COMP 273 Study guide

Francis Piche

April 19, 2018

Contents

Part I
Preliminaries

1 Introduction

This guide is based off of the lectures and slides by Professor Joseph Vybihal at McGill University, Winter 2018. Images are taken from Prof. Vybihal's lecture slides.

This guide is my best attempt to express the ideas of the course in a clear and concise way, given that there is less than a week until the final. That being said, I'll work backwards, starting from the latest material (and in my opinion the hardest) to ensure that the most difficult material will get covered in time for the exam.

2 System Board

3 Number Formats

Part II
Basic Circuits

4 Intro to Circuits

5 RAM & Registers

6 Arithmetic Logic Unit

7 Control Unit

8 Classical & MIPS Pipeline CPU

Part III
Assembler Programming

9 Basic Assembler

10 Complex Data

11 Recursion, Stack & Heap

Part IV
Advanced Circuitry

12 Polling & Interrupts

12.1 Peripheral Devices

Peripheral are external devices such as keyboards, mice, screens, and network adapters.

The devices each have a **controller** (simple CPU).
There are two main types of controllers:

- On-Board: controlling registers are integrated into the system board
- External: controlling registers are part of the card plugged into slots on the system board. These slots are connected to the bus.

A controller chip is made up of:

- Status register (ready, on/off, error-codes)
- Data register (information to be processed)
- Command register (in binary)
- ROM (to hold basic information)

Often the registers are combined into one.

Note that integrated, on-board controllers are faster, since they skip the slot. Their registers are directly connected to the bus and have addresses.

Also note that if we don't look at registers before the next key press, the data is lost.

12.2 IO & Communication

There are two main techniques for communication:

- Interrupt Driven: Device signals the CPU when state changes.
- Polling Driven: CPU looks at the device's status register

Within each of these techniques, there are two ways to exchange data:

- Synchronous I/O: The CPU monitors the device, sending and reading byte by byte
- Asynchronous I/O: CPU signals when to start, device signals back when finished.

Asynchronous is accomplished by this process: first the CPU loads into registers the start address, limit, and command. The CPU is then free to do anything else until the device sends an interrupt to signal that the task is complete.

The registers are accessed either using a general data path (for example the RAM zero page via the bus), or by using a specialized path such as a DMA or interrupt wire.

12.3 Memory Mapping

There is a special portion of RAM directly allocated to peripheral registers, called the zero page. This mean these special addresses are actually wired to go to the peripheral registers.

12.4 Polling

This is basically accomplished using a busy loop.

```
while(status != 0); // assume 0 means it's ready
```

Now in assembler:

```
LOOP:
  lw $t0, STATUS
  bne $t0, 0, LOOP
#else check the flags and handle
```

Problem is that it uses 100% CPU capacity.

12.5 MIPS I/O Communication

MARS is only able to do simulation for the keyboard and screen (text).

The keyboard is commonly referred to as the **receiver**, and the screen the **transmitter**. The receiver control register's address is at 0xffff0000, followed by the receiver data at 0xffff0004, transmitter control at 0xffff0008, and finally the transmitter data at 0xffff000c.

The first bit of the control registers is the "isReady" bit, 1 meaning ready, 0 meaning not ready.

The second bit of the control registers is to control whether the device is allowed to send an interrupt or not. 1 for yes, 0 for the default (depends on the machine).

12.5.1 GETCHAR and PUTCHAR functions

```
GETCHAR:
  lui $a3, 0xffff       #load address of control register

ISREADY:
  lw $t1, 0($a3)      #read from control register
  andi $t1, 0($t1), 1   #check if first bit is zero
  beqz $1, ISREADY    #if yes then check again
  lw $v0, 4($a3)      #if 1, then load the contents of data bit into v0
  jr $ra              #return

PUTCHAR:
  lui $a3, 0xffff
CHECK:
  lw $t1, 8($a3)      #check the transmitter this time
  andi $t1, $t1, 1    #check if ready
```

```
beqz $t1, CHECK       #if 0 then try again
sw $a0, 12($a3)       #if 1 then send character to data register of
    device
jr $ra
```

Notice the similarities and differences between the two programs. In getChar, we load FROM the data register to GET the character, whereas in putChar we need to load TO the data register to PUT the character. Notice the offsets when accessing the base address.

