50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Virtual Machine

Natalie Agus (Fall 2018)

1 MMU-Address Translation Example

Take a look at example below:

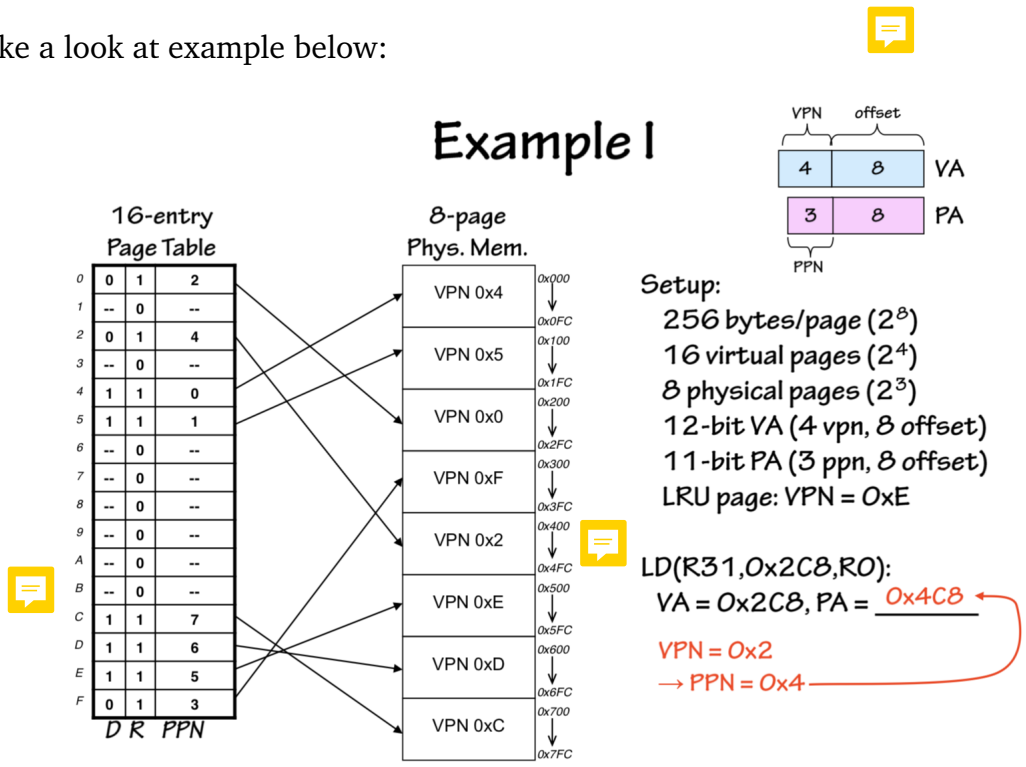

Figure 1

1. The CPU asks for VA 0x2C8 due to the LD instruction
2. Given a VA of 0x2C8, the VPN is 0x2 (4bits) and the PO is 0xC8 (8bits). According to the pagetable, a VPN of 2 gives a PPN of 4, hence the PA is the PPN and PO concatenated together: 0x4C8

3. Now consider another address request from CPU, for example, VA = 0x600
4. VA of 0x600 means VPN of 0x6 and PO of 0x00. From the pagetable, 0x6 VPN is not resident in RAM. Therefore it is **in the disk** (VA corresponds to address on disk)
5. However the RAM is full, so one has to replace a page in the RAM
6. According to the LRU, VPN = 0xE is the least recently used, this corresponds to PPN of 0x5. This corresponds to PA of 0x500 to 0x5FC (2^8 bytes per page)
7. Since D = 1 for this VPN = 0xE, one has to write the **content** in this PA 0x500 to 0x5FC to disk
8. Then mark the resident bit of VPN 0xE as 0
9. Next, the machine read page 0x600 (2^8 bits from 0x600) from disk into 0x500 to 0x5FC section in the RAM
10. And setup the page-map for VPN 0x6 = PPN 0x5 (not VPN 0xE)
11. Now the physical address of VA 0x600 is 0x500, but the **content** in this physical address has changed as from what it was in step 7 above.

