objects (or dynamically-linked libraries on Windows) into processor models. As an example, a processor model might allow user-defined instructions to be provided using a separate shared object.

This section describes functions for opening and closing shared objects, and for determining a symbol address in the loaded object. There is also a utility function to obtain any error generated by the object loader.

30.1 vmirtDLOpen

Prototype

```
vmiDLHandleP vmirtDLOpen(const char *fileName);
```

Description

This function attempts to open a shared object or dynamically-linked library of the passed name. If successful, the function returns an object of type `vmiDLHandleP`, which can be used subsequently to find symbol addresses in the loaded object (see function `vmirtDLSymbol`). If the load is unsuccessful, the function returns `NULL`; function `vmirtDLError` may be called to get further information about the cause of failure.

Example

```
Bool cpuExtInstall(cpuP cpu) {

    const char  *extFileAttr = "extFileName";
    const char  *configFn    = "cpuConfig";
    vmiDLHandleP handle;
    extConfigFn  configCB;

    // get the name of any extension file
    const char *extFileName = vmirtPlatformStringAttribute(
        (vmiProcessorP)cpu, extFileAttr
    );

    if(!extFileName) {

        // no extension file given

    } else if(!(handle=vmirtDLOpen(extFileName))) {

        // bad extension file
        vmiMessage("E", PREFIX "EXT_BFN",
            "Error loading extension file %s (%s)",
            extFileName, vmirtDLError()
        );

    } else if(!(configCB=vmirtDLSymbol(handle, configFn))) {

        // configuration function not found
        vmiMessage("E", PREFIX "EXT_SNF",
            "Configuration function %s not found in extension file %s",
            configFn, extFileName
        );

    } else {

        // .s0/.dll successfully loaded
        cpu->handle = handle;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

30.2 vmirtDLClose

Prototype

```
Int32 vmirtDLClose(vmiDLHandleP handle);
```

Description

This function closes a shared object of dynamically-linked library previously opened by `vmirtDLOpen`. This is usually not required except in special circumstances where a library must be unloaded while the program is executing.

Example

```
Bool cpuExtUninstall(cpuP cpu) {

    if(cpu->handle) {
        vmirtDLClose(cpu->handle);
        cpu->handle = 0;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

30.3 vmirtDLSymbol

Prototype

```
void *vmirtDLSymbol(vmiDLHandleP handle, const char *symbol);
```

Description

This function searches for the passed symbol in a previously-opened shared object of dynamically linked library. If the symbol is found, the function returns its address; if the symbol is not found, the function returns `NULL`.

Example

```
Bool cpuExtInstall(cpuP cpu) {

    const char  *extFileAttr = "extFileName";
    const char  *configFn    = "cpuConfig";
    vmiDLHandleP handle;
    extConfigFn  configCB;

    // get the name of any extension file
    const char *extFileName = vmirtPlatformStringAttribute(
        (vmiProcessorP)cpu, extFileAttr
    );

    if(!extFileName) {

        // no extension file given

    } else if(!(handle=vmirtDLOpen(extFileName))) {

        // bad extension file
        vmiMessage("E", PREFIX "EXT_BFN",
            "Error loading extension file %s (%s)",
            extFileName, vmirtDLError()
        );

    } else if(!(configCB=vmirtDLSymbol(handle, configFn))) {

        // configuration function not found
        vmiMessage("E", PREFIX "EXT_SNF",
            "Configuration function %s not found in extension file %s",
            configFn, extFileName
        );

    } else {

        // .s0/.dll successfully loaded
        cpu->handle = handle;
    }
}
```

Notes and Restrictions

1. It is the client program’s responsibility to ensure that the returned symbol is of an appropriate type if required – for example, there is no check that a symbol that is to be called as a function is in fact a function pointer.

QuantumLeap Semantics

Synchronizing

30.4 vmirtDLError

Prototype

```
const char *vmirtDLError(void);
```

Description

This function returns any error message from the previous call to `vmirtDLOpen`, `vmirtDLLClose` or `vmirtDLSymbol`. If there was no error, `NULL` is returned

Example

```
Bool cpuExtInstall(cpuP cpu) {

    const char  *extFileAttr = "extFileName";
    const char  *configFn    = "cpuConfig";
    vmiDLHandleP handle;
    extConfigFn  configCB;

    // get the name of any extension file
    const char *extFileName = vmirtPlatformStringAttribute(
        (vmiProcessorP)cpu, extFileAttr
    );

    if(!extFileName) {

        // no extension file given

    } else if(!(handle=vmirtDLOpen(extFileName))) {

        // bad extension file
        vmiMessage("E", PREFIX "EXT_BFN",
            "Error loading extension file %s (%s)",
            extFileName, vmirtDLError()
        );

    } else if(!(configCB=vmirtDLSymbol(handle, configFn))) {

        // configuration function not found
        vmiMessage("E", PREFIX "EXT_SNF",
            "Configuration function %s not found in extension file %s",
            configFn, extFileName
        );

    } else {

        // .s0/.dll successfully loaded
        cpu->handle = handle;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

31 Instruction Attributes

When writing intercept libraries, it is very often the case that the current processor instruction needs to be introspected. To simplify this process, the simulator supports a generic *instruction attributes* API. Using this API, it is possible to obtain the following information about an instruction:

1. The instruction *class*;
2. The *address* and *size* of the instruction fetches;
3. The registers *read* by the instruction;
4. The registers *written* by the instruction;
5. The potential *next instruction address* (or *addresses*, in the case of a conditional branch);
6. The *addresses* and *sizes* of any loads or stores performed by the instruction.

This information is all presented using the instruction attributes API described in the files `ImpPublic/include/host/oclia.h` and `ImpPublic/include/host/ocliaTypes.h`. The same interface functions can be used by both the VMI API and the OP API: when using the OP API, use function `opProcessorInstructionAttributes` to obtain instruction attributes for a given address; when using the VMI API, use function `vmiiaGetAttrs` (defined in header file `vmiInstructionAttrs.h`).

Most information returned by the instruction attributes API is automatically determined from the VMI morph-time primitives that are used to implement the instruction. In some cases the information cannot be completely determined automatically and extra calls must be made to present a complete view: see the *VMI Morph Time Function Reference Manual* for more information.

This section first describes the VMI-specific instruction attributes functions (defined in file `vmiInstructionAttrs.h`), and then gives information about the common instruction attributes functions, usable from both VMI and OP interfaces (defined in `oclia.h`).

31.1 vmiiaGetAttrs

Prototype

```
octiaAttrP vmiiaGetAttrs(
    vmiProcessorP   processor,
    Addr            simPC,
    octiaDataSelect select,
    Bool            applyDFA
);
```

Description

This function returns instruction attribute for the given processor for the instruction at address `simPC`. The precise information required is specified using the `select` parameter of type `octiaDataSelect`, defined as a bitmask enumeration in `ocliaTypes.h` as:

```
typedef enum octiaDataSelectE {
    OCL_DS_NONE    = 0x00, ///< empty mask
    OCL_DS_NODES   = 0x01, ///< record node list
    OCL_DS_REG_R   = 0x02, ///< record debug interface registers read
    OCL_DS_REG_W   = 0x04, ///< record debug interface registers written
    OCL_DS_RANGE_R = 0x08, ///< record field ranges read (and not registers)
    OCL_DS_RANGE_W = 0x10, ///< record field ranges written (and not registers)
    OCL_DS_FETCH   = 0x20, ///< record fetch ranges
    OCL_DS_NEXTPC  = 0x40, ///< record next PC expressions
    OCL_DS_ADDRESS = 0x80  ///< record load/store address expressions
} octiaDataSelect;
```

Members of this type are as follows:

1. `OCL_DS_NODES`: This option indicates that NMI node objects should be returned. This option is for internal use only.
2. `OCL_DS_REG_R`: This option indicates that registers read by the instruction should be returned.
3. `OCL_DS_REG_W`: This option indicates that registers written by the instruction should be returned.
4. `OCL_DS_RANGE_R`: This option indicates that information about processor structure fields read by the instruction should be returned. This is not normally required except when debugging the development of instruction attributes for a processor.
5. `OCL_DS_RANGE_W`: This option indicates that information about processor structure fields written by the instruction should be returned. This is not normally required except when debugging the development of instruction attributes for a processor.
6. `OCL_DS_FETCH`: This option indicates that information about the fetches made to decode the instruction should be returned.
7. `OCL_DS_NEXTPC`: This option indicates that information about possible next program counter values should be returned.
8. `OCL_DS_ADDRESS`: This option indicates that information about load/store addresses and sizes should be returned.

The `applyDFA` parameter indicates whether the returned instruction attributes should take into account dataflow analysis for an instruction. If `applyDFA` is `True`, then some register accesses implied by the VMI morph-time primitives that implement an instruction may be removed if the simulator determines that they have no effect. If `applyDFA` is `False`, then information about such accesses will be preserved.

If the processor is modal, then the returned instruction attributes are correct for the current processor mode.

When attributes for an instruction are no longer required, the returned structure should be freed using function `ocliaFreeAttrs`.

Example

The following template shows how this function will typically be used in the morpher callback in an intercept library:

