# Getting Started with MicroTESK’s MMU Plugin

## Introduction

A memory management unit (MMU) is the hardware that handles all accesses to the main memory. It performs virtual address translation, memory protection, caching, and some other functions. The MMU is one of the most sophisticated parts of the microprocessor design. Thus, it takes a lot of effort to verify it carefully, examining all possible execution scenarios. Fortunately, MicroTESK provides powerful facilities to automate test program generation for MMUs. They are implemented in MMU Plugin, a special extension responsible for memory subsystem specification and verification.

In this document, we will consider a simplistic microprocessor and show how to specify its MMU and how to generate test programs based on the specifications developed.

## Specification

Let us consider a very simple computer architecture. We call it Vmem, because it is used to illustrate MicroTESK facilities on virtual memory specification and verification.

### Instructions Specification

Vmem is a 16-bit microprocessor with 16 general-purpose registers (GPRs). The instruction set includes but not limited to the following instructions:

* ml rt, im — Move the 8-bit immediate value im into the lower bits of GPR rt.
* mh rt, im — Move the 8-bit immediate value im into the higher bits of GPR rt.
* ld rt, addr — Load the 16-bit data from the memory location into GPR rt. The memory location is identified by the virtual address specified in GPR addr.
* st rs, addr — Store the content of GPR rs into the memory location. The memory location is identified by the virtual address specified in GPR addr.

Our main interest focuses on the ld and st instructions; ml and mh are used to initialize registers with data and addresses.

Let us consider briefly how the Vmem registers and instructions are specified in nML (for more information on nML, see “nML Language Reference”).

Take a look at $MICROTESK\_HOME/arch/demo/vmem/model/vmem.nml.

#### Data Types, Registers, and Memories

First of all, the specifications contain data type definitions.

type BYTE = card(8)

type HWORD = card(16)

type INDEX = card(4)

Then, there are GPRs, physical memory, and a program counter (PC), which is a special register that holds the virtual address (VA) of the current instruction. As you can see, the physical memory consists of 214 bytes (or, equally, of 213 half words). It is addressed by 14-bit physical addresses (PAs), while VAs are of 16-bit length. Virtual address translation will be described later on.

reg GPR[16, HWORD]

mem MEM[2 \*\* 13, HWORD]

reg PC[HWORD]

#### Addressing Modes and Operations

The next section introduces an addressing mode used to access GPRs. The definition is straightforward. The only thing to be noticed is that the addressing mode is supplied with the syntax and image attributes: the former defines the assembly format, while the latter defines the binary encoding.

mode REG(i: INDEX) = GPR[i]

syntax = format("$%d", i)

image = format("%s", i)

Then, there are specified the instructions (operations). Here are descriptions of ml and ld (descriptions of mh and st look similar). You can see such attributes as syntax, image, and action. The latter defines the instruction semantics.

op ml(rt: REG, im: BYTE)

syntax = format("ml %s %x", rt.syntax, im)

image = format("0001%s%s", rt.image, im)

action = {

rt<7..0> = im;

}

Looking at the ld action, you might notice that the instruction raises the AddressError exception if the address is not aligned to a half-word boundary. Another interesting observation is that the MEM array, which represents the physical memory, is indexed by VAs, not PAs. It is not a bug; every access to MEM triggers virtual address translation (and other MMU functionality).

op ld(rt: REG, addr: REG)

syntax = format("ld %s %s", rt.syntax, addr.syntax)

image = format("10010000%s%s", rt.image, addr.image)

action = {

if (addr<0> != 0) then

exception("AddressError");

endif;

rt = MEM[addr >> 1];

}

Finally, the specifications define the root operation. In a sense, this is an entry point (like main in C programming language). The root operation, called instruction, encapsulates common logic for all instructions. For example, it may modify PC.

op command = nop | ml | mh | ld | st

op instruction(c: command)

syntax = c.syntax

image = c.image

action = {

c.action;

PC = PC + 2;

}

Compiling the nML specifications is done as follows:

cp $MICROTESK\_HOME/arch/demo/vmem/model

sh $MICROTESK\_HOME/bin/compile.sh vmem.nml

That is all about instructions. Now it is time to dive into VA-to-PA translation, caching, and all that stuff.

### MMU Specification

Let us specify Vmem’s MMU step by step.

The result will be accumulated in $MICROTESK\_HOME/arch/demo/vmem/model/vmem.mmu.

As you may have noticed, the file extension is .mmu, not .nml. That is right. We have designed a special language, called mmuSL, to specify MMUs.

#### MMU, Version 1

The first thing we need to do is to define data types for VAs and PAs. In Vmem, the physical memory is addressed by 14-bit PAs, while VAs are of 16-bit length.

address VA(value: 16)

address PA(value: 14)

In general, address is a structure with multiple fields (e.g., the address itself, the access type, the caching policy, etc.). By agreement, the first field stores the address value. However, you can change the default interpretation by specifying the address field explicitly.

address VA(…, value: 16, …): value

Another essential thing is the physical memory. The main memory is specified as a PA-addressable buffer with the same name as the memory array in the nML specification.

buffer MEM (pa: PA)

ways = 1

sets = 2 \*\* 13

entry = (data: 16)

index = pa.value<13..1>

match = 0

policy = NONE

Every buffer is assumed to be way-associative and is characterized with six attributes:

* ways — The number of ways in the buffer.
* sets — The number of sets in the buffer.
* entry — The entry format.
* index — The set index calculation function.
* match — The match predicate.
* policy — The data eviction policy.

Let us look at the MEM buffer.

* ways = 1 — The buffer is direct-mapped.
* sets = 2 \*\* 13 — The buffer consists of 8192 entries.
* entry = (data: 16) — Each entry includes nothing more than two-byte data.
* index = pa.value<13..1> — To access the entry, the lower bit of PA should be ignored.

The match and policy attributes will be explained later on.

Now we are ready to describe the MMU. The overall structure is as follows.

mmu vmem (va: VA) = (data: 16)

read = {…}

write = {…}

There are two actions: read and write. The read action takes a VA as an input and returns data from the corresponding memory location. The write action takes a VA and data and stores the data in the corresponding memory location.

Let us write the simplest specification of Vmem’s MMU. The MEM buffer is accessed via PAs. We need to do VA-to-PA translation. The easiest way to do it is to truncate VAs to 14 bits.

pa.value = va.value<13..0>;

Reading from the memory and writing into the memory are carried out as follows.

memEntry = MEM(pa);

MEM(pa) = memEntry;

Here, memEntry is a variable of the MEM.entry type.

Combining all together, the specification will be as follows.

mmu vmem (va: VA) = (data: 16)

var pa: PA;

var memEntry: MEM.entry;

read = {

pa.value = va.value<13..0>;

memEntry = MEM(pa);

data = memEntry.data;

}

write = {

pa.value = va.value<13..0>;

memEntry.data = data;

MEM(pa) = memEntry;

}

Compiling the nML specifications together with the MMU specifications is done as follows:

cp $MICROTESK\_HOME/arch/demo/vmem/model

sh $(MICROTESK\_HOME)/bin/compile.sh vmem.nml vmem.mmu

#### MMU, Version 2

Direct VA-to-PA translation is not exactly what is usually meant by “address translation”. Typically, address translation utilizes a page table, which is a special place in memory, where OS stores VA-to-PA mappings. Each mapping is known as a page table entry (PTE).

What we are going to do is to specify the page table. Before doing this, let us put the address translation logic into a separate function.

function TranslateAddress(va: VA): PA

var pa: PA;

{

pa.value = va.value<13..0>;

return pa;

}

Now we can replace the line pa.value = va.value<13..0> with pa = TranslateAddress(va) in the read and write actions. Having all address translation logic in one place allows not duplicating the code. For example, we can add the following code to the TranslateAddress function.

if (va.value<0> != 0) then

exception("AddressError");

endif;

It is just an illustration. The nML specifications provide MMU specifications with aligned addresses (address alignment checks should be done in nML).

#### MMU, Version 3

The address translation process is implied to be as follows. The two higher bits of VA identifies what to do with the address:

1. 00 — Perform the page table based translation (see “MMU, Version 4”).
2. 01 — Raise the AddressError exception.
3. 10 — Raise the AddressError exception.
4. 11 — Use the lower 14 bits of the VA (this section).

We have two segments: [0x0000, 0x3fff] (item 1) and [0xc000, 0xffff] (item 4). The first one corresponds to page table based translation (MAPPED); the second one corresponds to direct translation (DIRECT). In this section, we will specify the DIRECT segment and the related address translation function.

segment DIRECT (va: VA) = (pa: PA)

range = (0xc000, 0xffff)

read = {

pa.value = va.value<13..0>;

}

The definition is straightforward. There are two attributes: range, which specifies the address range, and read, which specifies the address translation function.

Now, the TranslateAddress function can be modified as follows.

function TranslateAddress(va: VA): PA

var pa: PA;

{

if (va.value<0> != 0) then

exception("AddressError");

endif;

if (DIRECT(va).hit) then

pa = DIRECT(va);

else

exception("AddressError");

endif;

return pa;

}

The construct DIRECT(va).hit checks that the address va hits the segment range. DIRECT(va) launches the address translation.

#### MMU, Version 4

To specify the MAPPED segment, we need a page table. Let us assume that virtual page numbers (VPNs) and physical frame numbers (PFNs) are encoded with 6 bits. It implies that VA is interpreted as follows:

* VA<15..14> = 00.
* VA<13..8> = VPN.
* VA<7..0> = offset.

VA-to-PA translation is done by replacing the VPN with the related PFN located in the page table. The page table is a memory-mapped buffer addressed by VAs. Each entry contains a 6-bit VPN and a 6-bit PFN. Vmem’s page table consists of 64 entries. Given a VA, the VPN is extracted and used as an index to access the page table.

memory buffer PageTable(va: VA)

ways = 1

sets = 2 \*\* 6

entry = (vpn: 6, pfn: 6, unused: 4)

index = va.value<13..8>

match = va.value<13..8> == vpn

policy = NONE

Do not pay much attention to the index, match, and policy attributes. They are ignored for memory-mapped buffers. Look at entry. You can see a special field, called unused. The thing is that the entry size of memory mapped buffers should be equal to the entry size of the physical memory. In Vmem, the size is 16 bits; thus, 4 more bits are required. It worth mentioning how entries’ fields are stored in memory. Lower bits hold right fields; upper bits hold left fields.

| vpn | pfn | unused |

| 15..10 | 9 ..4 | 3..0 |

Here comes the MAPPED segment specification.

segment MAPPED (va: VA) = (pa: PA)

range = (0x0000, 0x3fff)

var pteAddr: VA;

var pteData: PageTable.entry;

read = {

pteAddr.value = 0;

pteAddr.value<15..14> = 0b11;

pteAddr.value<6..1> = va.value<13..8>;

pteData = PageTable(pteAddr);

if (pteData.vpn != va.value<13..8>) then

exception("AddressError");

endif;

pa.value<13..8> = pteData.pfn;

pa.value<7..0> = va.value<7..0>;

}

The most interesting thing is how the page table is accessed. Being a memory-mapped buffer, the page table is addressed by VAs. To avoid recursion, the address, which is the page table base plus the doubled VPN, is made unmapped by setting two higher bits. If the PTE’s VPN differs from the given one, the AddressError exception is raised. Translation is performed as it is described above.

#### MMU, Version 5

To accelerate address translation, microprocessors use translation lookaside buffers (TLBs), special devices that cache recently used PTEs. In different architectures, TLBs are organized in different ways. Some of them, e.g. MIPS, utilize programmable TLBs; others utilize program-invisible ones. Let us add a transparent TLB to our specification.

buffer TLB(va: VA)

ways = 4

sets = 1

entry = (vpn: 6, pfn: 6)

index = 0

match = va.value<13..8> == vpn

policy = FIFO

This buffer is fully associative (there is one set) and uses the FIFO strategy to evict old VPN-to-PFN mappings. Adding the TLB will change the address translation function as follows.

segment MAPPED (va: VA) = (pa: PA)

…

var tlbEntry: TLB.entry;

…

read = {

if (TLB(va).hit) then

tlbEntry = TLB(va);

else

…

tlbEntry.vpn = pteData.vpn;

tlbEntry.pfn = pteData.pfn;

TLB(va) = tlbEntry;

endif;

pa.value<13..8> = tlbEntry.pfn;

pa.value<7..0> = va.value<7..0>;

}

A more complicated example with a programmable TLB and two transparent micro TLBs (for data and instructions) can be found in the MicroTESK for MIPS project.

#### MMU, Version 6

Similarly, we can add caches to the MMU specifications. Does it make any sense to specify program-invisible devices? Indeed, specifying transparent buffers is redundant for expressing ISA. However, it allows test generators to create such situation as cache hit and misses and thus to achieve a better test coverage.

An example below describes L1, a set associative data cache with 4 ways and 2 sets.

buffer L1 (pa : PA)

ways = 4

sets = 2

entry = (tag: 12, data: 16)

index = pa.value<1>

match = pa.value<13..2> == tag

policy = PLRU

Here is how the cache is used in the MMU specification.

mmu vmem (va: VA) = (data: 16)

…

var l1Entry: L1.entry;

var memEntry: MEM.entry;

read = {

…

if (L1(pa).hit) then

l1Entry = L1(pa);

else

memEntry = MEM(pa);

l1Entry.tag = pa.value<13..2>;

l1Entry.data = memEntry.data;

L1(pa) = l1Entry;

endif;

data = l1Entry.data;

}

write = {

…

l1Entry.tag = pa.value<13..2>;

l1Entry.data = data;

L1(pa) = l1Entry;

memEntry.data = data;

MEM(pa) = memEntry;

}

The read action is straightforward; the write needs some explanation. As you can see, storing data always causes the main memory to be written leading to high latency. This approach is referred to as *write-through*. There is another strategy called *write-back*. Data are stored in the cache without storing them in memory; the main memory is updated only if are the data evicted from the cache.

Currently, MicroTESK does not support the write-back strategy.

#### MMU, Version 7

Finally, let us add debug printing to the MMU specifications. There is a built-in function *trace*. Its signature is similar to C’s *printf*. Here comes a simple example.

if (TLB(va).hit) then

trace("TLB(%x).hit", va.value);

…

We suggest adding *trace* calls to all specification branches (hits, misses, exceptions, etc.). It will save your time when debugging specifications and tests.

## Testing

Base template. Text and data sections.

Simple template. Page table initialization. Accesses via mapped and unmapped virtual addresses.