2 Programs

In a computer, lots of programs can run at the same time. Each running program (a.k.a process) has a context. Context is a set of mapping from VA to PA as specified by the contents of the Pagetable in the MMU. Each process has its own context #. Benefit of having contexts:

1. Allow time sharing among programs
2. Separate context for each program (each program can be written without awareness of other programs)
3. The OS or kernel helps to manage each program, (e.g: P1, P2, and P3 in Fig. 1)
4. Each program has its own Virtual Memory, and therefore each program is an **instance of a Virtual Machine**

3 OS Multiplexing

The OS multiplex the CPU, so that it seemingly able to run multiple programs / processes in ”parallel”:

1. Look at Fig. 2, at first P1’s first task is running

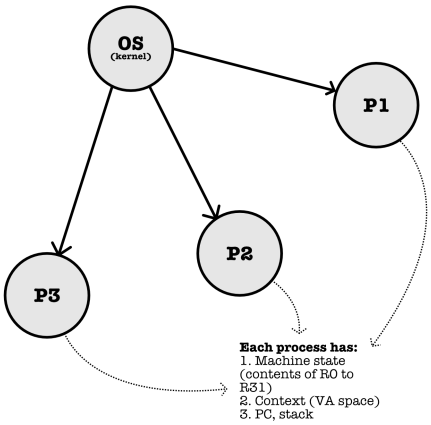

Figure 2

2. Then after its done, the OS interrupts, saves all the states of P1 to the Memory
3. The OS then load the states of P2 to the registers and
4. Let P2 runs another task
5. After P2 finishes running a task, another interrupt occurs,
6. The OS saves all the states of P2 to the memory, and load back all P1 states
7. Let P1 resumes the second task, and repeat Step 2.

4 The Interrupt Hardware

`IRQ`: the interrupt signal, typically triggered by a timer.

```
If(IRQ==1 && PC31 == 0)
     Reg[XP] <- PC + 4
     PC <- Xaddr
```

Pc will execute the interrupt handler at 'Xaddr' whenever the IRQ signal is 1.

4.1 Beta IRQ Handling

1. Beta always check for `IRQ` before each instruction fetch
2. On `IRQ`(j) (interrupt when running process j):
    - Stop process j
    - Copy PC+4 to XP

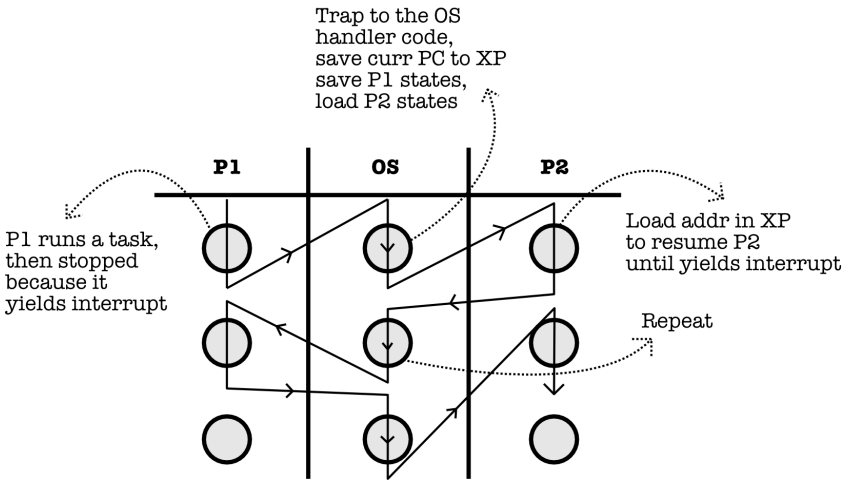

Figure 3

- Switch PC to ”Xaddr”

3. What is ”Xaddr”:
    - The address of the handler code, it first saves the user states in the corresponding j^{th} cell of the proctable.
    - Call the process to handle exception
    - After the process handling exception returns,
    - Reinstall the saved state from user
    - Return PC to Reg[XP]-4

4.2 Handlers Location

Handlers are located in the lower memory address:

- RESET in 0x8000 0000
- ILLOP in 0x8000 0004
- X-ADR in 0x8000 0008

4.3 Timer Example

Consider a case where the OS maintains the Time of Day (ToD), updated periodically by 60 Hz timer **hardware**. Program A doesnt know about CLK hardware. It can ask kernel for ToD in the code. The kernel manages the ToD.