```
#define SELECT_ATTRS (  \
    OCL_DS_REG_R   |    \
    OCL_DS_REG_W   |    \
    OCL_DS_RANGE_R |    \
    OCL_DS_RANGE_W |    \
    OCL_DS_FETCH   |    \
    OCL_DS_NEXTPC  |    \
    OCL_DS_ADDRESS      \
)

static VMIOS_MORPH_FN(morphCallback) {

    // get instruction attributes
    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, SELECT_ATTRS, False);

    if(attrs) {

        // print instruction disassembly
        vmiPrintf(
            "*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x"FMT_Ax" (%s)\n",
            thisPC,
            vmirtDisassemble(processor, thisPC, DSA_NORMAL)
        );

        // walk attributes
        walkAttrs(processor, attrs);

        // free attributes
        ocliaFreeAttrs(attrs);
    }

    // indicate that normal instruction translation should be done
    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

31.2 vmiiaConvertRegInfo

Prototype

```
vmiRegInfoCP vmiiaConvertRegInfo(octiaRegInfoP regInfo);
```

Description

Using the instruction attributes API, descriptions of registers used by an instruction are returned using the `octiaRegInfoP` type. This function returns the equivalent `vmiRegInfoCP` object for that type, so that the register description can be used in an intercept library context.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRegList(octiaRegListP regList, const char *type) {

    if(regList) {

        vmiPrintf("\n");

        for(; regList; regList=ocliaGetRegListNext(regList)) {
            vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(regList));
            vmiPrintf("  %5s %s\n", type, reg->name);
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.3 ocliaFreeAttrs

Prototype

```
void ocliaFreeAttrs (
    octiaAttrP attrs
);
```

Description

Each call to `opProcessorInstructionAttributes` or `vmiiaGetAttrs` allocates a structure to hold the instruction attributes for that instruction. The structure persists until `ocliaFreeAttrs` is called.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
#define SELECT_ATTRS (  \
    OCL_DS_REG_R   |    \
    OCL_DS_REG_W   |    \
    OCL_DS_RANGE_R |    \
    OCL_DS_RANGE_W |    \
    OCL_DS_FETCH   |    \
    OCL_DS_NEXTPC  |    \
    OCL_DS_ADDRESS      \
)

static VMIOS_MORPH_FN(morphCallback) {

    // get instruction attributes
    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, SELECT_ATTRS, False);

    if(attrs) {

        // print instruction disassembly
        vmiPrintf(
            "*** ATTRIBUTES FOR INSTRUCTION AT ADDRESS 0x"FMT_Ax" (%s)\n",
            thisPC,
            vmirtDisassemble(processor, thisPC, DSA_NORMAL)
        );

        // walk attributes
        walkAttrs(processor, attrs);

        // free attributes
        ocliaFreeAttrs(attrs);
    }

    // indicate that normal instruction translation should be done
    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.4 ocliaPrintAttrs

Prototype

```
void ocliaPrintAttrs (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function prints all aspects of that object.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
#define SELECT_ATTRS (  \
    OCL_DS_REG_R   |    \
    OCL_DS_REG_W   |    \
    OCL_DS_RANGE_R |    \
    OCL_DS_RANGE_W |    \
    OCL_DS_FETCH   |    \
    OCL_DS_NEXTPC  |    \
    OCL_DS_ADDRESS      \
)

static VMIOS_MORPH_FN(morphCallback) {

    // get instruction attributes
    octiaAttrP attrs = vmiiaGetAttrs(processor, thisPC, SELECT_ATTRS, False);

    if(attrs) {

        // print attributes
        ocliaPrintAttrs(attrs);

        // free attributes
        ocliaFreeAttrs(attrs);
    }

    // indicate that normal instruction translation should be done
    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.5 ocliaSetAttrsUserData, ocliaGetAttrsUserData

Prototypes

```
void ocliaSetAttrsUserData (
    octiaAttrP attrs,
    void *     userData
);

void * ocliaGetAttrsUserData (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, these functions assign and retrieve client-specific data associated with that object, respectively. This allows instruction attribute information to be extended if required.

Client code is responsible for memory management of the client data if required.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.6 ocliaGetInstructionClass

Prototype

```
octiaInstructionClass ocliaGetInstructionClass (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the *class* of the instruction, as a bit field enumeration. Classes can be standard classes specified using the `ictiaInstructionClass` type, or custom processor-specific classes:

```
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;
```

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
#define CLASS_ENTRY(_NAME) case OCL_IC_##_NAME: vmiPrintf(#_NAME); break

static void printClass(octiaAttrP attrs) {

    octiaInstructionClass class;

    if((class=ocliaGetInstructionClass(attrs))) {

        octiaInstructionClass mask = 1;

        vmiPrintf("\n");
        vmiPrintf("  class ");

        do {

            if(class & mask) {

                switch(mask) {
                    CLASS_ENTRY(NOP       );
                    CLASS_ENTRY(INTEGER   );
                    CLASS_ENTRY(FLOAT     );
                    CLASS_ENTRY(DSP       );
                    CLASS_ENTRY(MULTIPLY  );
                    CLASS_ENTRY(DIVIDE    );
                    CLASS_ENTRY(FMA       );
                    CLASS_ENTRY(SIMD      );
                    CLASS_ENTRY(TRIG      );
                    CLASS_ENTRY(LOG       );
                    CLASS_ENTRY(RECIP     );
                    CLASS_ENTRY(SQRT      );
                    CLASS_ENTRY(SYSREG    );
                    CLASS_ENTRY(IBARRIER  );
                    CLASS_ENTRY(DBARRIER  );
                    CLASS_ENTRY(ABARRIER  );
                    CLASS_ENTRY(ICACHE    );
                    CLASS_ENTRY(DCACHE    );
                    CLASS_ENTRY(MMU       );
                    CLASS_ENTRY(ATOMIC    );
                    CLASS_ENTRY(EXCLUSIVE );
                    CLASS_ENTRY(HINT      );
                    CLASS_ENTRY(SYSTEM    );
                    CLASS_ENTRY(FCONVERT  );
                    CLASS_ENTRY(FCOMPARE  );
                    CLASS_ENTRY(BRANCH    );
                    CLASS_ENTRY(BRANCH_DS );
                    CLASS_ENTRY(BRANCH_DSA);
                    CLASS_ENTRY(OPAQUE_INT);
                    CLASS_ENTRY(CUSTOM1   );
                    CLASS_ENTRY(CUSTOM2   );
                    CLASS_ENTRY(CUSTOM3   );
                    CLASS_ENTRY(CUSTOM4   );
                    CLASS_ENTRY(CUSTOM5   );
                    CLASS_ENTRY(CUSTOM6   );
                    CLASS_ENTRY(CUSTOM7   );
                    CLASS_ENTRY(CUSTOM8   );
                    CLASS_ENTRY(CUSTOM9   );
                    CLASS_ENTRY(CUSTOM10  );
                    CLASS_ENTRY(CUSTOM11  );
                    CLASS_ENTRY(CUSTOM12  );
                    CLASS_ENTRY(CUSTOM13  );
                    CLASS_ENTRY(CUSTOM14  );
                    CLASS_ENTRY(CUSTOM15  );
                    CLASS_ENTRY(CUSTOM16  );
                    default: vmiPrintf("*unknown*");
                }

                class &= ~mask;

                if(class) {
                    vmiPrintf("|");
                }
            }

            mask <<= 1;

        } while(class);

        vmiPrintf("\n");
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.7 ocliaGetFirstFetchRecord

Prototype