There was also the process of using logical AND to extract a bit. This is called **bit masking**. When an AND is performed between an unknown sequence of bits and all 1's, the result will be whatever the unknown sequence was!

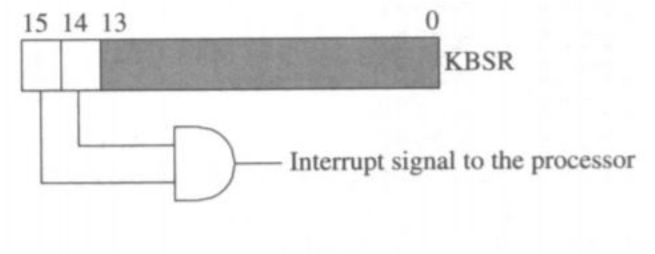
12.6 Interrupts

First, some definitions:

- Exception: Any event that stops the normal execution of the CPU. For example,stopping the program due to divide by zero, stack overflow etc.
- Interrupt: There are two kinds of interrupt: Signal, an event purposely triggered by a program to re-route the CPU flow to another process (think `throw` in Java), and trap, an event purposely triggered by a device.

So just remember: signal interrupt = program throw, trap = device throw. The difference between exceptions and interrupts is that the former is to handle instruction faults (division by zero, undefined op-code, etc) while the latter is to hand external events.

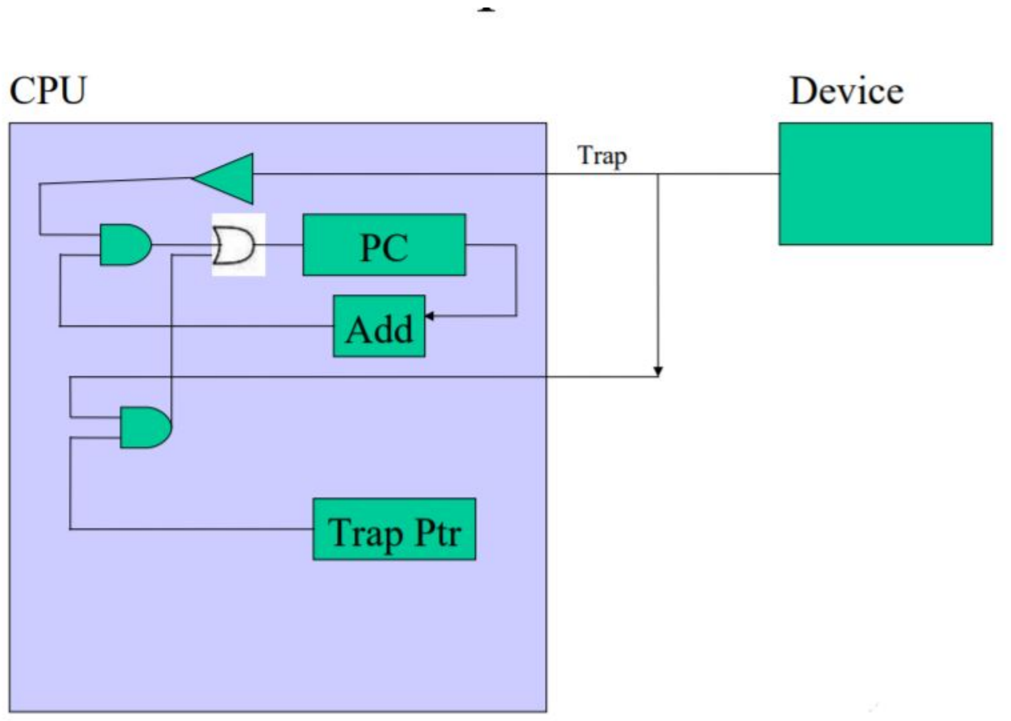
This is a picture of what an interrupt would look like on the register's end:



So basically there's an AND gate on the last two bits. One of the bits is the "enable interrupts bit" and the other is the "interrupt bit". The former is turned to 1 by the programmer (or by default), and then when an interrupt is to occur, the other bit will turn to 1, allowing the interrupt signal to flow out the AND gate.

