Computer Architecture II: Memory Management

M.A. Hillebrand, D.C. Leinenbach, W.J. Paul

20 May 2003 — 13 June 2003

1 Introduction

20 May 2003
second half

Memory management deals with techniques (cheap and efficient!) to provide user programs with sufficient memory especially in a multitasking environment. We will treat a primitive implementation of *virtual memory* in which a user program can access a memory larger than the actual (physical) RAM; excessive parts are stored dynamically on hard-disc.

Memory management concerns both hardware and (operating system) software:

- A new piece of hardware is the *memory management unit* (MMU).

  It has a medium-size, ugly specification.

- The MMU can cause exceptions, hence it interacts with the *interrupt mechanism*.

  Interrupts mechanisms are ugly, see the start of chapter 5 of [MP00].

- One of the exceptions caused by the MMU is the page fault exception. It leads to the execution of the *page fault handler*.

  This piece of software is part of the operating system (OS).

- As far as the *user program* is concerned, there are good things that should happen (simulation theorem) but also bad things that should not happen.

  An unwanted interaction with respect to the user program is, that it should not destroy the operating system. This is a theorem about the impossibility of hacking. Hence, through the combination of the hardware mechanism and the operating system it must be guaranteed that no user program can destroy the operating system.

We have not yet done specifications of software and especially of the operating system. In fact, such things have been only done for primitive example implementations. The general approach is the standard one:

- specify operating system components,
- implement these components,
- show that the implementation meets the specification (hierarchical correctness).

For our lecture, we will do only parts of this. We will take the following simplified course of action:

- We specify
  - the MMU and some additional hardware in the processor,
  - the OS software (initialization, page fault handlers)

  but only as far as memory management is concerned.

  It will be nice and easy to show some desirable properties. The real problem lies in the (ugly) coordination of hardware and software.

- Then we construct the hardware and the software.

  For the hardware part, again, this will be easy. However, to show correctness for the software, we need, among other things, to show also the absence of interrupts. This is an open problem.

We now try to sketch the correctness theorem that we will prove for memory management; we call such a theorem "virtual memory simulation theorem".

- The correct execution of the user program is formulated by introducing a new type of memory, the *virtual memory* (VM). The virtual memory is accessed by virtual address $va \in Va$ from the set of virtual addresses $Va := \{0, \ldots, 2^{32}-1\}$. Each address stores a byte, hence a virtual memory configuration vm is a mapping from virtual addresses to bytes:

  $$vm : [Va \to \{0,1\}^8]$$

  The user program accesses the virtual memory with the usual operations; in an implementation, the user program should 'see' a uniform virtual memory.

- The difficulty arises from the fact that we do not build implement the virtual machine directly. Instead the implementation has *two* memories that hold the contents of the (user) virtual memory: the (physical) main memory and the swap memory. The main memory is implemented by RAM, it corresponds to the regular implementation memory. The swap memory is implemented as a special file / partition on hard-disc. It is used to store parts of the virtual memory that are not present in main memory.

  We model both memories as functions mapping addresses to bytes. We have the set of main memory addresses $Ma \subseteq \{0, \ldots, 2^{32}-1\}$ and the swap memory addresses $Sa \subseteq \{0, \ldots, 2^{32}-1\}$, which usually are a *strict* subset of all the implementable addresses (for cost reasons). With these sets, a main memory configuration mm and a swap memory configuration sm map addresses to bytes:

  $$mm : [Ma \to \{0,1\}^8]$$
  $$sm : [Sa \to \{0,1\}^8]$$

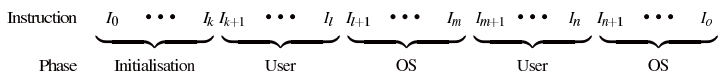

Figure 1: Simulated User Program Execution on the Implementation

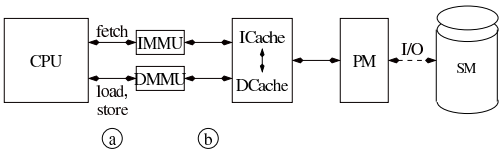

Figure 2: Overview of the hardware. At ⓐ we have to establish uniform virtual memory for the user (that means the MMUs are transparent). At ⓑ we have to establish uniform (regular) memory (that means the caches are transparent).

- When the user program is executed on such an implementation, it can only directly access the main memory. If it tries to access data that is not present in the main memory, an exception is caused. The page fault handler will fix the situation by copying data between the main and swap memory (I/O, slow). Then, the same user program instruction is repeated and hopefully also completed.

  Hence, a computation of the implementation can be divided into phases. At the start, the operating system initializes the system. Then, the operating system only takes over if the user program tries to access data not present in main memory. Hence, the user program and the operating system take turns in execution. Of course, in terms of execution efficiency, the less often the operating system is invoked, the better.

  This situation is depicted in figure 1.

  > **Data Consistency**: Every instruction execution of the user program that does not cause a page-fault, must have the same effect, as if the same instruction would be executed on virtual memory.
  >
  > **Liveness**: There must not occur page-faults indefinitely.

Figure 2 shows an overview of the hardware. Between the instruction and the data cache, there are two MMUs, the instruction MMU (IMMU) and the data MMU (DMMU). At mark ⓐ, the afore-mentioned virtual memory simulation theorem has to be established: when a user program executes, the MMU operation should be transparent to it, i.e. the user program should behave like it operates on a uniform virtual memory (ignoring the instructions executed by the operating system and the instructions with page-fault). At mark ⓑ, we have the well-known cache theorem, that states that the caches provide access to a uniform (main) memory.

ⓐ Theorem: processor in user mode sees uniform virtual memory.

ⓑ Theorem: Processor sees uniform memory (uniform: memory + definitions of operations on it) [SvenBeyer]

2 Notation

We denote *addresses* a by numbers: $a \in \{0, \ldots, 2^{32}-1\}$. Let $m \in \{vm, sm, mm\}$ be an arbitrary memory and $d \in \mathbb{N}$. By $m_d(a)$ we denote the d byte wide memory region starting at address a:

$$m_d(a) = m[a + d - 1 : a] \in \{0,1\}^{8 \cdot d}$$

We use this kind of notation only for aligned addresses, i.e. we assume that the address a is a multiple of d.

We further divide the set of memory addresses into subsets of addresses that are called *pages*. Let the parameter K denote the *page size*; we set $K = 2^{12} = 4096$. A *page* is a range of K addresses that starts at a base address being a multiple of K.

Using the page size, we can uniquely write addresses a as the binary value of two concatenated bitvectors, the *page index* $px(a)$ and the *byte index* $bx(a)$. The following conditions must be satisfied:

$$\begin{aligned} a &= \langle px(a), bx(a) \rangle \\ \langle px(a) \rangle &\in \{0, \ldots, 2^{20} - 1\} \\ \langle bx(a) \rangle &\in \{0, \ldots, 2^{12} - 1\} \end{aligned}$$

The *content of the memory page with index* x of memory m is defined as $page_m(x) = m_K(K \cdot x)$ where $x \in \{0, \ldots, 2^{20} - 1\}$.

Lemma 1 *For any memory* m *and every address* a *we have*

$$m(a) = page_m(\langle px(a) \rangle)(\langle bx(a) \rangle) \ .$$

3 Machines

Introducing the notion of the execution of user programs requires us to introduce a third machine, the virtual memory machine DLX_V. Also, the two old machines (the specification machine DLX_S and the implementation machine DLX_I) need modifications to support virtual memory.

1. The virtual machine DLX_V is basically the old DLX plus *rights*. Rights control whether a program can access (i.e. read or write) certain memory locations or not. Memory locations, which cannot be accessed are also called *protected*.

2. The specification machine DLX_S provides the new DLX instruction set specification.

   Most notably, DLX_S can run in two different modes, the user mode, in which the user program is executed, and the system mode, in which operating system code is executed.

   This requires also the addition of some extra registers.

3. The implementation machine DLX_I is the hardware implementation of the DLX_S. It features two MMUs to implement the modified (user) instruction architecture of DLX_S.

With these machines we want to prove the following two *simulations theorems*:

- $simulation_{3\to 2}$: DLX_I simulates DLX_S (hardware correctness)
- $simulation_{2\to 1}$: DLX_S and interrupt handlers simulate DLX_V (software correctness)

3.1 The Specification Machine DLX_S

The specification machine has an extended special-purpose register file. The additional registers are:

- The page-table origin register $pto \in \{0,1\}^{32}$ and the page-table length register $ptl \in \{0,1\}^{32}$.