```
octiaFetchRecordP ocliaGetFirstFetchRecord (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first fetch record for that instruction. Fetch records describe each fetch performed by the processor when decoding the instruction, and can be further queried by functions `ocliaGetFetchRecordBytes`, `ocliaGetFetchRecordLow` and `ocliaGetFetchRecordHigh`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printFetchRecords(octiaAttrP attrs) {

    octiaFetchRecordP fr;

    if((fr=ocliaGetFirstFetchRecord(attrs))) {

        vmiPrintf("\n");

        for(; fr; fr=ocliaGetNextFetchRecord(fr)) {

            Addr  fetchLow  = ocliaGetFetchRecordLow(fr);
            Addr  fetchHigh = ocliaGetFetchRecordHigh(fr);
            Uns8 *value     = ocliaGetFetchRecordBytes(fr);
            Uns32 bytes     = fetchHigh-fetchLow+1;
            Int32 i;

            vmiPrintf(
                "  fetch 0x"FMT_Ax":0x"FMT_Ax" (value:0x",
                fetchLow, fetchHigh
            );

            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", value[i]);
            }

            vmiPrintf(")\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.8 ocliaGetNextFetchRecord

Prototype

```
octiaFetchRecordP ocliaGetNextFetchRecord (
    octiaFetchRecordP prev
);
```

Description

Given a fetch record, this function returns the *next* fetch record for that instruction. Fetch records describe each fetch performed by the processor when decoding the instruction, and can be further queried by functions `ocliaGetFetchRecordBytes`, `ocliaGetFetchRecordLow` and `ocliaGetFetchRecordHigh`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printFetchRecords(octiaAttrP attrs) {

    octiaFetchRecordP fr;

    if((fr=ocliaGetFirstFetchRecord(attrs))) {

        vmiPrintf("\n");

        for(; fr; fr=ocliaGetNextFetchRecord(fr)) {

            Addr  fetchLow  = ocliaGetFetchRecordLow(fr);
            Addr  fetchHigh = ocliaGetFetchRecordHigh(fr);
            Uns8 *value     = ocliaGetFetchRecordBytes(fr);
            Uns32 bytes     = fetchHigh-fetchLow+1;
            Int32 i;

            vmiPrintf(
                "  fetch 0x"FMT_Ax":0x"FMT_Ax" (value:0x",
                fetchLow, fetchHigh
            );

            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", value[i]);
            }

            vmiPrintf(")\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.9 ocliaGetFetchRecordBytes

Prototype

```
Uns8* ocliaGetFetchRecordBytes (
    octiaFetchRecordP fr
);
```

Description

Given a fetch record, this function returns a pointer to the bytes that were fetched as part of that fetch.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printFetchRecords(octiaAttrP attrs) {

    octiaFetchRecordP fr;

    if((fr=ocliaGetFirstFetchRecord(attrs))) {

        vmiPrintf("\n");

        for(; fr; fr=ocliaGetNextFetchRecord(fr)) {

            Addr  fetchLow  = ocliaGetFetchRecordLow(fr);
            Addr  fetchHigh = ocliaGetFetchRecordHigh(fr);
            Uns8 *value     = ocliaGetFetchRecordBytes(fr);
            Uns32 bytes     = fetchHigh-fetchLow+1;
            Int32 i;

            vmiPrintf(
                "  fetch 0x"FMT_Ax":0x"FMT_Ax" (value:0x",
                fetchLow, fetchHigh
            );

            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", value[i]);
            }

            vmiPrintf(")\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.10 ocliaGetFetchRecordLow

Prototype

```
Addr ocliaGetFetchRecordLow (
    octiaFetchRecordP fr
);
```

Description

Given a fetch record, this function returns a pointer to the low address of that fetch.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printFetchRecords(octiaAttrP attrs) {

    octiaFetchRecordP fr;

    if((fr=ocliaGetFirstFetchRecord(attrs))) {

        vmiPrintf("\n");

        for(; fr; fr=ocliaGetNextFetchRecord(fr)) {

            Addr  fetchLow  = ocliaGetFetchRecordLow(fr);
            Addr  fetchHigh = ocliaGetFetchRecordHigh(fr);
            Uns8 *value     = ocliaGetFetchRecordBytes(fr);
            Uns32 bytes     = fetchHigh-fetchLow+1;
            Int32 i;

            vmiPrintf(
                "  fetch 0x"FMT_Ax":0x"FMT_Ax" (value:0x",
                fetchLow, fetchHigh
            );

            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", value[i]);
            }

            vmiPrintf(")\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.11 ocliaGetFetchRecordHigh

Prototype

```
Addr ocliaGetFetchRecordHigh (
    octiaFetchRecordP fr
);
```

Description

Given a fetch record, this function returns a pointer to the high address of that fetch.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printFetchRecords(octiaAttrP attrs) {

    octiaFetchRecordP fr;

    if((fr=ocliaGetFirstFetchRecord(attrs))) {

        vmiPrintf("\n");

        for(; fr; fr=ocliaGetNextFetchRecord(fr)) {

            Addr  fetchLow  = ocliaGetFetchRecordLow(fr);
            Addr  fetchHigh = ocliaGetFetchRecordHigh(fr);
            Uns8 *value     = ocliaGetFetchRecordBytes(fr);
            Uns32 bytes     = fetchHigh-fetchLow+1;
            Int32 i;

            vmiPrintf(
                "  fetch 0x"FMT_Ax":0x"FMT_Ax" (value:0x",
                fetchLow, fetchHigh
            );

            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", value[i]);
            }

            vmiPrintf(")\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.12 ocliaSetFetchRecordUserData, ocliaGetFetchRecordUserData

Prototypes

```
void ocliaSetFetchRecordUserData (
    octiaFetchRecordP fr,
    void *            userData
);

void * ocliaGetFetchRecordUserData (
    octiaFetchRecordP fr
);
```

Description

Given an instruction attributes fetch record object, these functions assign and retrieve client-specific data associated with that object, respectively. This allows instruction attribute information to be extended if required.

Client code is responsible for memory management of the client data if required.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.13 ocliaGetFirstReadReg

Prototype

```
octiaRegListP ocliaGetFirstReadReg (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first *read register* list record for that instruction. Such records list every register used as an input to the instruction, and can be further queried by function `ocliaGetRegListReg`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRegList(octiaRegListP regList, const char *type) {

    if(regList) {

        vmiPrintf("\n");

        for(; regList; regList=ocliaGetRegListNext(regList)) {
            vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(regList));
            vmiPrintf("  %5s %s\n", type, reg->name);
        }
    }
}

static void walkAttrs(octiaAttrP attrs) {

    printClass(attrs);
    printFetchRecords(attrs);

    printRegList(ocliaGetFirstReadReg(attrs), "read");
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");

    printNextPCList(attrs);
    printAddressExpressionList(attrs);

    opPrintf("\n");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.14 ocliaGetFirstWrittenReg

Prototype

```
octiaRegListP ocliaGetFirstWrittenReg (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first *written register* list record for that instruction. Such records list every register assigned as an output by the instruction, and can be further queried by function `ocliaGetRegListReg`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRegList(octiaRegListP regList, const char *type) {

    if(regList) {

        vmiPrintf("\n");

        for(; regList; regList=ocliaGetRegListNext(regList)) {
            vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(regList));
            vmiPrintf("  %5s %s\n", type, reg->name);
        }
    }
}

static void walkAttrs(octiaAttrP attrs) {

    printClass(attrs);
    printFetchRecords(attrs);

    printRegList(ocliaGetFirstReadReg(attrs), "read");
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");

    printNextPCList(attrs);
    printAddressExpressionList(attrs);

    opPrintf("\n");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.15 ocliaGetRegListNext