12.6.1 Implementation of Interrupt

The hardware view of this is a bit strange:



What happens here, is that when the device is not sending a trap signal (like in the previous picture), the PC can increment happily as normal. When the trap signal is turned on, this causes the PC to "jump" to the location of the "trap pointer", causing a halt of the normal program execution. Note that the AND/OR gates are multi-bit.

12.7 The Exception Co-Processor

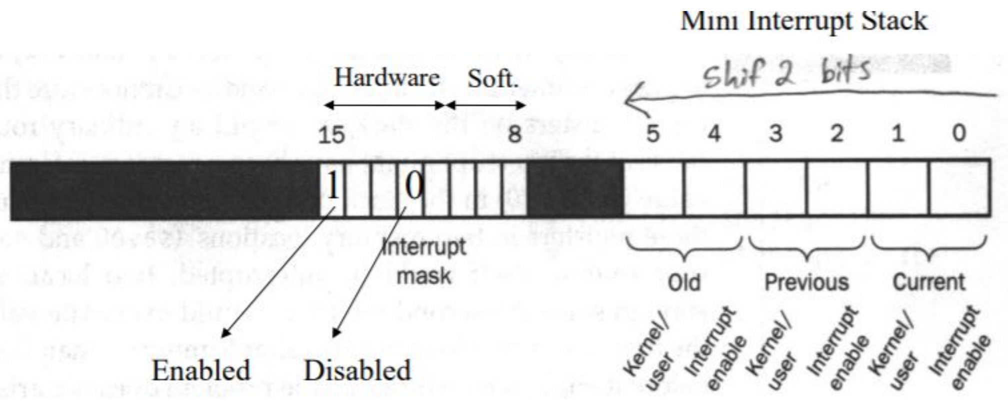
This is co-processor 0, (recall each co-processor is identified by a number). This co-processor contains:

- Error PC: Contains the address of where the exception occurred (which instruction).
- Cause Register: Contains information about the exception type, and what may have been the cause.
- BadVAddr: Has the address that cause the bad memory reference (the non-existent address that caused the error)

- Status Register: More on this below:

12.7.1 Status & Cause Registers

The status register contains the **interrupt mask**, which is used to check which devices have interrupt enabled/disabled. It also contains a mini "stack", where it holds some information about the interrupt.



The interrupt stack can hold information about 3 interrupts. This information is whether the interrupt occurred while in kernel mode (for privileged, low-level stuff), or in user mode, and it contains a bet for whether interrupt was enabled for that device. Every time an interrupt occurs, the "stack" shifts left two bits, losing the old data.

The cause register contains pending interrupts (interrupts that haven't been processed yet), and some exception code. The exception code is a binary encoding for what kind of exception occurred.

12.8 Where Interrupts Go

All interrupts get routed to a special kernel address called the **iterrupt handler** that processes the interrupts. Programmers can put their own code in this location to change what happens when an interrupt occurs.

This interrupt handling code is generally just a switch statement that handles depending on the cause found in the cause register.

13 Cache & Performance

In computing, we often need large amounts of storage, and it needs to be accessed very quickly. The issue is that large storage and speed usually conflict.

<u>In terms of size</u> : Disk > RAM > cache > registers.
<u>In terms of speed</u>: Disk < RAM < cache < registers.

(note that $a > b$ implies a is better than b) So cache is a necessary part of fast computation.

This comes with some issues:

- Cache is generally smaller than a program
- Most instructions are stored in RAM.

13.1 Cache-Loading

13.1.1 Locality & Measurement

In programs, typically items are referenced several times. (ie, you call printf() several times, recursive functions etc.). This idea is called **temporal locality**. And typically adjacent items are executed in sequence (loops, functions etc). This is called **spatial locality**

When trying to optimize cache-loading, this idea of locality will help a lot.

The measure of how good or bad a loading method is, is the **hit-to-miss-ratio** or "cache miss rate". A hit means that we found the instruction in cache when we needed it, and a miss means we had to go to RAM to find it.

We associate a **cost** to missing and needing to refill the cache, since it takes time.

13.1.2 Wide-Bus Method

Often a wide bus is used to load data into cache. This works by: whenever we need to go into RAM to get an instruction, we just load that instruction, plus the next 16 bytes after it, in hopes that we'll need them soon (taking advantage of locality).

This takes one clock cycle to send the address to RAM, 1 cycle to find the block of data in RAM, and 1 cycle to send the data back to the cache. So the cost of missing with this method is 3. But that's the cost of loading a single byte anyway, so if we use the others that we loaded, we save lots of time.

13.2 Handling Misses

When the IR tries to receive the missed instruction from cache[PC], the MDR loads from RAM[PC]. (takes a few ticks). Then the cache[PC] is updated from the MDR, and we start the instruction over again.

Now a few definitions:

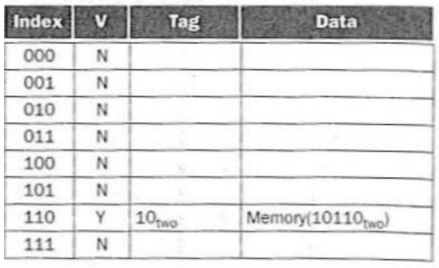
- Miss penalty = cycles to upload data into cache
- Cost of missing = miss frequency * penalty
- Program speed = n + m*penalty (n = number of instructions, m = number of instructions that miss)

13.3 Cache Addressing

Cache is smaller than RAM, so we need a way of shrinking the addresses down. This is done using modulo! In binary, using modulo is the same as just chopping off the last bits. For example $(1011)_2 \bmod (100)_2 = (11)_2$, which is the same as if we had just cut off the first 10. In hardware this is done by just grounding wires.

So all addresses in RAM ending in, say 001, so 000001, 101001, 111001 etc would all map to the address 001 in cache!

This results in some overlaps. These are handled by cache having this structure:

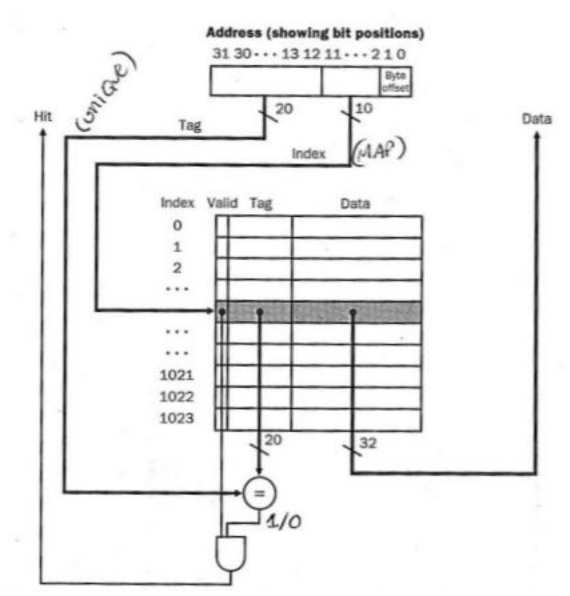


Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

The index is what was mentioned above, the "cache address", or the last bits of the RAM address.

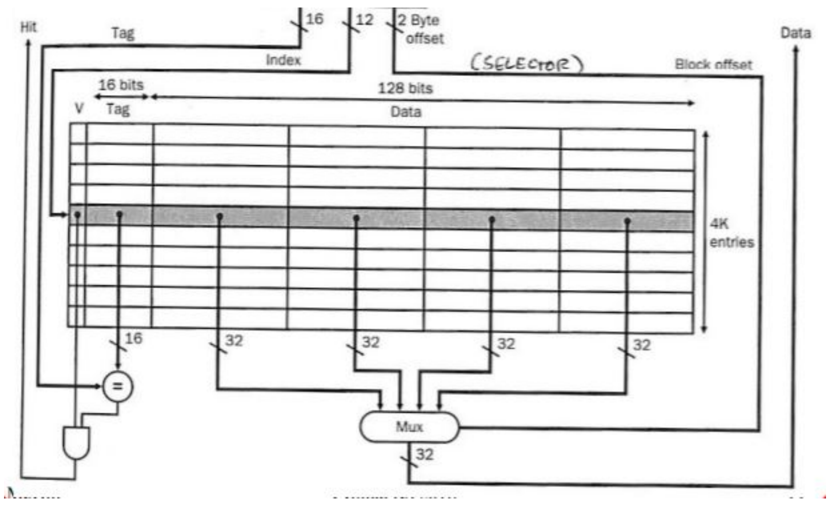
The column V keeps track of whether that table entry is valid, meaning whether it has any data in it or not. (or else how would we distinguish 00000 (not initialized), from 00000 (the number zero)?).

The tag column keeps track of the bits that we "cut-off", as a unique identifier of the address we came from. This helps deal with overlaps



So when we try to access the data at this index, we check to see if the tag matches the first (usually 20 or 16) bits that were cut from the RAM address. If it does, and the "is-Valid" column reads 1, then we know the data is correct, and we have a hit. If not, then we missed. Maybe the data was overwritten by an overlap, maybe it was never there to begin with.

In reality, the data is 128 bits, to account for blocks of data. (4 instructions) and then to access the particular instructions, there is a two bit offset to select which one you want. (Using a multiplexer)



So all 4 instructions are fed simultaneously into a multiplexer, and then the offset selects the instruction.

13.4 Performance

13.4.1 Amdahl's Law

The formula for calculating how much a new component will speed up your computer is:

$$s = \frac{1}{(1-f)+\frac{f}{k}}$$

where s is the resulting speed increase, f is the fraction of work the part will do, and k is the advertised speed-up of the new component.

13.4.2 Examples

Problem 1. *Assume your daytime processes spend 70% of their time running in the CPU and 30% waiting for service from a disk. You find a computer that functions 50% faster and costs $10,000. You also find a new disk drive for $7000 with a speed increase of 2.5 times. What do you do?*

Solution. First we look at the speed increase for the faster CPU. Plugging into the formula for $f = 0.7$, $k = 1.5$:

$$\begin{aligned} s &= \frac{1}{(1-0.7)+\frac{0.7}{1.5}} \\ &= 1.3 \end{aligned}$$

So we get a 30% boost from the new CPU. But for 10000$, this is $\frac{10000}{30} = 333\$$ per percent increase.

We do exactly the same for the new hard-drive, and find that we have a 22% speed boost at 318$ per percentage.

So the new CPU gives a bigger overall increase but the new drive is more cost-effective.

Problem 2. *Assume polling takes 400 ticks on a CPU that runs at 500MHz. How much CPU time is used to poll a mouse 30 times per second?*

Solution. So here there's no formula, just use some math and unit conversions..

$$\frac{400tick}{poll} * \frac{30poll}{second} = 12000\frac{tick}{second}$$

$$\begin{aligned} \Rightarrow &\frac{12000\frac{tick}{second}}{500,000,000Hz} \\ &= 0.000024\% \end{aligned}$$

Problem 3. *How much CPU time used if a floppy disk data transfer rate is 16 bits per tick and needs to move data at a rate of 50KB/sec?*

Solution.

$$\frac{16bits}{tick} = 2\frac{bytes}{tick}$$

$$\Rightarrow \frac{50,0000\frac{bytes}{second}}{2\frac{bytes}{tick}} * 400ticks = 10,000,000\frac{ticks}{second}$$

$$\Rightarrow \frac{10,000,000\frac{tick}{second}}{500,000,000Hz}$$

$$= 0.20\%$$

All the problems on the slides are similar. Make sure you can do all of them without looking at the answers!

14 Virtual Memory & Performance

14.1 Background

Virtual memory (VM) is the idea of using the hard-disk as imaginary RAM, so that our programs can be large.

So when we write programs, and we access say address 010010101, this address is "fake". The true address in RAM is something else. When the program runs, it swaps pieces of our program from the disk into RAM to be executed.

In general, out programs execute function by function, as seen in the previous cache section. VM takes advantage of this by breaking up the program into "functional units" things like loops and functions, which will (hopefully) be executed many times.

Since we as programmers are dealing with "fake" addresses, we need a FAST way to convert to real addresses in RAM, and keep track of where things are. (more on that later)

14.2 VM Steps

- Step 1: Open the program
- Step 2: OS copies the program into disk, (VM), and chops the program into functional units
- Step 3: The OS loads one functional unit into RAM (if no space, remove another unit).
- Step 4: Execute that functional unit.

14.3 Memory Management

Theres a few types of management for the VM:

- Do nothing, let chaos reign on the world.
- None, but managed partially by the programmer.
- Compiler managed
- OS managed (with page swapping & VM)

This course only looks at the last one.

14.3.1 Page vs Frame

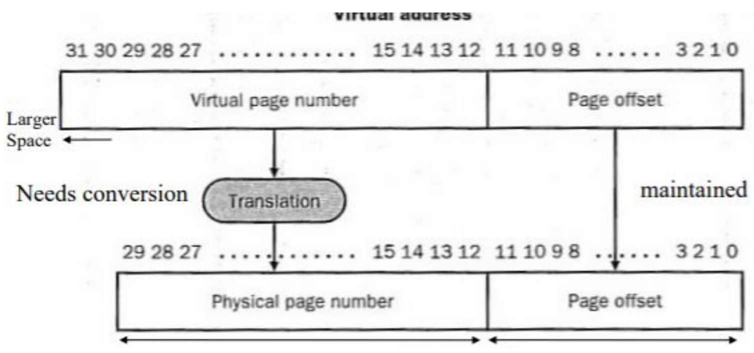
A **page** is the abstract idea of a "functional unit". It has varying size (like functions, while loops etc).

A **frame**, on the other hand, is a physical, fixed-size space of RAM. You can picture RAM being divided into an array-like structure where each frame has an index from 0 to n.

14.3.2 Page Loading

A **page fault** is similar to the idea of a "miss" in cache. It is when the page is not found in RAM. Then we need to load from the disk (super super slow).

A **dirty page** is a page in RAM who's data has been selected to be overwritten. To deal with it, we need to save this page back into the disk, and read in the new page.



Mapping from VM to RAM is somewhat similar to mapping from RAM to cache. Mod is used to shrink the number of addresses down to a smaller amount, and a "page offset" is maintained to keep track of where you are in the page.

14.3.3 Types of VM

Paging: the overlays have the same size, overlay matches the framesize, and the addressing is done simply by concatenating the page number with the offset:

Page #	Offset

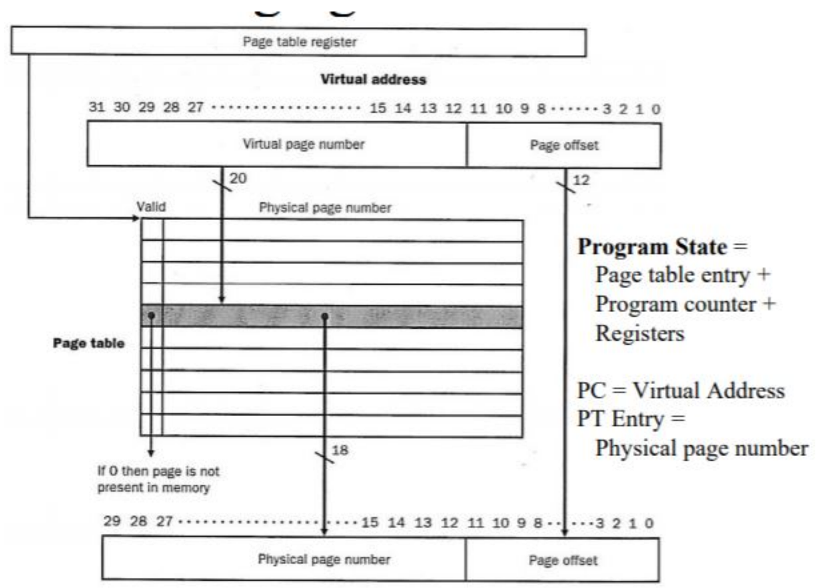
The other way is by segmentation, where there is variable sized overlays, (multiples of the frame size), and addressing is done by additionally concatenating the segment number (since multiples of the frame size, need to know which part to go to)

Segment #	Page #	Offset

14.4 VM Hardware

The **page table** is used to keep track of the addresses of pages.

Similar to cache, the logic is to have an isValid column, and a ”tag”, which retains the first bits of the virtual page number. And we also keep the offset so that we can keep track of which instruction in the page we want.



this page table lives in the ”reserved” part of RAM. (recall that RAM has an OS section, zero page and program section). This linked list keeps track of the virtual address number.

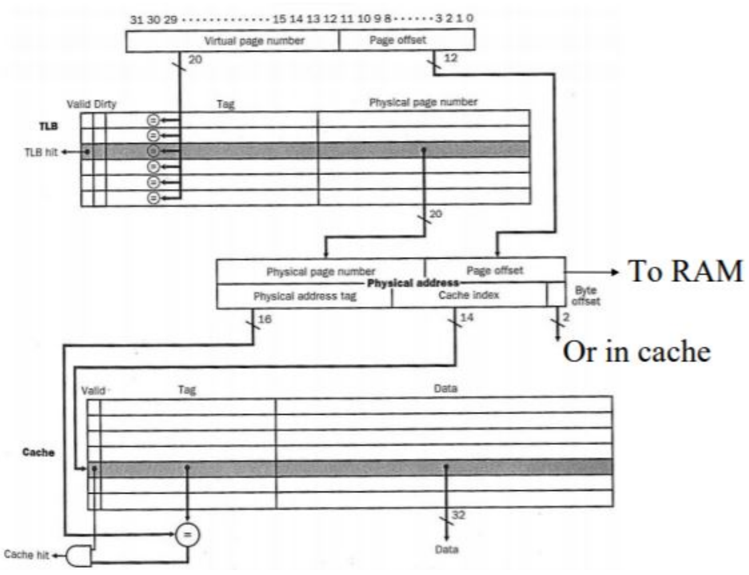
Of course, going to RAM constantly is slow, so we need a faster way to do the conversion.

14.5 Fast Address Translation: The TLB

The TLB, or Translation Lookaside Buffer, is a special cache that keeps a small subset of the page table available within the CPU for fast access.

It works the same way as a regular cache, but with some added features. It still has the isValid column, but also has an isDirty column, to keep track of which pages are dirty (need to be replaced). It still has a tag which needs to be checked for overlaps, and it keeps the physical page number as it's data.

It is implemented like this:



Essentially whats happening is that in the middle, depending on what we want to do, we either put instructions into cache or into ram. (This picture includes the full cache picture at the bottom). The virtual address at the top is actually the PC. So when the PC increments, hopefully it will match an address in the TLB, and then the address can flow and get checked against cache to see if the data is in there.