1. So at first, Program A executes its instructions as per normal.

2. Then, when its time for the hardware timer to trigger ToD increase, it interrupts program A (set IRQ = 1)
3. The Beta recognize the IRQ signal, and stores the PC+4 of Program A to XP, and jump to the timer IRQ handler:

```
clock_h : ST(R0, user)
          ST(R1, user+4)
          ....
          ST(R30, user+30*4) //save states of user

          CMOVE(kstack, SP) //use kernel stack
          BR(clock_handler, LP) // branch to the clock handler
                                              code that can
                                            increment the ToD

          LD(user,R0)
          ...
          LD(user+30*4, R30) //restore user states
          SUBC(XP, 4, XP) //to re execute the line of program A that
                                    was interrupted by timer
          JMP(XP) //back to resuming Program A
```

4. Note that in the code above, ”user” is basically a memory location in the RAM to store the user states

5 OS Kernel Timesharing Scheduler

The timesharing scheduler in the OS switch between processes to run:

1. Figure 3 shows a scheduler that is running P2
2. When the OS wants to interrupt P2 and run P3,
3. It saves all the user states in the proctable,
4. Paste the user state from P3 to the registers
5. Resume P3 operation by JMP(XP-4)

What if there’s another interrupt call in the middle of the OS Kernel saving states in step 3 above?

- The setup procedure NEVER interrupts a kernel (the OS is written as such)
- Indicated by the MSB of a PC:
  - If 0, PC in user mode, allow interrupt

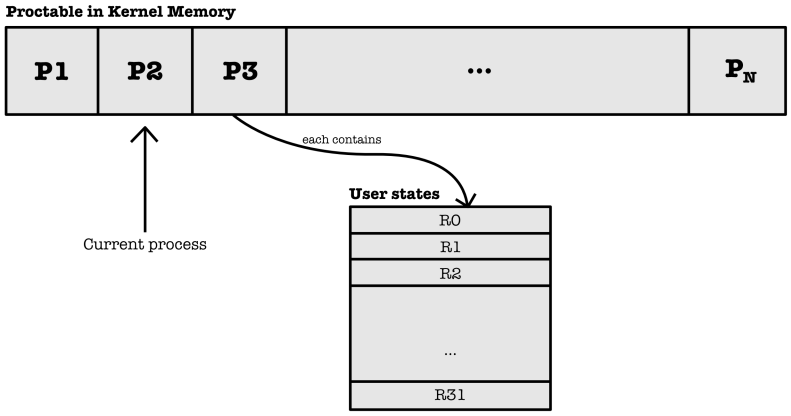

Figure 4

  - if 1, PC is kernel mode, disallow any interrupt

- Therefore IRQ is simply NOT ALLOWED until the PC's MSB is back to 0
- One has to be **very careful to never run into infinite loop in kernel**. If this happens, computer freezes, and you have to unplug the computer.

6 Programs Communicating with OS

This section explains how the program communicates with the OS: **known as supervisor call (SVC)**. You can call the OS from the code by executing a particular **ILLOP** (bad opcode). One can purposely cause an exception, then OS code will recognise this particular ILLOP as user-mode supervisor call. The procedure for SVC is that, the user has to provide arguments in the registers and the OS will return the result in R0. One scenario where the user needs to communicate with the OS is when asking for keyboard / mouse input, since the I/O is controlled by the kernel.