Prototype

```
octiaRegListP ocliaGetRegListNext (
    octiaRegListP prev
);
```

Description

Given an instruction attributes register list object, this function returns the *next* register list record in the list. The function is used to iterate across read register and written register lists.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRegList(octiaRegListP regList, const char *type) {

    if(regList) {

        vmiPrintf("\n");

        for(; regList; regList=ocliaGetRegListNext(regList)) {
            vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(regList));
            vmiPrintf("  %5s %s\n", type, reg->name);
        }
    }
}

static void walkAttrs(octiaAttrP attrs) {

    printClass(attrs);
    printFetchRecords(attrs);

    printRegList(ocliaGetFirstReadReg(attrs), "read");
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");

    printNextPCList(attrs);
    printAddressExpressionList(attrs);

    opPrintf("\n");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.16 ocliaGetRegListReg

Prototype

```
octiaRegInfoP ocliaGetRegListReg (
    octiaRegListP entry
);
```

Description

Given an instruction attributes register list object, this function returns the *register* record for that instruction. In an intercept library context, the result of this will typically then be used as a parameter to function `vmiiaConvertRegInfo` (see section 31.2).

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRegList(octiaRegListP regList, const char *type) {

    if(regList) {

        vmiPrintf("\n");

        for(; regList; regList=ocliaGetRegListNext(regList)) {
            vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(regList));
            vmiPrintf("  %5s %s\n", type, reg->name);
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.17 ocliaSetRegListUserData, ocliaGetRegListUserData

Prototypes

```
void ocliaSetRegListUserData (
    octiaRegListP entry,
    void *        userData
);

void * ocliaGetRegListUserData (
    octiaRegListP entry
);
```

Description

Given an instruction attributes register list object, these functions assign and retrieve client-specific data associated with that object, respectively. This allows instruction attribute information to be extended if required.

Client code is responsible for memory management of the client data if required.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.18 ocliaGetFirstMemAccess

Prototype

```
octiaMemAccessP ocliaGetFirstMemAccess (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first *memory access* list record for that instruction. Such records list every load or store made by the instruction, and can be further queried by functions `ocliaGetMemAccessAddrExp`, `ocliaGetMemAccessDomain`, `ocliaGetMemAccessFirstDepend`, `ocliaGetMemAccessMemBits` and `ocliaGetMemAccessType`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddressExpressionList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        vmiPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            vmiPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            vmiPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(depend));
                vmiPrintf("    depend %s\n", reg->name);
            }
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.19 ocliaGetNextMemAccess

Prototype

```
octiaMemAccessP ocliaGetNextMemAccess (
    octiaMemAccessP prev
);
```

Description

Given an instruction attributes memory access object, this function returns the *next* memory access list record for that instruction. Such records list every load or store made by the instruction, and can be further queried by functions `ocliaGetMemAccessAddrExp`, `ocliaGetMemAccessDomain`, `ocliaGetMemAccessFirstDepend`, `ocliaGetMemAccessMemBits` and `ocliaGetMemAccessType`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddressExpressionList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        vmiPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            vmiPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            vmiPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(depend));
                vmiPrintf("    depend %s\n", reg->name);
            }
        }
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Thread Safe

31.20 ocliaGetMemAccessType

Prototype

```
octiaMemAccessType ocliaGetMemAccessType (
    octiaMemAccessP ma
);
```

Description

Given an instruction attributes memory access object, this function returns the type of access, given by the `octiaMemAccessType` enumeration:

```
typedef enum octiaMemAccessTypeE {
    OCL_MAT_LOAD       , ///< Load
    OCL_MAT_STORE      , ///< Store
    OCL_MAT_PRELOAD_LD , ///< Preload for likely load
    OCL_MAT_PRELOAD_ST , ///< Preload for likely store
    OCL_MAT_PRELOAD_EX   ///< Preload for likely fetch
} octiaMemAccessType;
```

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddressExpressionList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        vmiPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            vmiPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            vmiPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(depend));
                vmiPrintf("    depend %s\n", reg->name);
            }
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.21 ocliaGetMemAccessAddrExp

Prototype

```
octiaAddrExpP ocliaGetMemAccessAddrExp (
    octiaMemAccessP ma
);
```

Description

Given an instruction attributes memory access object, this function returns the *address expression* for that memory access. Address expressions are a tree of octiaAddrExp objects, defined in ocliaTypes.h as follows:

```
typedef enum octiaAddrExpTypeE {
    OCL_ET_UNKNOWN , ///< unknown value
    OCL_ET_CONST   , ///< constant value
    OCL_ET_REG     , ///< register value
    OCL_ET_EXTEND  , ///< extended value
    OCL_ET_UNARY   , ///< unary value
    OCL_ET_BINARY  , ///< binary value
    OCL_ET_LOAD      ///< load from address
} octiaAddrExpType;

typedef struct octiaAddrExpS {
    octiaAddrExpType type; ///< expression type
    Uns32            bits; ///< expression bit size
    union {
        Offset             c; ///< if type==OCL_ET_CONST
        octiaRegInfoP      r; ///< if type==OCL_ET_REG
        octiaAddrExpExtend e; ///< if type==OCL_ET_EXTEND
        octiaAddrExpUnary  u; ///< if type==OCL_ET_UNARY
        octiaAddrExpBinary b; ///< if type==OCL_ET_BINARY
        octiaAddrExpLoad   l; ///< if type==OCL_ET_LOAD
    };
} octiaAddrExp;
```

An expression can be one of these types (in the `type` field):

1. `OCL_ET_UNKNOWN`: an expression of unknown type. This should not be seen in practice;
2. `OCL_ET_CONST`: a constant value, held in the `c` field;
3. `OCL_ET_REG`: a register value, held in the `r` field;
4. `OCL_ET_EXTEND`: an extended value, held in the `e` field;
5. `OCL_ET_UNARY`: a unary expression value, held in the `u` field;
6. `OCL_ET_BINARY`: a binary expression value, held in the `b` field;
7. `OCL_ET_LOAD`: a load expression value, held in the `l` field;

The size of the expression result, in bits is given by the `bits` field.

Expressions of type `OCL_ET_EXTEND` are specified using type `octiaAddrExpExtend`:

```
typedef struct octiaAddrExpExtendS {
    Bool          signExtend; ///< is extension signed?
    octiaAddrExpP child     ; ///< child expression
} octiaAddrExpExtend;
```

The `signExtend` field specifies whether the extension is signed and the `child` field specifies the expression to extend.

Expressions of type `OCL_ET_UNARY` are specified using type `octiaAddrExpUnary`:

```
typedef struct octiaAddrExpUnaryS {
    octiaUnop     op    ; ///< unary operation type
    const char *  opName; ///< unary operation name
    octiaAddrExpP child ; ///< unary operation name
} octiaAddrExpUnary;
```

The `opName` field can be used to print the operation name. The `op` field specifies what unary operation is being performed, as defined by the `octiaUnop` enumeration:

```
typedef enum octiaUnopE {
    OCL_UN_MOV   , ///< d = a
    OCL_UN_SWP   , ///< d = byteswap(a)
    OCL_UN_NEG   , ///< d = -a
    OCL_UN_ABS   , ///< d = (a<0) ? -a : a
    OCL_UN_NEGSQ , ///< d = saturate_signed(-a)
    OCL_UN_ABSSQ , ///< d = (a<0) ? saturate_signed(-a) : a
    OCL_UN_NOT   , ///< d = ~a
    OCL_UN_CLZ   , ///< d = count_leading_zeros(a)
    OCL_UN_CLO   , ///< d = count_leading_ones(a)
    OCL_UN_CTZ   , ///< d = count_trailing_zeros(a)
    OCL_UN_CTO   , ///< d = count_trailing_ones(a)
    OCL_UN_BSFZ  , ///< d = least_significant_zero_index(a)
    OCL_UN_BSFO  , ///< d = most_significant_zero_index(a)
    OCL_UN_BSRZ  , ///< d = most_significant_zero_index(a)
    OCL_UN_BSRO  , ///< d = most_significant_one_index(a)
    OCL_UN_LAST    ///< KEEP LAST
} octiaUnop;
```

The `child` field specifies the unary expression operand.

Expressions of type `OCL_ET_BINARY` are specified using type `octiaAddrExpBinary`:

```
typedef struct octiaAddrExpBinaryS {
    octiaBinop    op      ; ///< binary operation type
    const char *  opName  ; ///< binary operation name
    octiaAddrExpP child[2]; ///< children expressions
} octiaAddrExpBinary;
```

The `opName` field can be used to print the operation name. The `op` field specifies what binary operation is being performed, as defined by the `octiaBinop` enumeration:

```
typedef enum octiaBinopE {
    OCL_BIN_ADD     , ///< d = a + b
    OCL_BIN_ADC     , ///< d = a + b + C
    OCL_BIN_SUB     , ///< d = a - b
    OCL_BIN_SBB     , ///< d = a - b - C
    OCL_BIN_RSBB    , ///< d = b - a - C
    OCL_BIN_RSUB    , ///< d = b - a
    OCL_BIN_IMUL    , ///< d = a * b (signed)
    OCL_BIN_MUL     , ///< d = a * b (unsigned)
    OCL_BIN_IDIV    , ///< d = a / b (signed)
    OCL_BIN_DIV     , ///< d = a / b (unsigned)
    OCL_BIN_IREM    , ///< d = a % b (signed)
    OCL_BIN_REM     , ///< d = a % b (unsigned)
    OCL_BIN_CMP     , ///< a - b
    OCL_BIN_ADDSQ   , ///< d = saturate_signed(a + b)
    OCL_BIN_SUBSQ   , ///< d = saturate_signed(a - b)
    OCL_BIN_RSUBSQ  , ///< d = saturate_signed(b - a)
    OCL_BIN_ADDUQ   , ///< d = saturate_unsigned(a + b)
    OCL_BIN_SUBUQ   , ///< d = saturate_unsigned(a - b)
    OCL_BIN_RSUBUQ  , ///< d = saturate_unsigned(b - a)
    OCL_BIN_ADDSH   , ///< d = ((signed)(a + b)) / 2
    OCL_BIN_SUBSH   , ///< d = ((signed)(a - b)) / 2
    OCL_BIN_RSUBSH  , ///< d = ((signed)(b - a)) / 2
    OCL_BIN_ADDUH   , ///< d = ((unsigned)(a + b)) / 2
    OCL_BIN_SUBUH   , ///< d = ((unsigned)(a - b)) / 2
    OCL_BIN_RSUBUH  , ///< d = ((unsigned)(b - a)) / 2
    OCL_BIN_ADDSHR  , ///< d = round(((signed)(a + b)) / 2)
    OCL_BIN_SUBSHR  , ///< d = round(((signed)(a - b)) / 2)
    OCL_BIN_RSUBSHR , ///< d = round(((signed)(b - a)) / 2)
    OCL_BIN_ADDUHR  , ///< d = round(((unsigned)(a + b)) / 2)
    OCL_BIN_SUBUHR  , ///< d = round(((unsigned)(a - b)) / 2)
    OCL_BIN_RSUBUHR , ///< d = round(((unsigned)(b - a)) / 2)
    OCL_BIN_OR      , ///< d = a | b
    OCL_BIN_AND     , ///< d = a & b
    OCL_BIN_XOR     , ///< d = a ^ b
    OCL_BIN_ORN     , ///< d = a | ~b
    OCL_BIN_ANDN    , ///< d = a & ~b
    OCL_BIN_XORN    , ///< d = a ^ ~b
    OCL_BIN_NOR     , ///< d = ~(a | b)
    OCL_BIN_NAND    , ///< d = ~(a & b)
    OCL_BIN_XNOR    , ///< d = ~(a ^ b)
    OCL_BIN_ROL     , ///< d = a << b | a >> [bits]-b
    OCL_BIN_ROR     , ///< d = a >> b | a << [bits]-b
    OCL_BIN_RCL     , ///< (d,c) = (a,c) << b | (a,c) >> [bits]-b
    OCL_BIN_RCR     , ///< (d,c) = (a,c) >> b | (a,c) << [bits]-b
    OCL_BIN_SHL     , ///< d = a << b
    OCL_BIN_SHR     , ///< d = (unsigned)a >> b
    OCL_BIN_SAR     , ///< (signed)a >> b
    OCL_BIN_SHLSQ   , ///< d = saturate_signed(a << b)
    OCL_BIN_SHLUQ   , ///< d = saturate_unsigned(a << b)
    OCL_BIN_SHRR    , ///< d = round((unsigned)a >> b)
    OCL_BIN_SARR    , ///< d = round((signed)a >> b)
    OCL_BIN_LAST      ///< KEEP LAST
} octiaBinop;
```

The `child` field specifies the binary expression operands.

Expressions of type `OCL_ET_LOAD` are specified using type `octiaAddrExpLoad`:

```
typedef struct octiaAddrExpLoadS {
    memDomainP    domain; ///< domain for load
    octiaAddrExpP child ; ///< address expression in domain
} octiaAddrExpLoad;
```

The `domain` field specified the memory domain from which to load an operand value. The `child` field specifies the address expression used for the load.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddrExp(octiaAddrExpP exp, octiaAttrP attrs) {

    if(exp->type==OCL_ET_UNKNOWN) {

        vmiPrintf("UNKNOWN");

    } else if(exp->type==OCL_ET_CONST) {
        vmiPrintf("0x"FMT_Ax, exp->c);

    } else if(exp->type==OCL_ET_REG) {

        vmiRegInfoCP reg = vmiiaConvertRegInfo(exp->r);

        if(reg) {
            vmiPrintf("%s", reg->name);
        } else {
            vmiPrintf("UNKNOWN_REGISTER");
        }

    } else if(exp->type==OCL_ET_EXTEND) {

        vmiPrintf(
            "(%cext-%u-to-%u ",
            exp->e.signExtend?'s':'z',
            exp->e.child->bits,
            exp->bits
        );
        printAddrExp(exp->e.child, 0);
        vmiPrintf(")");

    } else if(exp->type==OCL_ET_UNARY) {

        vmiPrintf("(%s ", exp->u.opName);
        printAddrExp(exp->u.child, 0);
        vmiPrintf(")");

    } else if(exp->type==OCL_ET_BINARY) {

        vmiPrintf("(%s ", exp->b.opName);
        printAddrExp(exp->b.child[0], 0);
        vmiPrintf(", ");
        printAddrExp(exp->b.child[1], 0);
        vmiPrintf(")");

    } else if(exp->type==OCL_ET_LOAD) {

        vmiPrintf("[");
        printAddrExp(exp->l.child, 0);
        vmiPrintf("]");

    } else {

        vmiPrintf("{unexpected expression type %u}", exp->type);
    }

    // print evaluated expression unless it is a constant
    if(attrs && (exp->type!=OCL_ET_CONST)) {

        Uns32 bytes = BITS_TO_BYTES(exp->bits);
        Uns8  result[bytes];

        // evaluate the expression
        if(ocliaEvaluate(attrs, exp, result)) {

            Int32 i;

            // print result
            vmiPrintf(" {current value:0x");
            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", result[i]);
            }
            vmiPrintf("}");

        } else {

            vmiPrintf(" {current value UNKNOWN}");
        }
    }
```

```
}

static void printAddressExpressionList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        vmiPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            vmiPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            vmiPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(depend));
                vmiPrintf("    depend %s\n", reg->name);
            }
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.22 ocliaGetMemAccessDomain

Prototype

```
memDomainP ocliaGetMemAccessDomain (
    octiaMemAccessP ma
);
```

Description

Given an instruction attributes memory access object, this function returns the *memory domain* for that memory access. This could be used in the intercept library to replicate the load or store.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.23 ocliaGetMemAccessFirstDepend

Prototype