  Both describe a special region in main memory called the *page table*. The page table will be used by the MMU to redirect user mode memory accesses to a different address. This will be defined below.

- The mode register $mode \in \{0,1\}$ consists only of a single bit.

  The processor runs in *system mode* if $mode = 0$; otherwise it runs in *user mode*.

- The $emode \in \{0,1\}$ register keeps a copy of the mode register during exception handling.

  It receives a copy of the *mode* register on entering the exception handler and copies its value to the *mode* register on `rfe`. Hence, its use is analogous to the other exception registers ($EDPC$, ESR, ...).

A configuration of the specification machine DLX_S is a triple

$$C = (R, mm, sm)$$

where R is a function mapping names of visible registers (e.g. DPC, $GPR(x)$) to their content, mm is the main and sm the swap memory configuration. The contents of individual registers are denoted by $R(r) \in \{0,1\}^{32}$ where r is a register name, i.e. $r \in \{DPC, PC', GPR(x), \ldots\}$. For memories, we let $mm(a) \in \{0,1\}^8$ and $sm(a) \in \{0,1\}^8$ denote the contents of memory cell $a \in \{0, \ldots, 2^{32}-1\}$.

As was already mentioned, the *pto* and *ptl* registers specify a table in main memory that is called the page table. The *page table* of configuration C maps indices $x \in \{0, \ldots, \langle ptl \rangle - 1\}$ to words $PT_C(x)$:

$$PT_C : [\{0, \ldots, \langle ptl \rangle - 1\} \to \{0,1\}^{32}]$$

For (page) indices greater or equal than the page-table length, the page table is undefined. Such accesses will lead to an exception. For all indices x less or equal the page-table length, the page-table entry of x is defined by

$$PT_C(x) = mm_4(\langle pto \rangle + 4 \cdot x) \ .$$

The structure of a page table entry can be seen in figure 3. We define abbreviations to access the components of a page table entry given a virtual address va.

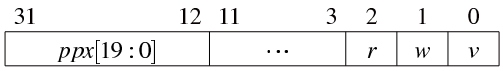

Figure 3: Page table entry

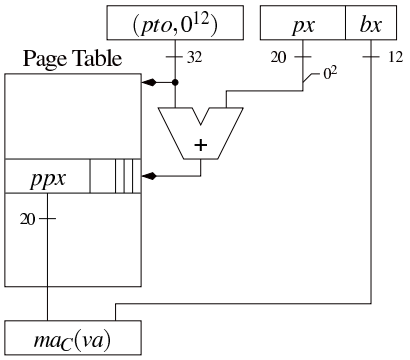

Figure 4: Address Translation for the Virtual Address $va = (px, bx)$

- With $ppx_C(va) = \langle PT_C(\langle px(va)\rangle).ppx\rangle$ we denote the physical page index of va. Under certain conditions (no exception is caused for the memory access), it indicates the page in main memory in which the contents of va are stored.
- The valid bit of va is denoted by $v_C(va) = PT_C(\langle px(va)\rangle).v$. It is 1 iff the page is valid (i.e. it is not swapped out and can be found in main memory by looking at $ppx_C(va)$).
- The read bit of va is denoted by $r_C(va) = PT_C(\langle px(va)\rangle).r$. It is 1 iff the page is readable.
- The write bit of va is denoted by $w_C(va) = PT_C(\langle px(va)\rangle).w$. It is 1 iff the page is writable.

With the physical page index, we can define the translated main memory address $ma_C(va)$ of the virtual address va by

$$\begin{aligned} ma_c(va) &= ppx_c(va) \cdot K + \langle bx(va)\rangle \\ &= \langle PT_C(\langle px(va)\rangle).ppx, bx(va)\rangle \ . \end{aligned}$$

Figure 4 shows the page-table lookup and the address translation.

If a virtual address is not valid, its page can be found in the swap memory. To look it up, we need to define a swap-memory address translation mechanism, which maps a virtual address va to a swap address sa. At the moment, the exact memory layout of the swap memory is not important, therefore, we define a trivial translation mechanisms that maps virtual addresses to an associated swap memory address by merely adding an offset *sbase* to them. The offset *sbase* should be a multiple of the page size K. We define the *swap memory address* $sa_C(va)$ of va by $sa_C(va) = sbase \cdot K + va$ and the *swap space index* $spx_C(va)$ by $spx_C(va) = sbase + \langle px(va)\rangle$. Figure 5 shows this setup.

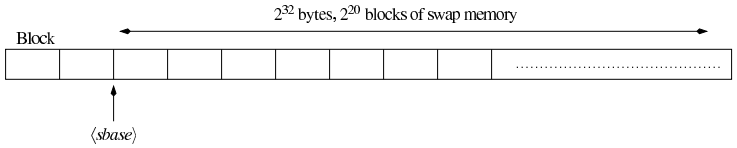

Figure 5: Swap Memory

3.2 The Virtual Machine DLX_V

A configuration of the virtual machine DLX_V is a triple $C_V = (R_V, vm, r)$. The first component R_V denotes the virtual machine registers, the second component denotes the virtual memory and the third denotes the rights function. The rights function r stores access rights for each virtual address va.[1]

$$r : [\{0, \ldots, 2^{32} - 1\} \rightarrow 2^{\{\mathrm{R},\mathrm{W}\}}]$$

The rights function controls accesses to virtual memory addresses. We denote accesses by a pair (va, mw) of a virtual address and a boolean flag.

- A read accesses $(va, 0)$ (which is either a fetch or a load) can be executed iff $\mathrm{R} \in r(va)$.
- A write access $(va, 1)$ can be executed iff $\mathrm{W} \in r(va)$.

Rights can only be defined for individual pages, addresses in the same page have the same access right. Formally, two virtual addresses va and va' that have the same page index, also must have the same access rights:

$$px(va) = px(va') \Rightarrow r(va) = r(va')$$

The definition of the configuration of the virtual machine implies the *intended* meaning of some parts of the page table:

- The read bit of page table entry for virtual address va is set iff the read right R is element of the rights function r of the configuration:

$$r_C(va) = 1 \Leftrightarrow \mathrm{R} \in r(va)$$

- The same holds for the write right:

$$w_C(va) = 1 \Leftrightarrow \mathrm{W} \in r(va)$$

- The valid bit indicates, whether we can find the contents of va in the main memory or the swap memory. The specific address is determined by the main memory or swap memory translation function.

  So, we want to satisfy the following equation:

$$vm(va) = \begin{cases} mm_C(ma_C(va)) & \text{if } v_C(va) = 1 \\ sm_C(sa_C(va)) & \text{otherwise} \end{cases}$$

[1] Notation: Given a set M the powerset, i.e. the set of all subsets, of M is denoted by 2^M.

3.3 Instruction Execution

Now we define the next step function of the specification machine and the virtual machine:

$$\delta_S(C) = (R'_S, mm', sm')$$
$$\delta_V(C_V) = (R'_V, vm', r)$$

For the specification machine, we are interested only in the case in which the user mode is active, i.e. for this section we generally assume that $R_V(mode) = 1$. In case that we are in system mode the next step function is the same as for the old DLX (no translation, no restrictive rights). Additionally, we also assume, that no exception are caused for the specification machine, i.e. we will not switch to system mode. Exception handling will be considered in later lectures.

The interesting cases for the next-state functions are obviously the instruction fetch and the execution of load/store operations. The definitions follow.

- For instruction fetch, we define the (invisible) instruction registers for both machines. For the virtual machine, the instruction register is obtained by reading four bytes from the location that the delayed PC points to. For the specification machine, we read four bytes from the *translated* address of the delayed PC.

  $$IR_V = vm_4(DPC_V)$$
  $$IR_S = mm_4(ma_C(DPC_S)) \text{ if } \neg excp_C(DPC_S, 0)$$

  Note: the exception predicate $excp_C(va, mw)$ has yet to be defined.