7 Detailed Scenarios of Cache + TLB misses and hits for *Case2: Cache after MMU*

7.1 Case 1: TLB miss, Pagetable miss, cache miss

In this case, the scenario is:

1. Given a VA + Context from CPU, look for translation in TLB, results in miss
2. Hence look for translation in Pagetable, results in miss
3. This means that item is **on disk**
4. Find LRU from pagetable, check if D==1, write to disk
5. Replace the LRU page in RAM with the new page **from disk**
6. Update the corresponding TLB so that the VPN points to this new page PPN in RAM
7. Update the cache (see cache algorithm notes, need to check for LRU if FA or NW is used, and check for dirty bit before replacing, if dirty, write to RAM first before replacing the cache). **New extension to cache algorithm: If RAM is full**,
    - Repeat step 4
    - Mark the corresponding VPN of that LRU page as **R == 0** (non resident)
    - Change the VPN of that LRU page to be **the same as the address of that LRU page on disk**
    - Then, replace that LRU page in the RAM with the LRU in the cache (if FA / NW), or with whatever overwritten page in DM
8. Return data to CPU
9. Figure 5 basically shows that the VA goes to the disk because everything else results in misses

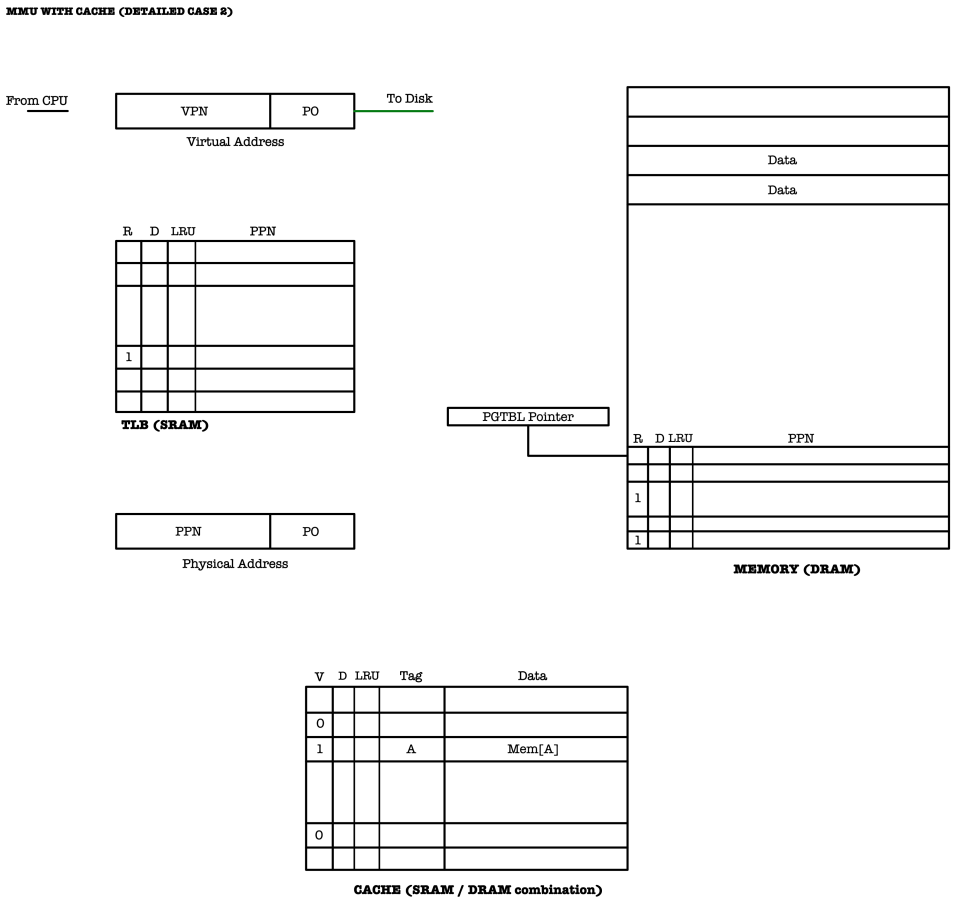

Figure 5

7.2 Case 2: TLB hit, Cache hit

In this case, the scenario is:

1. Given a VA + Context from CPU, look for translation in TLB, results in hit, one can obtain the PPN
2. Look for the PPN+PO = PA in the cache, results in hit
3. Return data to CPU
4. Figure 6 shows the scenario