```
octiaRegListP ocliaGetMemAccessFirstDepend (
    octiaMemAccessP ma
);
```

Description

Given an instruction attributes memory access object, this function returns the *first register dependency* for that memory access. Register dependencies are the registers whose values must be known in order to perform the memory access.

The result is an object of type `octiaRegListP`. See section 31.15 for more information about use of structures of this type.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddressExpressionList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        vmiPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            vmiPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            vmiPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(depend));
                vmiPrintf("    depend %s\n", reg->name);
            }
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.24 ocliaGetMemAccessMemBits

Prototype

```
Uns32 ocliaGetMemAccessMemBits (
    octiaMemAccessP ma
);
```

Description

Given an instruction attributes memory access object, this function returns the size on bits of the memory access (so, for example, a value of 8 implies a single-byte access).

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddressExpressionList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaMemAccessP ma;

    // memory access type strings
    static const char *memAccessTypeString[] = {
        [OCL_MAT_LOAD]       = "load",
        [OCL_MAT_STORE]      = "store",
        [OCL_MAT_PRELOAD_LD] = "preload-for-load",
        [OCL_MAT_PRELOAD_ST] = "preload-for-store",
        [OCL_MAT_PRELOAD_EX] = "preload-for-fetch"
    };

    if((ma=ocliaGetFirstMemAccess(attrs))) {

        vmiPrintf("\n");

        for(; ma; ma=ocliaGetNextMemAccess(ma)) {

            octiaAddrExpP addrExp = ocliaGetMemAccessAddrExp(ma);
            octiaRegListP depend;

            // print characteristics of memory access
            vmiPrintf(
                "  %u-bit %s address (bits %u): ",
                ocliaGetMemAccessMemBits(ma),
                memAccessTypeString[ocliaGetMemAccessType(ma)],
                addrExp->bits
            );

            // print load/store address expression
            printAddrExp(addrExp, attrs);
            vmiPrintf("\n");

            // print all dependencies of this load/store (registers that must
            // be known before it can be evaluated)
            for(
                depend=ocliaGetMemAccessFirstDepend(ma);
                depend;
                depend=ocliaGetRegListNext(depend)
            ) {
                vmiRegInfoCP reg = vmiiaConvertRegInfo(ocliaGetRegListReg(depend));
                vmiPrintf("    depend %s\n", reg->name);
            }
        }
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Thread Safe

31.25 ocliaSetMemAccessUserData, ocliaGetMemAccessUserData

Prototypes

```
void ocliaSetMemAccessUserData (
    octiaMemAccessP ma,
    void *          userData
);

void * ocliaGetMemAccessUserData (
    octiaMemAccessP ma
);
```

Description

Given an instruction attributes memory access object, these functions assign and retrieve client-specific data associated with that object, respectively. This allows instruction attribute information to be extended if required.

Client code is responsible for memory management of the client data if required.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.26 ocliaGetFirstNextPC

Prototype

```
octiaNextPCP ocliaGetFirstNextPC (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first *next PC* record for that instruction. Such records list every potential address at which subsequent evaluation could occur after the current instruction, and can be further queried by functions `ocliaGetNextPCAddrExp`, `ocliaGetNextPCDS`, `ocliaGetNextPCHint` and `ocliaGetNextPCOffset`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printNextPCList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        vmiPrintf("\n");

        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            vmiJumpHint   hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            vmiPrintf("  next PC ");

            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                vmiPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&vmi_JH_RELATIVE)) {
                vmiPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                vmiPrintf(" (DS)");
            }

            vmiPrintf("\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.27 ocliaGetNextNextPC

Prototype

```
octiaNextPCP ocliaGetNextNextPC (
    octiaNextPCP prev
);
```

Description

Given an instruction attributes next PC object, this function returns the *next* next PC object for that instruction. Such records list every potential address at which subsequent evaluation could occur after the current instruction, and can be further queried by functions `ocliaGetNextPCAddrExp`, `ocliaGetNextPCDS`, `ocliaGetNextPCHint` and `ocliaGetNextPCOffset`.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printNextPCList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        vmiPrintf("\n");

        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            vmiJumpHint   hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            vmiPrintf("  next PC ");

            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                vmiPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&vmi_JH_RELATIVE)) {
                vmiPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                vmiPrintf(" (DS)");
            }

            vmiPrintf("\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.28 ocliaGetNextPCAddrExp

Prototype

```
octiaAddrExpP ocliaGetNextPCAddrExp (
    octiaNextPCP entry
);
```

Description

Given an instruction attributes next PC object, this function returns the *address expression* for that next PC object. See section 31.21 for a detailed description of address expressions.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printNextPCList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        vmiPrintf("\n");

        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            vmiJumpHint   hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            vmiPrintf("  next PC ");

            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                vmiPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&vmi_JH_RELATIVE)) {
                vmiPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                vmiPrintf(" (DS)");
            }

            vmiPrintf("\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.29 ocliaGetNextPCDS

Prototype

```
Bool ocliaGetNextPCDS (
    octiaNextPCP entry
);
```

Description

Given an instruction attributes next PC object, this function returns a Boolean indicating whether the next PC will be branched to after a delay slot instruction has been executed.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printNextPCList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        vmiPrintf("\n");

        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            vmiJumpHint   hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            vmiPrintf("  next PC ");

            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                vmiPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&vmi_JH_RELATIVE)) {
                vmiPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                vmiPrintf(" (DS)");
            }

            vmiPrintf("\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.30 ocliaGetNextPCHint

Prototype

```
octiaJumpHint ocliaGetNextPCHint (
    octiaNextPCP entry
);
```

Description

Given an instruction attributes next PC object, this function returns any hint associated with the address. The hint is described by the `octiaJumpHint` bit mask enumeration, which conforms to the `vmiJumpHint` enumeration:

```
typedef enum octiaJumpHintE {
    OCL_JH_NONE     = 0x00, ///< no jump hint
    OCL_JH_CALL     = 0x01, ///< call
    OCL_JH_RETURN   = 0x02, ///< return
    OCL_JH_INT      = 0x04, ///< interrupt
    OCL_JH_RELATIVE = 0x08  ///< target is relative
} octiaJumpHint;
```

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static const char *getHintText(vmiJumpHint hint) {

    if(hint==vmi_JH_NONE) {

        return 0;

    } else {

        static char buffer[32];

        sprintf(
            buffer, "hint:%s%s%s%s",
            (hint&vmi_JH_CALL)     ? "call"        : "",
            (hint&vmi_JH_RETURN)   ? "return"      : "",
            (hint&vmi_JH_INT)      ? "int"         : "",
            (hint&vmi_JH_RELATIVE) ? " (relative)" : ""
        );

        return buffer;
    }
}

static void printNextPCList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        vmiPrintf("\n");

        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            vmiJumpHint   hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            vmiPrintf("  next PC ");
            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                vmiPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&vmi_JH_RELATIVE)) {
                vmiPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                vmiPrintf(" (DS)");
            }

            vmiPrintf("\n");
        }
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Thread Safe

31.31 ocliaGetNextPCOffset

Prototype

```
Offset ocliaGetNextPCOffset (
    octiaNextPCP entry
);
```

Description

Given an instruction attributes next PC object encoding a *relative* branch (i.e., which was created with the `vmi_JH_RELATIVE` hint), this function returns the target address offset from the current PC. If the jump is not relative, 0 is returned.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printNextPCList(octiaAttrP attrs, vmiProcessorP processor) {

    octiaNextPCP nextPC;

    if((nextPC=ocliaGetFirstNextPC(attrs))) {

        vmiPrintf("\n");

        for(; nextPC; nextPC=ocliaGetNextNextPC(nextPC)) {

            vmiJumpHint   hint      = ocliaGetNextPCHint(nextPC);
            const char   *hintText  = getHintText(ocliaGetNextPCHint(nextPC));
            octiaAddrExpP nextPCExp = ocliaGetNextPCAddrExp(nextPC);

            vmiPrintf("  next PC ");

            printAddrExp(nextPCExp, attrs);

            if(hintText) {
                vmiPrintf(" %s", hintText);
            }
            if((nextPCExp->type==OCL_ET_CONST) && (hint&vmi_JH_RELATIVE)) {
                vmiPrintf(" offset "FMT_Ad, ocliaGetNextPCOffset(nextPC));
            }
            if(ocliaGetNextPCDS(nextPC)) {
                vmiPrintf(" (DS)");
            }

            vmiPrintf("\n");
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.32 ocliaSetNextPCUserData, ocliaGetNextPCUserData

Prototypes

```
void ocliaSetNextPCUserData (
    octiaNextPCP entry,
    void *       userData
);

void * ocliaGetNextPCUserData (
    octiaNextPCP entry
);
```

Description

Given an instruction attributes next PC object, these functions assign and retrieve client-specific data associated with that object, respectively. This allows instruction attribute information to be extended if required.