- If IR_V is no load/store instruction, the memory contents and the rights function do not change and the register update can be described by a function f_1 that takes the current registers and the instruction register as input:

  $$vm' = vm$$
  $$r' = r$$
  $$R'_V = f_1(IR_V, R_V)$$

  For the specification machine, the update is similar: if IR_S is no load/store instruction, the memory contents do not change and the register update is described by the same function f_1:

  $$mm' = mm$$
  $$sm' = sm$$
  $$R'_S = f_1(IR_S, R_S)$$

- If (any) IR is a load/store instruction, we compute the effective address by

  $$ea = GPR(RS1(IR)) + imm(IR)$$

  Also, we have an access width $d(IR) \in \{1, 2, 4, 8\}$.

For loads, the virtual machine's load result is computed by a direct access to the virtual memory. The specification machine's load result is computed by a *translated* access to the the main memory provided there is no translation exception.

Hence, we have:

$$\begin{aligned} \mathit{lresult}_V &= \mathit{vm}_d(\mathit{ea}) \\ \mathit{lresult}_S &= \mathit{mm}_d(\mathit{ma}_C(\mathit{ea})) \text{ if } \neg \mathit{excp}(\mathit{ea}, 0) \end{aligned}$$

For both machines, the updated register content is computed by

$$R' = f_2(R, \mathit{IR}, \mathit{lresult})$$

and the other components of the machine configuration do not change.

For stores, we have a store operand $\mathit{GPR}(\mathit{RD}(\mathit{IR}))$. The virtual machine stores the store operand in the effective address's location of the virtual memory. The specification machine stores the store operand in the *translated* effective address's location in the main memory provided there is no translation exception. Otherwise, no memory locations change.

We have:

$$\begin{aligned} \mathit{vm}_d(\mathit{ea}) &= \mathit{GPR}(\mathit{RD}(\mathit{IR}_V)) \\ \mathit{mm}_d(\mathit{ma}_C(\mathit{ea}(\mathit{IR}))) &= \mathit{GPR}(\mathit{RD}(\mathit{IR}_S)) \text{ if } \neg \mathit{excp}(\mathit{ea}, 1) \end{aligned}$$

For both machines, the updated register content is computed by a function f_3 operating on the old register content and the instruction register:

$$R' = f_3(R, \mathit{IR})$$

3.4 Specification Machine Exceptions

27 May 2003

Repetition. We have three machines, the virtual, the specification and the implementation machine, between which we want to establish simulation theorems:

$$DLX_V \xleftarrow{SW} DLX_S \xleftarrow{HW} DLX_I$$

The implementation machine DLX_I we have yet to define. For the other two, the important parts of our definition concerned the memory accesses.

- A configuration of DLX_V is a triple $C_V = (R_V, vm, r)$ where

  $$r : [Va \to \{\emptyset, \{\mathrm{R}\}, \{\mathrm{W}\}, \{\mathrm{R}, \mathrm{W}\}\}]$$

  is the rights function and $Va = \{0, \ldots, 2^{32} - 1\}$ is the set of virtual addresses.

  A memory access is a pair $(va, mw) \in Va \times \{0, 1\}$. We distinguish:

  - Fetch, where $va = DPC$ and $mw = 0$
  - Load, where $va = ea = GPR(RS1) + imm$ and $mw = 0$
  - Store, where $va = ea$ and $mw = 1$

  The machine aborts if $mw = 0 \wedge \mathrm{R} \notin r(va)$ or $mw = 1 \wedge \mathrm{W} \notin r(va)$.

- A configuration of DLX_S is a triple $C_V = (R_S, mm, sm)$. There are extra registers *pto*, *ptl*, *mode*, and *emode* $\in R_S \setminus R_V$.

  Page-table lookup and translation:

  $$\begin{aligned}
  PT_C(x) &= mm_4[\langle pto \rangle + 4x] \\
  &= \begin{array}{|c|c|c|c|c|}\hline \overset{31}{} & \overset{12\ 11}{} & \overset{3\ 2}{} & \overset{1}{} & \overset{0}{} \\ \hline ppx[19:0] & \cdots & r & w & v \\ \hline \end{array} \\
  va &= \langle px(va), bx(va) \rangle \text{ with } px(va) \in \{0,1\}^{20}, bx(va) \in \{0,1\}^{12} \\
  ppx_C(va) &= \langle PT_C(\langle px(va) \rangle).ppx \rangle \\
  r_C(va) &= PT_C(\langle px(va) \rangle).r \\
  w_C(va) &= PT_C(\langle px(va) \rangle).w \\
  v_C(va) &= PT_C(\langle px(va) \rangle).v \\
  ma_C(va) &= ppx_C(va) \cdot K + \langle bx(va) \rangle \text{ for the page size } K = 4096 \\
  sa_C(a) &= sbase \cdot K + sa \text{ for } sbase \cdot K \in Sa \text{ and } K \text{ divides } sbase
  \end{aligned}$$

The translation mechanism above (which maps C, va to $ma_C(va)$) we used to define the semantics of the (new) DLX_S:

- In system mode, $mode = 0$, "nothing" new happens, no translation.
- In user mode, $mode = 1$, we map $va \mapsto ma_C(va)$ and use it for the access, if $\neg excp_C(va, mw_C)$. The virtual address va is either $\langle DPC \rangle$ or $\langle ea \rangle$.

We have yet to define when (and which) exceptions are caused in the DLX_S due to page-table lookups / address translation. We start by defining some *auxiliary* signals. Let (va, mw) be a memory access.

Corrected stuff ahead!

1. The illegal operation exception $ill_C(va, mw)$ is signalled if the page index of va lies outside the page-table, or the memory operation type mw

violates the rights stored in the page table entry. For $mw = 0$, such a violation is called read rights violation, for $mw = 1$ it is called write rights violation. We define:

$$ill_C(va, mw) = (\langle px(va) \rangle \geq ptl) \vee (mw \wedge \neg w_C(va)) \vee (\neg mw \wedge \neg r_C(va))$$

2. The pagefault exception $pf_C(va, mw)$ is signalled if the page-table entry is not valid (and there is no illegal operation exception):

$$pf_C(va, mw) = \neg ill_C(va, mw) \wedge \neg v_C(va)$$

3. Both conditions are subsumed in the general translation exception condition $excp_C(va, mw)$:

$$excp_C(va, mw) = ill_C(va, mw) \vee pf_C(va, mw)$$

With these definitions, we define three new types of exceptions connected to memory management:

1. An illegal memory operation exception is caused iff the fetch was an illegal operation or if the fetch has caused no translation exception but there is an illegal load/store operation:

$$illm_C \Leftrightarrow ill_C(DPC, 0) \vee \\ \neg excp(DPC, 0) \wedge loadstore(IR_S) \wedge ill_C(ea, mw(IR_S))$$

2. A page-fault on fetch exception is caused iff the fetch caused a page fault:

$$pff_C \Leftrightarrow pf(DPC, 0)$$

3. A page-fault on load/store exception is caused iff the fetch caused no translation exception but there was a page-faulting load/store operation:

$$pfls_C \Leftrightarrow \neg excp(DPC, 0) \wedge loadstore(IR_S) \wedge pf_C(ea, mw(IR_S))$$

Excursion: Interrupt Handling. In a jump-interrupt-service-routine condition ($JISR = 1$), the machine performs several updates simultaneously:

$$SR = 0^{32}$$
$$ESR = SR$$
$$ECA = MCA$$
$$EPC = PC', \quad EDPC = DPC \text{ for repeat}$$
$$EPC = \text{next}PC', \quad EDPC = \text{next}DPC \text{ for continue}$$
$$EDATA = \begin{cases} imm26 & \text{if } trap \\ ea & \text{if } loadstore \\ DPC & \text{if } continue \end{cases}$$

Conclusion of all these updates:

- the ISR can compute the interrupt level $il = \min\{j \mid ECA[j] = 1\}$.
- For *pff*, *pfls* and *illm* it can reconstruct the virtual address va that caused the interrupt: for *pff*, it is $EDPC$, for *pfls* it is $EDATA$.

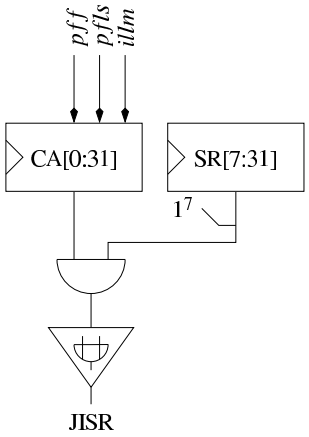

Figure 6: JISR Computation

To protect sensitive registers in user mode, we must cause an illegal operation exception on access of *pto*, *ptl* and several other registers if we are in user mode: `movi2s` with *mode*, *pto*, *ptl* or *emode* as destination (and several others, too) should cause an *ill* exception.[2] Hence, the definition of the illegal operation exception now depends also on the *mode* register.