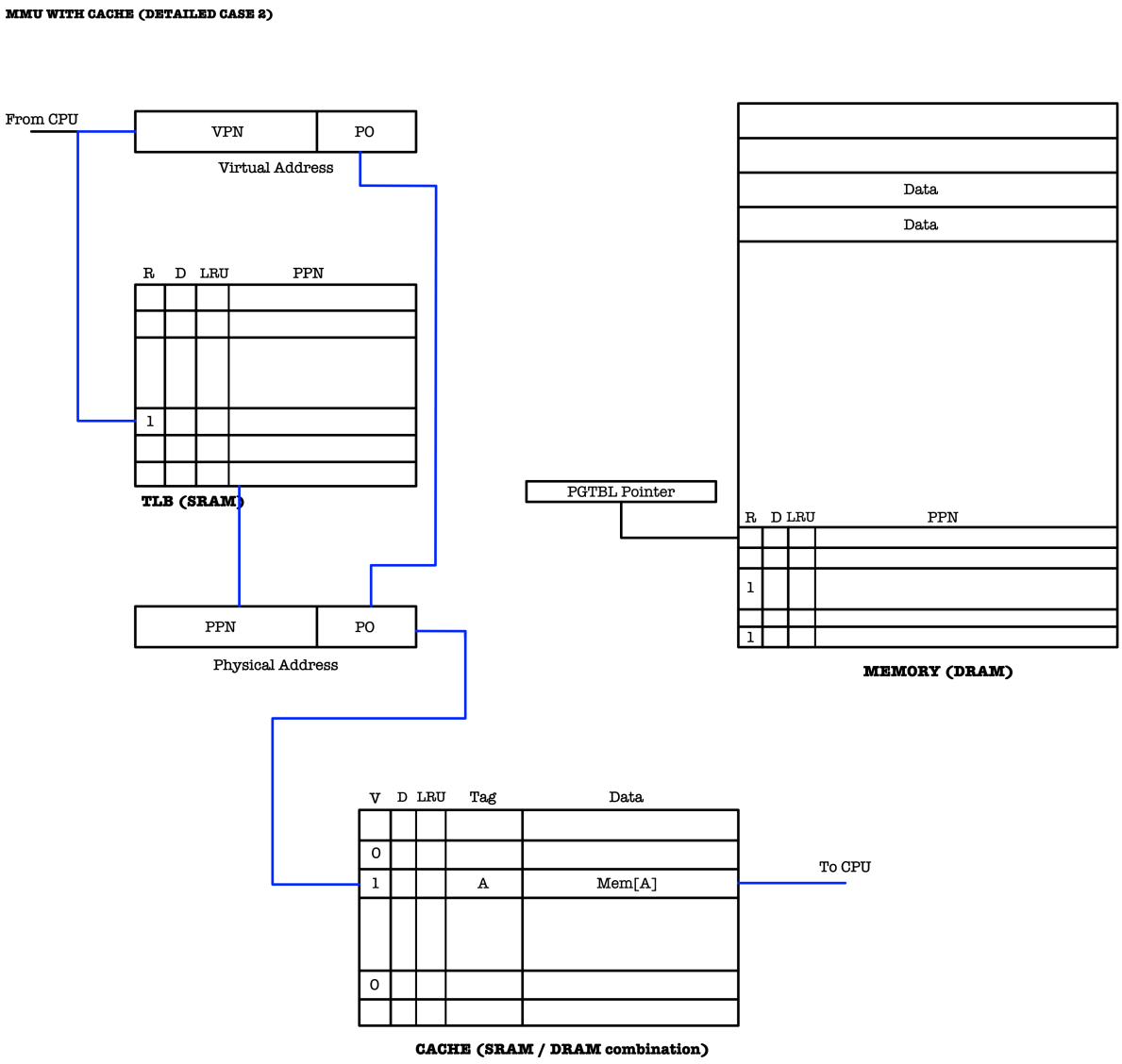

Figure 6

7.3 Case 3: TLB hit, Cache miss

In this case, the scenario is:

1. Given a VA + Context from CPU, look for translation in TLB, results in hit
2. Look for the PPN+PO = PA in the cache, results in miss
3. Look for the PA in RAM, update cache accordingly (see cache algorithm)
4. Return data to CPU
5. Figure 7 shows the scenario

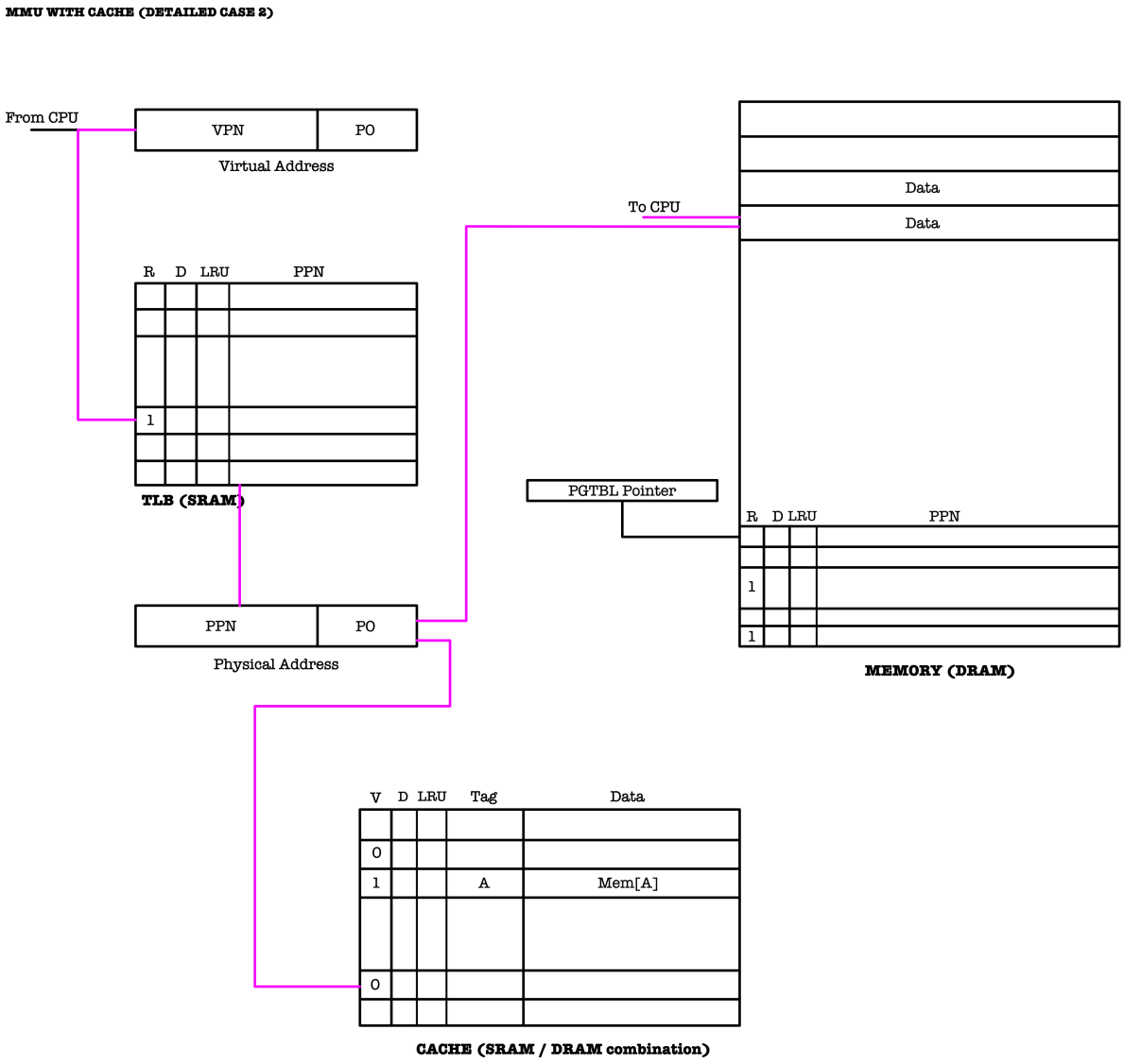

Figure 7

7.4 Case 4: TLB miss, Pagetable hit, Cache miss

In this case, the scenario is:

1. Given a VA + Context from CPU, look for translation in TLB, results in miss
2. Look for translation in pagetable, PPN + PO = PA is obtained
3. Look for the PA in cache, results in miss,
4. Look for the PA in RAM, update cache accordingly (see cache algorithm)
5. Return data to CPU
6. Figure 8 shows the scenario