Client code is responsible for memory management of the client data if required.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.33 ocliaGetFirstReadRange

Prototype

```
octiaRawRangeP ocliaGetFirstReadRange (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first *read range* list record for that instruction. Such records list every section of the processor structure that has been accessed using a `vmiReg` description but which is not associated with a register described using a `vmiRegInfoCP` structure in the debug interface, and can be further queried using functions `ocliaGetRangeLow` and `ocliaGetRangeHigh`.

Any such ranges correspond to processor state that has not been exposed via the register interface, implying that either the register interface or the instruction attributes support for the processor is incomplete. See the *VMI Morph Time Function Reference Manual* for information about how instruction attributes support should be enhanced in a processor model to provide completeness in cases where instruction attributes information cannot be automatically derived.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRangeList(octiaRawRangeP range, const char *type) {

    if(range) {

        vmiPrintf("\n");

        for(; range; range=ocliaGetRangeNext(range)) {
            vmiPrintf(
                "  %5s "FMT_PTRd":"FMT_PTRd"\n",
                type,
                ocliaGetRangeLow(range),
                ocliaGetRangeHigh(range)
            );
        }
    }
}

static void walkAttrs(vmiProcessorP processor, octiaAttrP attrs) {

    printClass(attrs);
    printFetchRecords(attrs);

    printRegList(ocliaGetFirstReadReg(attrs), "read");
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");

    printNextPCList(attrs, processor);
    printAddressExpressionList(attrs, processor);

    vmiPrintf("\n");
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Thread Safe

31.34 ocliaGetFirstWrittenRange

Prototype

```
octiaRawRangeP ocliaGetFirstWrittenRange (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the first *written range* list record for that instruction. Such records list every section of the processor structure that has been assigned using a `vmiReg` description but which is not associated with a register described using a `vmiRegInfoCP` structure in the debug interface, and can be further queried using functions `ocliaGetRangeLow` and `ocliaGetRangeHigh`.

Any such ranges correspond to processor state that has not been exposed via the register interface, implying that either the register interface or the instruction attributes support for the processor is incomplete. See the *VMI Morph Time Function Reference Manual* for information about how instruction attributes support should be enhanced in a processor model to provide completeness in cases where instruction attributes information cannot be automatically derived.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRangeList(octiaRawRangeP range, const char *type) {

    if(range) {

        vmiPrintf("\n");

        for(; range; range=ocliaGetRangeNext(range)) {
            vmiPrintf(
                "  %5s "FMT_PTRd":"FMT_PTRd"\n",
                type,
                ocliaGetRangeLow(range),
                ocliaGetRangeHigh(range)
            );
        }
    }
}

static void walkAttrs(vmiProcessorP processor, octiaAttrP attrs) {

    printClass(attrs);
    printFetchRecords(attrs);

    printRegList(ocliaGetFirstReadReg(attrs), "read");
    printRangeList(ocliaGetFirstReadRange(attrs), "read");
    printRegList(ocliaGetFirstWrittenReg(attrs), "write");
    printRangeList(ocliaGetFirstWrittenRange(attrs), "write");

    printNextPCList(attrs, processor);
    printAddressExpressionList(attrs, processor);

    vmiPrintf("\n");
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Thread Safe

31.35 ocliaGetRangeNext

Prototype

```
octiaRawRangeP ocliaGetRangeNext (
    octiaRawRangeP prev
);
```

Description

Given an instruction attributes range list object, this function returns the *next* object in that list.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRangeList(octiaRawRangeP range, const char *type) {

    if(range) {

        vmiPrintf("\n");

        for(; range; range=ocliaGetRangeNext(range)) {
            vmiPrintf(
                "  %5s "FMT_PTRd":"FMT_PTRd"\n",
                type,
                ocliaGetRangeLow(range),
                ocliaGetRangeHigh(range)
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.36 ocliaGetRangeLow

Prototype

```
IntPS ocliaGetRangeLow (
    octiaRawRangeP range
);
```

Description

Given an instruction attributes range list object, this function returns the *low* bound of that range, expressed as a byte offset from the start of the processor object.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRangeList(octiaRawRangeP range, const char *type) {

    if(range) {

        vmiPrintf("\n");

        for(; range; range=ocliaGetRangeNext(range)) {
            vmiPrintf(
                "  %5s "FMT_PTRd":"FMT_PTRd"\n",
                type,
                ocliaGetRangeLow(range),
                ocliaGetRangeHigh(range)
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.37 ocliaGetRangeHigh

Prototype

```
IntPS ocliaGetRangeHigh (
    octiaRawRangeP range
);
```

Description

Given an instruction attributes range list object, this function returns the *high* bound of that range, expressed as a byte offset from the start of the processor object.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printRangeList(octiaRawRangeP range, const char *type) {

    if(range) {

        vmiPrintf("\n");

        for(; range; range=ocliaGetRangeNext(range)) {
            vmiPrintf(
                "  %5s "FMT_PTRd":"FMT_PTRd"\n",
                type,
                ocliaGetRangeLow(range),
                ocliaGetRangeHigh(range)
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.38 ocliaGetInstructionCondition

Prototype

```
Uns32 ocliaGetInstructionCondition (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns any *condition* associated with the instruction. A return value of 0 is special and indicates the instruction is unconditional; other values are model-specific. The returned condition may be evaluated using the current processor state using `vmirtEvaluateCondition`. See function `vmimtSetInstructionCondition` in the *VMI Morph Time Function Reference* manual for information about how models can associate conditions with instructions and implement condition evaluation.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printCondition(octiaAttrP attrs) {

    Uns32 condition = ocliaGetInstructionCondition(attrs);

    if(condition) {
        vmiPrintf(
            "\n  condition %u (%c)\n",
            condition,
            ocliaEvaluateInstructionCondition(attrs) ? 'T' : 'F'
        );
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.39 ocliaEvaluateInstructionCondition

Prototype

```
Bool ocliaEvaluateInstructionCondition (
    octiaAttrP attrs
);
```

Description

Given an instruction attributes object, this function returns the current value of any model-specific *condition* associated with that instruction. If the instruction is unconditional, the function returns `True`. See function `vmimtSetInstructionCondition` in the *VMI Morph Time Function Reference* manual for information about how models can associate conditions with instructions and implement condition evaluation.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printCondition(octiaAttrP attrs) {

    Uns32 condition = ocliaGetInstructionCondition(attrs);

    if(condition) {
        vmiPrintf(
            "\n  condition %u (%c)\n",
            condition,
            ocliaEvaluateInstructionCondition(attrs) ? 'T' : 'F'
        );
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread Safe

31.40 ocliaEvaluate

Prototype

```
Bool ocliaEvaluate (
    octiaAttrP    attrs,
    octiaAddrExpP exp,
    void*         result
);
```

Description

Given an instruction attributes object and an expression object, this function evaluates the expression using the current state of the processor associated with the instruction attributes object. The result is assigned to the `result` buffer, and the Boolean return value indicates whether the evaluation was successful.

It is the client's responsibility to ensure that the result buffer is large enough to hold the result. This can be done by declaring an `Uns8`-valued array of size:

```
BITS_TO_BYTES(exp->bits)
```

as in the example below.

Example

The following template shows how this function could be used as part of an intercept library that reports instruction attributes:

```
static void printAddrExp(octiaAddrExpP exp, octiaAttrP attrs) {

    if(exp->type==OCL_ET_UNKNOWN) {

        vmiPrintf("UNKNOWN");