The mode register is reset on *JISR* and restored from *emode* on `rfe`.

This completes the specifications of our machines.

4 Hardware Implementation

Next we will:

1. Construct hardware and prove a hardware simulation theorem.
2. Write handlers for *pff* and *pfls* and prove a software simulation theorem.

Naturally, we start with 1.

Arguments for memory accesses are *mode* for loads, stores and fetches, and, if $mode = 1$, also *pto* and *ptl*. We want to have a lemma of the form that during one access the arguments should be stable (using $I(k,T) = i$ and R^i_S).

- For load/store, we could use extra arguments in reservation station for load/store functional unit.
- This does not work for *pff*, hence, we have to establish by other means, that the arguments do not change during an access.

Conjecture: only a *few* precautions are necessary, since

- for $mode = 1$ the registers *pto*, *ptl* and *mode* will stay constant,
- For $mode = 0$ the registers *pto*, *ptl* and *mode* are not used for address computation (system mode = untranslated!).

In the following lectures, we will make this precise.

[2] Attention, $ill \neq illm$!

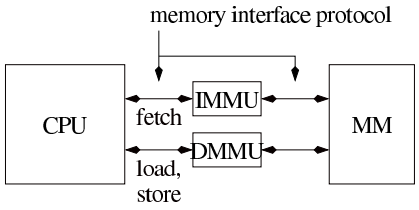

Figure 7: Processor Overview with MMUs

4.1 MMU Design

30 May 2003

In our processor implementation, we have two MMUs which are placed between the caches and the processor core. The instruction MMU is used for instruction fetches, the data MMU is used for loads and stores. This is shown in figure 7.

Recall:

$$
\begin{aligned}
R_S &= R_V \cup \{pto, ptl, mode, emode\} \\
C &= (R_S, mm, sm) \\
C \mapsto PT_C(x) &= mm_4(\langle pto \rangle + 4 \cdot x) \\
&= \begin{array}{|c|c|c|c|c|}\hline 31 \ldots 12 & 11 \ldots 3 & 2 & 1 & 0 \\ \hline ppx[19:0] & \cdots & r & w & v \\ \hline \end{array} \\
ma_C(va) &= \langle PT_c(\langle px(va) \rangle).ppx, bx(va) \rangle \text{ for } va = \langle px(va), bx(va) \rangle
\end{aligned}
$$

The MMU performs translation and exception computation if $mode = 1$.

The following signals form the interface between the processor and a single MMU:

- MMU inputs from the processor:
  - $p.addr$ ($= DPC$ for instruction fetch, $= ea$ for load/store)
  - $pto, ptl, mode$
  - $p.mw, p.mr$ processor memory write and read
  - $m.data.in$ for stores
- MMU outputs to the processor:
  - $m.data.out$ ($= I$ for instruction fetch, $= dout$ for load)
  - $m.busy$
  - Exceptions:
    - *illm*
    - *pff*
    - *pfls*

The interface between the MMU and the cache is unchanged, it is the old interface between the processor and the cache. Both interfaces must obey the standard memory protocol, of which figure 8 shows two examples. For a cache hit, the memory responds in the same cycle the request was started. For a cache miss, it may take many cycles, before the cache responds with $\neg m.busy$ after it has loaded the necessary line. As an input convention, the processor has to keep

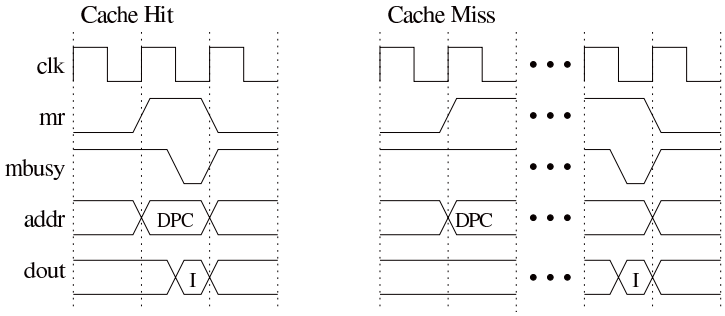

Figure 8: Memory Protocol Example: Cache Hit and Cache Miss for Instruction Fetch.

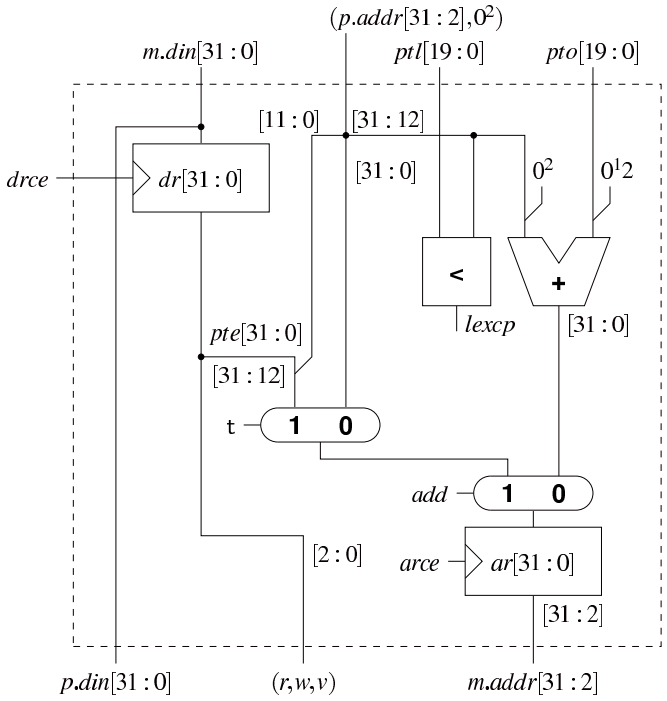

Figure 9: Datapaths of the MMU where $p.t = t = mode$ and ar, dr are the address and the data register.

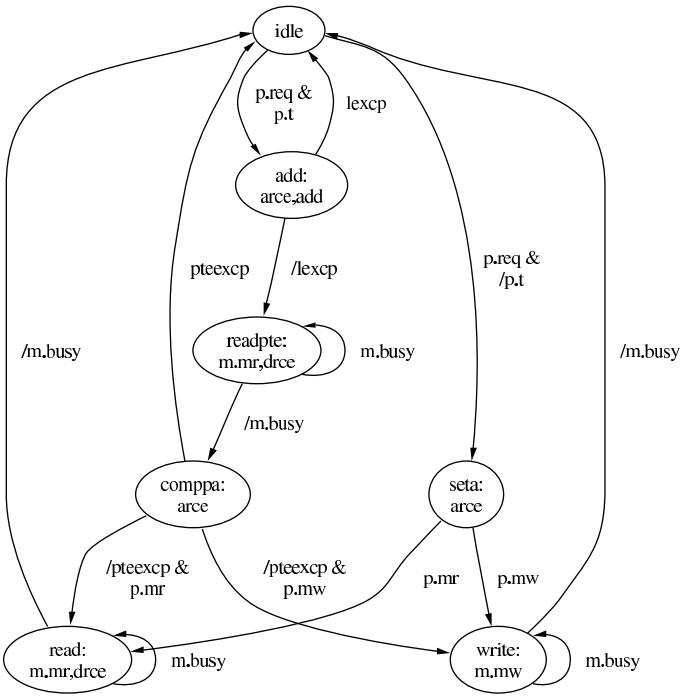

Figure 10: Control of the MMU. Define $p.req := p.mr \vee p.mw$. Additionally, we have the Mealy control signal $p.busy := \neg((state = read \vee state = write) \wedge \neg m.busy)$.

the input data and the requests signals stable (i.e. it must not change them) as long as the memory is busy.

We will examine a *slow* MMU design in this lecture. The datapaths of this design are shown in figure 9, the control of this design (FSD) is shown in figure 10. The busy signal of the MMU to the processor is a Mealy signal. We must pay attention that the MMU signals busy even in the first cycle of the request. Our approach is therefore, to make the MMU signal not busy only when it enters the *idle* state again, i.e. when it is in state *read* or *write* and the cache / memory signals not busy $\neg m.busy$:

$$p.busy_{new} := \neg((state = read \vee state = write) \wedge \neg m.busy)$$

Local Correctness of the MMU. We have several cases according to the following criteria:

- Translated / Untranslated.
- Read operation / Write operation.
- Exception / No exception.

Lemma 2 (Paths) *Claims about the path followed for the different cases through the FSD:*

- *For translated read without exception, the path followed in the FSD is*

$$idle \rightarrow add \rightarrow readpte^{+} \rightarrow comppa \rightarrow read^{+} \rightarrow idle \ .$$

- *Similar claims for all the other cases.*

Lemma 3 (Correctness) *In case i on path for i "happens what we want" (i.e. what is defined by the DLX_S).*

Proof. We proof both lemmas only for the translated read case without exceptions. The proofs for the other cases are similar.

- Assume the request starts in cycle t in state *idle*:

$$\begin{aligned} p.req^{t-1} &= 0 \\ p.req^{t} &= 1 \\ state^{t} &= idle \end{aligned}$$

- In cycle $t+1$ we are in state *add*. Therefore, the address register at time $t+2$ contains the address of the page-table entry:

$$\begin{aligned} \langle ar^{t+2} \rangle &= \langle pto \rangle^{t+1} + 4 \cdot \langle px(p.addr^{t+1}) \rangle \\ &= \langle pto \rangle^{t} + 4 \cdot \langle px(p.addr^{t}) \rangle \text{ (if inputs stable!)} \end{aligned}$$

- Between time $t+2$ and t' for some $t' \geq t+2$, we access the cache to read the page-table entry. I.e., we have for $\tilde{t} \in \{t+2, \ldots, t'-1\}$, that $m.busy^{\tilde{t}}$ and $m.mr^{\tilde{t}}$. Also, we have $\neg m.busy^{t'}$ and $m.mr^{t'}$.

By the cache / the memory specification, the contents of the data register at time $t'+1$ correspond to the data read from the memory. More precisely, the memory read that is acknowledged (i.e. signalled not busy for the first time) in cycle t' returns the data of the memory configuration in the same cycle t' looked up at the input address that was supplied in cycle $t+2$.

$$dr^{t'+1} = \begin{cases} (\star^{32}, mm_4^{t'}(ar^{t+2})) & \text{if } ar^{t+2}[2] = 0 \\ (mm_4^{t'}(ar^{t+2}), \star^{32}) & \text{otherwise} \end{cases} .$$

Observe, that during the whole request to the cache, the address register does not change its value, i.e. for all $x \in \{t+2, \ldots, t'\}$ we have $ar^x = ar^{t+2}$. Since we already assume, that the processor address bus $p.addr$ does not change, we obtain that this reads the desired page-table entry, i.e. we have

$$\left.\begin{array}{ll} dr^{t'+1}[31:0] & \text{if } ar^{t+2}[2] = 0 \\ dr^{t'+1}[63:32] & \text{otherwise} \end{array}\right\} = PT^{t'}(\langle px(p.addr^t)\rangle) .$$

- As we are in state *comppa* in time $t'+1$, we get that the value of the address register in time $t'+1$ is correct, i.e. it corresponds to the translation defined by the DLX_S:

  $$\langle ar^{t'+2}\rangle = ma_C(p.addr^t)$$

  Attention: to keep this well defined, aside from the regular inputs, the expressions

  $$PT_C(\langle px(p.addr)\rangle), \quad pto, \quad ptl, \quad mode$$

  must stay constant during the request. We will treat this problem when we integrate the MMU into the whole processor design.

- At time $t'+2$ we start another memory request (state *read*). This requests ends at time t'' for some $t'' \geq t'+2$. I.e., we have for $\tilde{t} \in \{t'+2, \ldots, t''-1\}$, that $m.busy^{\tilde{t}}$ and $m.mr^{\tilde{t}}$. Also, we have $\neg m.busy^{t'}$ and $m.mr^{t'}$. The end of this memory request t'' is also the end of the MMU request, since $\neg p.busy^{t''}$.

  At the time of acknowledgment, the data read from the memory configuration at the time of acknowledgment is returned.

  So, finally we have

  $$p.din^{t''} = mm_8^{t''}(\langle ar^{t''}[31:3], 000\rangle) .$$

  As before, we need additional arguments for our implicit assumptions here.

4.2 MMU Integration

3 Jun 2003

In this lecture we explain how the MMU is correctly integrated into the processor. Our local MMU correctness proof already has had several assumptions that all talk about certain inputs of the address translation being constant over the whole duration of a translation request. Guaranteeing these assumptions is a non-trivial task; in fact, we identify four groups of inputs, that all need *different* arguments to be provably stable. When we do this, we will (again)

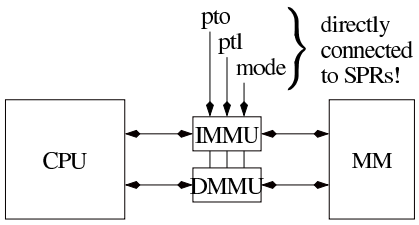

Figure 11: MMU SPR Inputs

mainly consider the case of a translated read access without exception. Such a request occurs for instruction fetches or data loads. Of these two cases, we will concentrate on the instruction fetch. All the other cases are similar or simpler.

We assume that the request starts at time t, i.e. $p.req^t$ and $\neg p.req^{t-1}$ (or $\neg p.busy^{t-1}$). We have seen from the proof of the last lecture, that there is a time $t'' > t$, such that for all times $\tilde{t} \in \{t, \ldots, t''-1\} : p.busy^{\tilde{t}}$ and $\neg p.busy^{t''}$.

Relevant inputs for a translated memory operation request to the MMU consist of four groups G_i:

- G_0 consists of the "regular" inputs supplied by the processor, $p.addr$, $p.mw$, $p.mr$ and $p.dout$.
- G_1 consists of the special purpose registers ptl, pto and $mode$.
- G_2 consists of the page table.
- G_3 (for reads) consists of the accessed memory contents, $mm(ma(va))$.

Consider now the instruction fetch of instruction I_i. The time t_i at which this instruction fetch starts, is the minimal time in which the request to the instruction MMU is active and the scheduling function indicates I_i to be in the fetch stage:

$$t_i := \min\{t' \mid p.req^{t'} \wedge I(fetch, t') = i\}$$

An access that was started in cycle t is not finished in cycle τ iff the busy signal is still active. We denote this by the predicate $nf(t, \tau)$ that is defined as

$$nf(t, \tau) := \tau \geq t+1 \wedge \forall x \in \{t+1, \ldots, \tau\} : p.busy^x = 1 \ .$$

Since our MMU is slow, for $t = t_i$ the fetch of I_i is not finished.

We define the end of the request. It is the time after the start of the requests in which the busy signal first becomes inactive:

$$t''(t) = \min\{x \geq t+1 : p.busy^x = 0\}$$

(The existence of t'' requires a liveness proof. This proof was implicit in the last lecture, it uses that $m.busy$ cannot stay active indefinitely for ongoing requests to the cache.)

The following lemma is an easy implication of the local MMU correctness shown in the last lecture.

Lemma 4 (Translation Lemma) *Hypothesis: we assume the input signals stay constant during the access, i.e. we have for all* $i \in \{0, \ldots, 3\}$

$$(G_i)^\tau = (G_i^t) \text{ for } nf(t, \tau) \ .$$

Then, the data returned by the MMU is that of the translated memory location at the start of the request:

$$p.din^{t''} = mm_4^t(ma^t(p.addr^t))$$

Proof. We have seen in the previous lecture, that

$$p.din^{t''} = mm_4^{t''}(ma^{t'}(p.addr^t)) \ .$$

With the hypothesis, this already gives the claim.

The big question is now, how do we guarantee the hypothesis $(G_i)^\tau = (G_i)^t$? As it turns, we require quite different arguments for the different i.

- G_0.

  By construction, $p.busy^\tau = 1$ (the *new* busy signal!). Hence, we have $in^{\tau+1} = in^t$ for $in \in \{p.addr, p.mw, p.mr, p.dout\}$, because the processor does not change the inputs as long as the MMU signals that it is still busy, i.e. *p.busy* is active. (This is an inductive argument.)

- G_1.

  The idea is to stall the instruction fetch. We define the fetch signal which is the read request signal for the IMMU. This definition consists of two parts. The first part stalls the instruction fetch if there is an instruction in the *decode* stage (stage S_1) which may modify *pto*, *ptl* or *mode*. The second part stalls the instruction fetch if there is a non-terminated instruction modifying *pto*, *ptl* or *mode*.

  $$\begin{aligned} IMMU.p.req = \\ fetch = \neg(S_1.full \wedge (S_1.ID.rfe \vee \\ S_1.ID.movi2s \text{ writing } pto, ptl, mode)) \\ \wedge\ pto.v \wedge ptl.v \wedge mode.v \end{aligned}$$

  See figure 12 for a sketch of how the stage 1 decode signals are computed.

  Assume that I_j is a `rfe` or `movi2s` writing *pto*, *ptl* or *mode*. If it is issued at time x, then at time $x+1$ stage 1 may be empty, i.e. we possible have

  $$I(issue, j) = x \Rightarrow \neg S_1.full^{x+1} \ .$$

  But at time $x+1$, one of the valid bits of *pto*, *ptl* or *mode* would be zero, i.e.

  $$\exists y \in \{pto, ptl, mode\} : \neg y.v^{x+1} \ .$$

  Now, assume that I_j terminates in time z, i.e. $I(wB, j) = z$. Then again, for all the inspected valid bits are turned on again, $\forall y \in \{pto, ptl, mode\} : y.v^{z+1}$. *Then*, we can fetch.

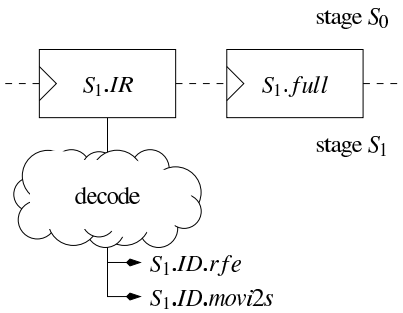

Figure 12: Computation of Stage 1 Decode Signals

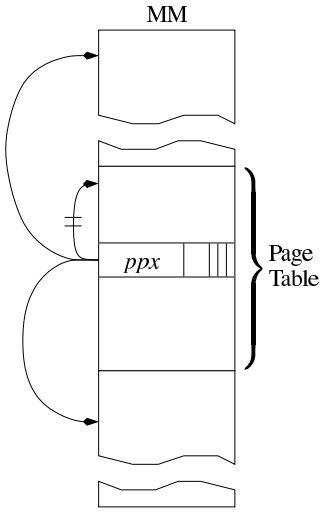

Figure 13: Page Table Convention

Lemma 5 (Fetch Lemma) *$I(fetch, t) = i \wedge fetch^t = 1$ implies all instructions I_j with $j < i$ that write to ptl, pto or mode have terminated.*

Hence, we get $G_1^\tau = G_1^t$ (this works only for fetch, not for data access).

- G_2, the page table.

  We assume that the operating system satisfies the following *page table convention*:

  Let s be the index of any page containing a part of the page table. Consider a page-table entry $PT(x)$. If the page-table entry is valid, $PT(x).v = 1$, then the physical page index must be different from s, i.e. $\langle PT(x).ppx \rangle \neq s$. Hence, in translated mode the page table cannot be accessed (neither written to or read from).

  The convention is visualized in figure 13. Mappings outside the page table are allowed, mappings inside the page table are not.

  The page-table convention gives us the following lemma.

Lemma 6 (Page Table Lemma) *If the page-table convention holds, the page table stays constant during translation.*

Proof. Consider the translated fetch of I_i starting at time t. Thus, we have $mode^t = 1$.

There exists a cycle $t' < t$, where the processor was last in system mode, i.e. $mode^{t'} = 0$ and t' is maximal $(\forall \tilde{t} \in \{t'+1, \ldots, t\} : mode^{\tilde{t}} = 1)$.

Assume that the instruction in the write-back stage at this time t' has index $j < i$,

$$I(wB, t') = j \ .$$

The instruction I_j can only be an `rfe` or `movi2s` writing the *mode* register.

By the definition of the fetch signal, I_j has already terminated at time t and also by in-order termination all instructions before I_j have terminated.

By the page-table convention, no user mode instruction I_k for $j < k < i$ can change the page-table.

Hence, the page table does not change: $G_2^\tau = G_2^t$.

Let us reiterate. 6 Jun 2003

We have identified four groups of inputs to the MMU. The first two groups are concerned with the inputs coming from the processor:

$$G_0 : va = p.addr^t, p.rd^t, p.wr^t, p.req^t$$
$$G_1 : pto^t, ptl^t, mode^t$$

The other two groups are concerned with inputs coming from the memory:

$$G_2 : PT^t$$
$$G_3 : mm^t(ma^t(va)) \text{ for reads}$$

We were in the proof of the translation lemma: if the input groups G_i do not change, the MMU computes translated read operation. We summarize the reasoning again here and treat the missing case for G_3:

- G_0

  By generation of $p.busy$

- G_1

  By fetch stall condition and fetch lemma.

  $$\begin{aligned} fetch = \ & \neg(S_1.full \wedge (S_1.ID.rfe \vee S_1.ID.movi2s \text{ writing } pto, ptl, mode)) \\ & \wedge pto.v \wedge ptl.v \wedge mode.v \end{aligned}$$

  Fetch lemma: if I_i set the mode bit then for all $j \leq i$, the instruction I_j is terminated before the translation of fetch for I_{i+1} starts.

- G_2

  By the page-table convention guaranteed by the operating system (figure 13) and the page table lemma (if the page-table convention holds, the page table stays constant during translation).

- G_3

  Sync condition: there must be a sync (`movs2i R0, IEEEf`) before a fetch from a modified location. Formally, let y be a physical address. If the instruction I_i is a translated fetch from y, and instruction I_j for $j < i$ writes to y, there must be a k with $j < k < i$ such that I_k is a sync instruction.

  For this to work, we need a strengthened sync, which prevents fetching of the next instruction already: $S_1.full \wedge S_1.ID.sync$ should imply $\neg fetch$.

  Lemma 7 (Sync Lemma) *From the sync condition we get* $(G_3)^\tau = (G_3)^t$ *for all* $\tau \in \{t, \ldots, t''\}$.

  Proof. Omitted here, similar to the page-table lemma.

Now we can show the global fetch correctness: the instructions that the implementation machine fetches correspond to the instructions that the specification machine fetches.

Theorem 1 (Fetch Theorem) *Let* $I(fetch, t) = i$ *and assume the translation starts in cycle* t *and ends in cycle* t''*. We claim:*

$$p.din^{t''} = IR_S^i$$

Proof. Assume correct simulation until the start of cycle t. With this obtain

$$pto^t = pto_S^i, \quad ptl^t = ptl_S^i, \quad mode^t = mode_S^i \ .$$

The special purpose register are used in the MMU without testing the valid bits! But, by the fetch condition, the valid bits are on, since otherwise we would not yet have started to fetch.

By the assumption, the address fed into the instruction MMU is equal to the delayed PC of the specification instruction i:

$$IMMU.p.addr^t = DPC_S^i$$

From (G_2) we get that $PT^t = PT_S^i$ and $ma^t(IMMU.p.addr^t) = ma_S^i$.

By induction assumption, the memory contents of time t correspond to the memory contents of the specification machine at step i:

$$mm_4^t(IMMU.p.addr^t) = mm_{S,4}^i(ma_S^i(DPC_S^i))$$

Abbreviate $y := ma_S^i(DPC_S^i)$. Let j be the index of the instruction which last wrote $mm(y)$:

$$j = \max\{k < i \mid k \text{ writes } mm_S(y)\}$$

From the Tomasulo proof we know that $mm^t(y) = mm_S^i(y)$ if I_j is terminated at time t. This is true by the sync lemma.

By applying the translation lemma we get our claim:

$$\begin{aligned} IMMU.p.din^{t''+1} &= mm_4^t(ma^t(IMMU.p.addr^t)) \\ &= mm_S^i(ma_S^i(DPC_S^i)) \\ &= IR_S^i \end{aligned}$$

Now we prove a similar theorem for a translated load.

Assume the instruction I_i is a translated load and is in the memory stage at time t:

$$I(mem, t) = i$$

Furthermore, assume that the request signal to the data MMU is already activated, i.e. $DMMU.p.req^t$, and that t is minimal with respect to these conditions.

Let t'' denote the end of the request. Let $d^i \in \{1, 2, 4, 8\}$ denote the width of the operation in bytes (and $mbw^i \in \{0,1\}^8$ the corresponding byte write signals). Let ea^i denote the effective address.

Theorem 2 (Load Theorem) *The data MMU returns at time $t''+1$ the specified load data of instruction i:*

$$DMMU.p.din^{t''+1} = mm^i_{S,d^i}(ma^i_S(ea^i_S))$$

Proof. Assume correct simulation until the start of cycle t. Especially, this assumption already contains fetch correctness, which we proved in the fetch theorem. With this obtain

$$pto^t = pto^i_S, \quad ptl^t = ptl^i_S, \quad mode^t = mode^i_S \ .$$

Because of the assumption, we get that the signals fed to the data MMU correspond to their specified counterparts:

$$DMMU.p.addr^t = ea^i_S$$
$$DMMU.p.mbw^t = mbw^i_S$$

From this, we also obtain $ma^t(DMMU.p.addr^t) = ma^i_S(ea^t_S)$. Since we do in-order load/store, the memory contents in time t correspond to the specified memory contents in step i, i.e. $mm^t = mm^i_S$.

Therefore $mm^t(ma^t(DMMU.p.addr^t)) = mm^i_S(ma^i_S(ea^t_S))$.

By the assumptions on G_2 and in-order load/store, for all $x \in \{t, \ldots, t''\}$ we also have

$$mm^x(ma^x(DMMU.p.addr^x)) = mm^i_S(ma^i_S(ea^t_S))$$

This satisfies the assumptions of the translation lemma, we can conclude the claim

$$DMMU.p.din^{t''+1} = mm^i_{S,d^i}(ma^i_S(ea^i_S))$$

We will not prove the following theorem, which is similar to the load theorem:

Theorem 3 (Store Theorem) *The data MMU executes translated stores correctly.*

All in all, these are the main results to obtain hardware correctness:

Theorem 4 *DLX_I simulates DLX_S.*

(Actually, we would have to go through the whole Tomasulo proof again to establish this result. We can argue, the rest of the processor hardware, apart from the MMUs, was not touched, and so, the same proof will go through without change. In a theorem prover, like in PVS, this is possibly *much* work.)

5 Software for the DLX_S

We will now talk about the software for the DLX_S, especially about the page fault handler part.

The operating system software enforces a memory organization on the user program in which the memory used for OS purposes and the memory used by the hardware ($\rightarrow$ translation) is strictly separated from the memory that the user program can access. The part used by the user program is called the *user memory*, the part used by the operating system and the hardware is called *system memory*.

In particular, the system memory is part of the main memory. It is modelled by a set of page indices in main memory $Sys \subseteq \{0, \ldots, 2^{20} - 1\}$. The following are examples of the data structures maintained in the system memory by the operating system:

- Operating system code & data
- The page table
- *sbase* (from the swap memory translation: $sa_c(va) = sbase + va$)
- MRL (an index of the most recently loaded, i.e. swapped-in, page)

The operating system will let the user program access a part of the main memory that is called the user memory. Through its data structure, the operating system maintains two sets of page indices:

- *allocP* is the set of allocated pages for the user memory.

  To guarantee that the user program cannot access the system memory (e.g. the page table), we want that

  $$Sys \cap allocP = \emptyset \ .$$

- $freeP \subseteq allocP$ is the set of free pages in the user memory. Any $p \in freeP$ is reserved for the user but has not been used yet.

  After initialization, the whole user memory is free, and we have $freeP = allocP$. Then, $freeP$ will get smaller whenever a reference for a yet unreferenced page was made. When $freeP = \emptyset$, swapping starts, i.e. if there is a page fault, one page must be swapped out (written back to the swap memory) and one page must be swapped in (loaded from the swap memory).

Additionally, the operating system needs to maintain information on the used (i.e. non-free) pages of the user memory: for every such page we want to determine efficiently which is the virtual page index that has been mapped to it.

We present now concrete data structures with which we implement the page fault handlers.

We assume that the operating system is willing to let the user program use a pages in main memory starting from the page with index *abase*. These pages form the user memory. We index the user memory pages from 0 to $a - 1$; user memory page i corresponds to the main memory page $abase + i$.

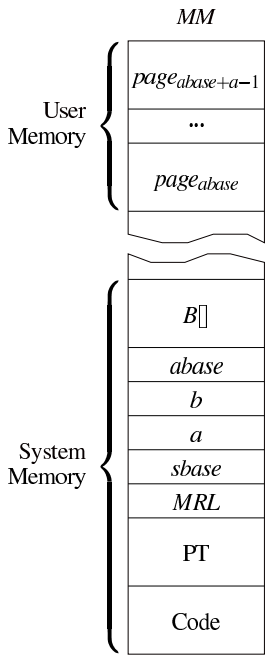

Figure 14: Memory Map with the System Area and the User Area

The operating system keeps a counter $b \leq a$ that denotes the number of used (i.e. non-free) pages in the user memory. The used pages will occupy the lower part of the user memory. This means that the user memory page i is used iff $i < b$.

An array of words $B[0 : a-1]$ is used to keep track of the *virtual* page indices of used pages in the user memory: the entry $B[i]$ (for $i \in \{0, \ldots, b-1\}$) contains the virtual page index of the page occupying the user memory page i. Let *validVP* denote the set of all (virtual) page indices stored in the array B, i.e. we set

$$validVP = \{B[0], \ldots, B[b-1]\} \ .$$

The set *validVP* can be used when we look for eviction pages (i.e. pages that are to be swapped out when the user memory is full—which happens when $a = b$). Eviction candidates can be found by choosing elements from the set $validVP \setminus \{MRL\}$.

The set of page indices we mentioned before can be defined in terms of the variables a, *abase* and b:

$$allocP = \{abase, \ldots, abase + a - 1\}$$
$$freeP = \{abase + b, \ldots, abase + a - 1\}$$

Figure 14 shows a detailed memory map for our system. Recall that we use for a page index x and any memory m the notation $page_m(x)$ to denote the contents of page x in memory m:

$$page_m(x) := m_{4096}(x \cdot 4096)$$

Handler for Page-Fault on Fetch. The algorithm must handle several cases. Here, we treat only the easy case: there is a free, allocated physical page left. The other case will be treated for the handler for page-fault on load/store.

So, assume $\mathit{freeP} \neq \emptyset$. Let e denote the minimal element from freeP (in fact we must have $e = b$). The page fault handler has to do the following things:

1. Update the ppx-field of the faulting page-table entry point to the page index e (where the swapped-in page will be placed):

$$\mathit{ppx}(\mathit{EDPC}) := e$$

2. Swap in the page:

$$\mathit{page}_{mm}(e) := \mathit{page}_{sm}(\mathit{px}(\mathit{sbase} + \mathit{EDPC}))$$

3. The $\mathit{validVP}$ has to be updated by adding $\mathit{px}(\mathit{EDPC})$ to it. This can be achieved by storing $\mathit{px}(\mathit{EDPC})$ in its last entry.

$$B[b-1] := \mathit{px}(\mathit{EDPC})$$

4. Update the freeP set by incrementing b:

$$b := b + 1$$

5. Update the MRL variable by setting

$$\mathit{MRL} := \mathit{px}(\mathit{EDPC}).$$

6. Return from exception by `rfe`.

Observe:

- Using `rfe` satisfies the synchronization condition.
- Since $e \in \mathit{freeP} \subseteq \mathit{allocP}$ and $\mathit{allocP} \cap \mathit{Sys} = \emptyset$ we also have $e \notin \mathit{Sys}$. This helps us to keep the PT condition satisfied.

Handler for Page-Fault on Load/Store. We sketch the case for $\mathit{freeP} = \emptyset$. In this case, we must swap out a page to make room in the user memory and then swap in the faulting page. Choose an eviction candidate victim e from the set $\mathit{validVP} \setminus \{\mathit{MRL}\}$. (Note: e is now a *virtual* page index not a physical.)

This choice guarantees $e \neq \mathit{MRL}$. If we do not have this, we could deadlock the user program: Consider the following, endless sequence of page faults:

1. Page-fault on fetch, swap-in the fetch page.
2. Page-fault on load-store, swap-in the load-store page by evicting the fetch page (bad!).
3. Page-fault on fetch, swap-in the fetch page by evicting the load-store page (bad!)
4. Goto 2.

So, choosing a purely random victim will not work for the proof.

Now we perform the following steps:

1. Save the eviction page:

$$page_{sm}(sbase/4096 + e) := page_{mm}(ppx(e \cdot 4096))$$

   Note: if e is the value of the array entry $B[i]$, then of course, we have $abase + i = ppx(e \cdot 4096)$ and can simplify the code accordingly.

2. Mark the swapped out page as invalid:

$$v(e \cdot 4096) = 0$$

3. Swap in the page:

$$page_{mm}(ppx(e \cdot 4096)) := page_{sm}(px(sa(EDATA)))$$

4. Update the ppx field and the valid bit v in the PTE:

$$ppx(EDATA) = ppx(e \cdot 4096)$$
$$v(EDATA) = 1$$

5. If i is the index of e in the array B, then we must update $B[i]$ as follows:

$$B[i] := px(EDPC)$$

6. Update the MRL variable by setting

$$MRL := px(EDATA).$$

7. Return from exception by `rfe`.

6 Simulation Theorem

13 Jun 2003

We have already established hardware correctness: DLX_I simulates DLX_V.

- DLX_I is the new implementation machine, i.e. the old DLX_I plus two MMUs and a few gates.
- DLX_S is the new specification machine with two modes. In mode 0 (system mode), it operates like the old DLX_S. In mode 1 (user mode), it uses address translation for pages that are in main memory and causes a page-fault exception otherwise.

Now we establish a simulation theorem between the virtual machine DLX_V and the specification machine DLX_S. DLX_V is the virtual machine. It resembles the old DLX_S (assume for now: without exceptions) and has a rights function for memory protection. We assume that the rights never change.

Recall how we defined instruction execution for the DLX_V and for the DLX_S (in user mode).

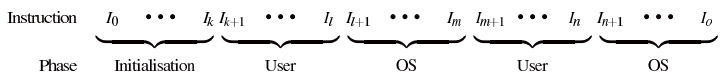

Figure 15: Phases of Computation of DLX_S

Have a virtual machine configuration $c_V^i = (R_V^i, vm_V^i, r_V^i)$. The (hidden) instruction register is defined by $IR_V^i = vm_{V,4}^i(DPC_V^i)$. If IR_V^i is no load nor store instruction (which we denoted by $\neg loadstore(IR_V^i)$), then the next processor configuration c_V^{i+1} can be computed by the function f_1 applied to the instruction register and the register contents:

$$c_V^{i+1} = (f_1(IR_V^i, R_V^i), vm_V^i, r_V^i)$$

Have a specification machine configuration in user mode $c_S^j = (R_S^j, mm_S^j, Sm_S^j)$. We set $IR_S^j = (mm_{S,4}^j(ma^j(DPC_S^j)))$ if the instruction word is in main memory, i.e. we get no page-fault on fetch exception. The DLX_V updates its configuration according to the equation

$$c_S^{j+1} = (f_1(IR_S^j, R_S^j), mm_S^j, sm_S^j) \ .$$

Note: DLX_S has more registers than DLX_V. These additional register do not change their value under f_1. (Strictly speaking f_1 is another function here).

Theorem 5 *DLX_S and OS simulate DLX_V for the user program.*

In this theorem, we denote the configuration of the DLX_V machine in step j by c_V^j. The configuration of the DLX_S machine in cycle i is denoted by c_S^i.

We define a projection Π that maps specification machine configurations to virtual machine configurations. Have $c_S = (c_S, mm_S, sm_S)$ and $\Pi(c_S) = (R_V, vm_V, r) \in C_V$. Then, the components are (uniquely) defined as follows:

- For any register name r of DLX_V:

$$R_V(r) := R_s(r)$$

- $\mathrm{R} \in r(va) \Leftrightarrow r_c(va) = 1$
- $\mathrm{W} \in r(va) \Leftrightarrow w_c(va) = 1$
- $vm(va) = \begin{cases} mm_s(ma_c(va)) & \text{if } v_c(va) = 1 \\ sm_s(sa_c(va)) & \text{otherwise} \end{cases}$

We assume that after power-up (or reset), the machine DLX_S reaches a cycle $\alpha + 1$ in which the first system mode phase is completed and the initialization of the user program is finished. This means we have $mode^t = 0$ for $t \leq \alpha$ and $mode^{\alpha+1} = 1$.

We take the projected configuration of cycle $\alpha + 1$ to be the initialization configuration of the virtual machine.

$$c_V^0 := \Pi(c_S^{\alpha+1})$$

The initialization procedure is actually quite easy. It guarantees that for all virtual addresses there is nothing in the main memory and the rights are noted in the page table. Let $va \in Va$. We must have $vm(va) = sm(sa(va))$, correct rights in the page table and a cleared valid bit in the page-table entry, i.e. $v^{\alpha}(va) = 0$.

Lemma 8 (Step Lemma) *Assume the i-th configuration of the specification machine and the j-th configuration of the virtual machine are equal by projection, i.e. $\Pi(c_S^i) = c_V^j$. Then, the projection of the successor configuration of the next pagefault-free user mode step (of the specification machine) is equivalent to the next configuration of the virtual machine*

$$\Pi(c_S^{s_2(i)}) = c_V^{j+1}$$

where the function s_2 is defined in two steps:

1. *We define the function s_1 which returns for a cycle i either the same cycle i if no pagefault occurred or the first cycle after the return of the pagefault handler:*

$$s_1(i) = \begin{cases} i & \textit{if } \neg pff^i \wedge \neg pfls^i \\ \min\{j > i, mode^j\} & \textit{otherwise} \end{cases}$$

2. *Next, we define s_2 for a cycle i. If there is no pagefault in cycle i, we just increment i. Otherwise, we use the s_1 function to obtain the first cycle after the return of the pagefault handler. If in this cycle, there is no pagefault, we can again just increment its number. Otherwise, we at the cycle after still another execution of the pagefault handler and increment this.*

   In a compact, this can be written as follows:

$$s_2(i) = \begin{cases} i+1 & \textit{if } \neg pff^i \wedge \neg pfls^i \\ s_1(s_1(i))+1 & \textit{otherwise} \end{cases}$$

   Note that our pagefault handlers have the property (liveness!), that in cycle $s_2(i) - 1$ no pagefault occurs.

Proof. Mainly bookkeeping. Use translated accesses of the DLX_S if instruction / data in main memory. Otherwise, handlers swap the needed pages into the main memory.

Consider the example case that IR_V^j is neither load nor store. Consider the subcase that there is no pagefault on fetch, i.e. $\neg pff^i$.

First we verify that both instruction registers are equal:

$$\begin{aligned} IR_S^i &= mm_4^i(ma^i(DPC_S^i)) \\ &= vm^j(DPC_S^i) \text{ because } v_c(DPC_S^i) = 1 \\ &= vm^j(DPC_S^j) \\ &= IR_V^j \end{aligned}$$

By construction of the instruction set and the assumption of the lemma, we obtain that the updated registers are equal again:

$$\begin{aligned} R_S^{i+1} &= f_1(IR_S^i, R_S^i) \\ &= f_1(IR_V^j, R_V^j) \\ &= R_V^{j+1} \end{aligned}$$

Consider now the subcase that pagefault on fetch occurs at address DPC_S^i, i.e. pff^i holds. Validity of this case follows from the specification of the page fault handler. It guarantees that after its execution the faulting page is swapped in. Also, by the `rfe` mechanism, the PCs are restored on the return of the page fault handlers (page fault is of type repeat!). Therefore, we have cycle $s_1(i)$ a valid fetch page:

$$v^{s_1(i)}(DPC_S^{s_1(i)}) = v^{s_1(i)}(DPC_S^i) = 1$$

With this we can establish that both instruction registers are the same again:

$$\begin{aligned} IR_S^{s_1(i)} &= mm_4^{s_1(i)}(ma^{s_1(i)}(DPC_S^{s_1(i)})) \\ &= mm_4^{s_1(i)}(ma^{s_1(i)}(DPC_S^i) \\ &= sm_4^i(DPC_S^i) \text{ (correct swap-in)} \\ &= vm^j(DPC_V^j) \text{ (assumption of the lemma)} \end{aligned}$$

References

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness.* Springer, 2000.