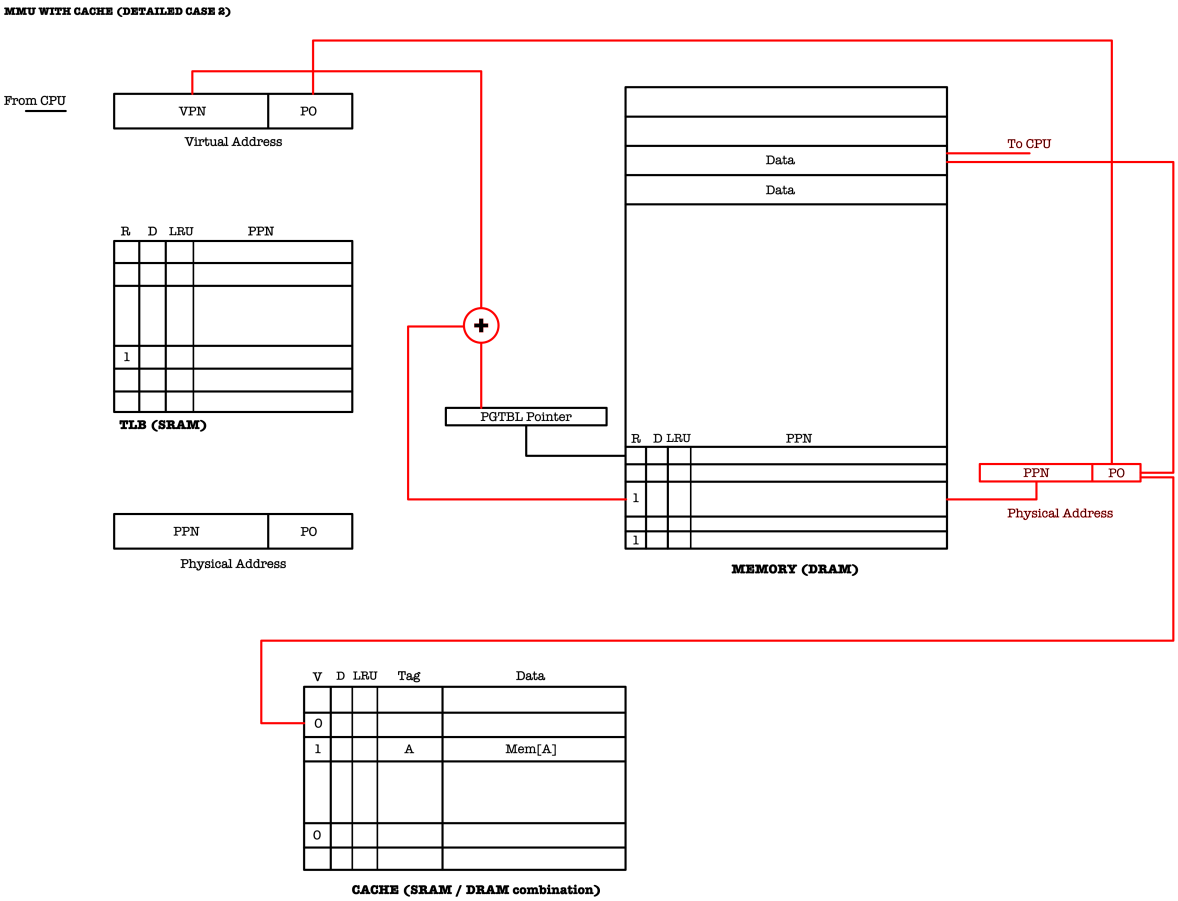
MMU WITH CACHE (DETAILED CASE 2)
From CPU
VPN
PO
Virtual Address
R
D
LRU
PPN
1
TLB (SRAM)
PPN
PO
Physical Address
PGTBL Pointer
Data
Data
To CPU
R
D
LRU
PPN
1
1
MEMORY (DRAM)
PPN
PO
Physical Address
V
D
LRU
Tag
Data
0
1
A
Mem[A]
0
CACHE (SRAM / DRAM combination)

Figure 8