    } else if(exp->type==OCL_ET_CONST) {

        vmiPrintf("0x"FMT_Ax, exp->c);

    } else if(exp->type==OCL_ET_REG) {

        vmiRegInfoCP reg = vmiiaConvertRegInfo(exp->r);

        if(reg) {
            vmiPrintf("%s", reg->name);
        } else {
            vmiPrintf("UNKNOWN_REGISTER");
        }

    } else if(exp->type==OCL_ET_EXTEND) {

        vmiPrintf(
            "(%cext-%u-to-%u ",
            exp->e.signExtend?'s':'z',
            exp->e.child->bits,
            exp->bits
        );
        printAddrExp(exp->e.child, 0);
        vmiPrintf(")");

    } else if(exp->type==OCL_ET_UNARY) {
        vmiPrintf("(%s ", exp->u.opName);
        printAddrExp(exp->u.child, 0);
        vmiPrintf(")");

    } else if(exp->type==OCL_ET_BINARY) {

        vmiPrintf("(%s ", exp->b.opName);
        printAddrExp(exp->b.child[0], 0);
        vmiPrintf(", ");
        printAddrExp(exp->b.child[1], 0);
        vmiPrintf(")");

    } else if(exp->type==OCL_ET_LOAD) {

        vmiPrintf("[");
        printAddrExp(exp->l.child, 0);
        vmiPrintf("]");

    } else {

        vmiPrintf("{unexpected expression type %u}", exp->type);
    }

    // print evaluated expression unless it is a constant
    if(attrs && (exp->type!=OCL_ET_CONST)) {

        Uns32 bytes = BITS_TO_BYTES(exp->bits);
        Uns8  result[bytes];

        // evaluate the expression
        if(ocliaEvaluate(attrs, exp, result)) {

            Int32 i;

            // print result
            vmiPrintf(" {current value:0x");
            for(i=bytes-1; i>=0; i--) {
                vmiPrintf("%02x", result[i]);
            }
            vmiPrintf("}");

        } else {

            vmiPrintf(" {current value UNKNOWN}");
        }
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Thread Safe

32 Miscellaneous Functions

This section describes miscellaneous functions that do not fall into previously-described categories.

32.1 vmirtCacheRegister

Prototype

```
const char *vmirtCacheRegister( \
    vmiProcessorP        processor,
    vmiCacheType         cacheType,
    Bool                 enable,
    vmiCacheHitFn        hitCB,
    vmiCacheMissFn       missCB,
    vmiCacheInvalidateFn invalidateCB,
    void                *userData
);
```

Description

On processor models that implement a cache model (such as the OVP MIPS model) this function may be used to enable/disable the cache model and register call back functions for cache events.

When enable is `True` the cache is enabled; when `False` it is disabled.

The types `vmiCacheType`, `vmiCacheHitFn`, `vmiCacheMissFn` and `vmiCacheInvalidateFn` are defined in `vmiCacheAttrs.h`:

```
typedef enum vmiCacheTypeE {
    VMI_CT_I = 0,  // level 1 instruction
    VMI_CT_D = 1,  // level 1 data
    VMI_CT_LAST    // KEEP LAST - May be used for array size
} vmiCacheType, *vmiCacheTypeP;

#define VMI_CACHE_HIT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,      \
    Uns32          rowIndex,       \
    Addr           hitAddr,        \
    void          *userData        \
)
typedef VMI_CACHE_HIT_FN((*vmiCacheHitFn));

#define VMI_CACHE_MISS_FN(_NAME) void *_NAME( \
    vmiProcessorP  processor,      \
    Uns32          rowIndex,       \
    Addr           missAddr,       \
    Addr           victimAddr,     \
    void          *cacheContext,   \
    void          *userData        \
)
typedef VMI_CACHE_MISS_FN((*vmiCacheMissFn));

#define VMI_CACHE_INVALIDATE_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,      \
    Uns32          rowIndex,       \
    Addr           invalidateAddr, \
    Addr           victimAddr,     \
    void          *cacheContext,   \
    void          *userData        \
)
typedef VMI_CACHE_INVALIDATE_FN((*vmiCacheInvalidateFn));
```

`cacheType` is used to specify either the instruction or data cache. (Only level 1 caches are supported)

The `hitCB`, `missCB` and `invalidateCB` function pointers, if not `NULL`, specify the function to be called when that event occurs in the cache.

The `vmiCacheMissFn` returns a `void *` value that will be associated with the cache line that is added as a result of the miss. If that line is evicted this value will be passed as the `cacheContext` value on the miss or invalidate callback. This can be useful for tracking the context (e.g. process and/or function) that was active when a line was added to the cache.

When called the functions are passed the following arguments:

- `processor`: The processor making the access that caused the event
- `rowIndex`: The row of the cache being accessed
- `hitAddr`: The address accessed on a cache hit
- `missAddr`: The address accessed on a cache miss
- `invalidateAddr`: The address accessed on a cache invalidate
- `victimAddr`: The address accessed when victim cache line was added
- `cacheContext`: Value returned by `cacheMiss` when the victim cache line was added
- `userData`: The value provided as the last argument to `vmirtCacheRegister`

`vmirtCacheRegister` returns a unique name for the cache if successful. If the same cache is connected to multiple processors they will each get the same name returned.

If the call is not successful `NULL` will be returned. This may occur if the processor model does not support a cache model.

Example

This example is from an intercept library compatible with the OVP MIPS processor model:

```
static VMI_CACHE_HIT_FN(notifyHit) {
    ...
}
static VMI_CACHE_MISS_FN(notifyMiss) {
    ...
}
static VMI_CACHE_INVALIDATE_FN(notifyInvalidate) {
    ...
}

const char *mipsCacheEnable(
    vmiosObjectP  object,
    vmiCacheType  type,
    Bool          enable)
{

    VMI_ASSERT(type < VMI_CT_LAST, "Illegal Cache type");

    const char *cacheName =
        vmirtCacheRegister(
            object->processor,
            type,
            enable,
            notifyHit,
            notifyMiss,
            notifyInvalidate,
            cache
        );

    if (!cacheName) {

        // Cache not supported
        vmiMessage (
            "I", PLUGIN_PREFIX,
            "%s : unable to register %s\n",
            vmirtProcessorName(object->processor),
            vmiCacheTypeName(type)
        );
   }

   Return cacheName;
}
```

Notes and Restrictions

1. When a cache is shared between multiple processors (e.g. in a multi-threaded processor such as the MIPS 34K) callbacks must be registered separately on every processor connected to the cache.

QuantumLeap Semantics

Non-self-synchronizing

32.2 vmirtEvaluateCondition

Prototype

```
Bool vmirtEvaluateCondition(vmiProcessorP processor, 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.

The Instruction Attributes API described in chapter 31 allows the condition associated with an instruction to be found using function `ocliaGetInstructionCondition`. The current value of the condition can then be found by calling function `vmirtEvaluateCondition`. A common use for this function is in an intercept library used for timing analysis; in this case, the time required to execute an instruction may depend on whether the condition is satisfied or not. See the *VMI Morph Time Function Reference* manual for more information about how conditions are specified and how support for condition evaluation is implemented in a processor model.

Example

This example is from an intercept library compatible with the OVP ARM processor model. Depending on the current value of a condition, it emits code to add either `cyclesT` or `cyclesF` to a register that accumulates a count of processor cycles executed:

```
void ceAddCycleCountCondCC(
    vmiosObjectP object,
    Uns32        cyclesT,
    Uns32        cyclesF,
    Uns32        condCode
) {
    vmiReg flag      = vmimtGetExtReg(object->processor, &object->flag);
    vmiReg tmpCycles = vmimtGetExtReg(object->processor, &object->tmpCycles);

    // get required result in tmpCycles
    vmimtArgProcessor();
    vmimtArgUns32(condCode);
    vmimtCallResult((vmiCallFn)vmirtEvaluateCondition, 8, flag);
    vmimtCondMoveRCC(32, flag, True, tmpCycles, cyclesT, cyclesF);

    // call internal interface
    addCycleCountRInt(object);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing