COMP 273 Study guide

Francis Piche

April 23, 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

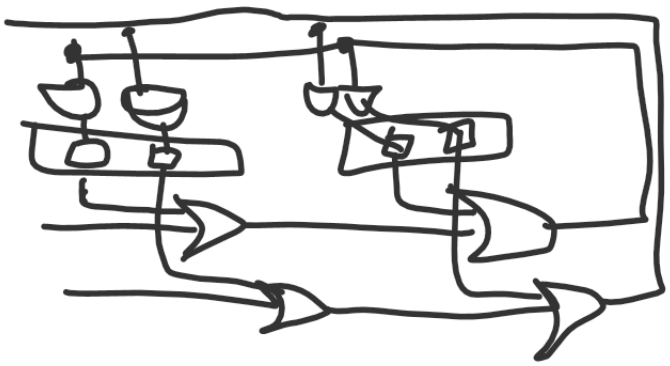
4 RAM & Registers

4.1 Registers

A register is defined as a self-contained unit storing a single data value.

A register is the fastest memory in the computer. It uses D-Flip flops as the bits since d-flip flops always output the bit being stored.

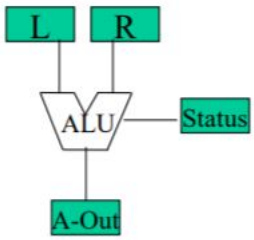
The CU is connected to every register in the CPU to allow reading/writing. (connected on both sides of the register).



As seen above, a U-bus is used to connect registers together, where the OR gates are used to merge to the bus, and AND gates control data coming in.

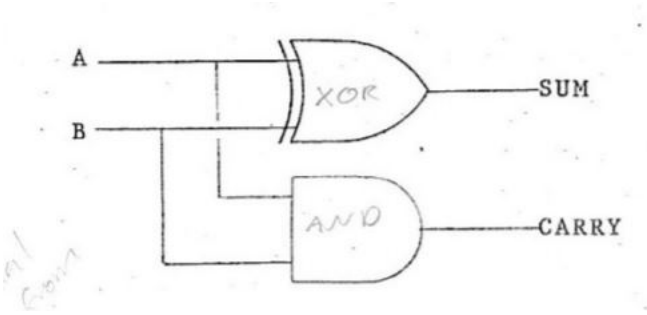
5 Arithmetic Logic Unit

The ALU performs mathematical operations. Special hardware needed for multiplication and division, and also floating point. The ALU does things like add, subtract, greater than, less than, logical AND OR etc.



The A-Out register holds the result of th operation, and the Status register keeps information such as: was the result negative? Zero? Did an overflow occur?

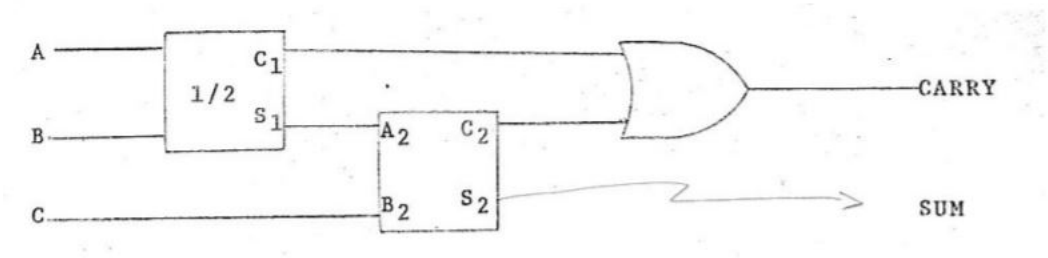
5.1 Half Adder



The half adder is used to add 1 bit operands. Note that when performing addition by hand, a carry occurs if the sum exceeds the base for that position. For example: $(9+2)_10$ has a carry of 1. The same applies in binary, so $(1+1)_2 = (10)_2$ or, 0 with a carry of 1. So the XOR gate will take care of the adding, and the AND will take care of the carry.

5.2 Full Adder

To perform multi-bit addition we will need a full-adder. This allows for a "carry-in", meaning it will add the carry of the previous bit along with the current two bits being added.



Here the two boxes are just half adders. If the sum of the addition, or the sume of the addition added to the carry of the last addition results in a carry, then the carry will continue onward.

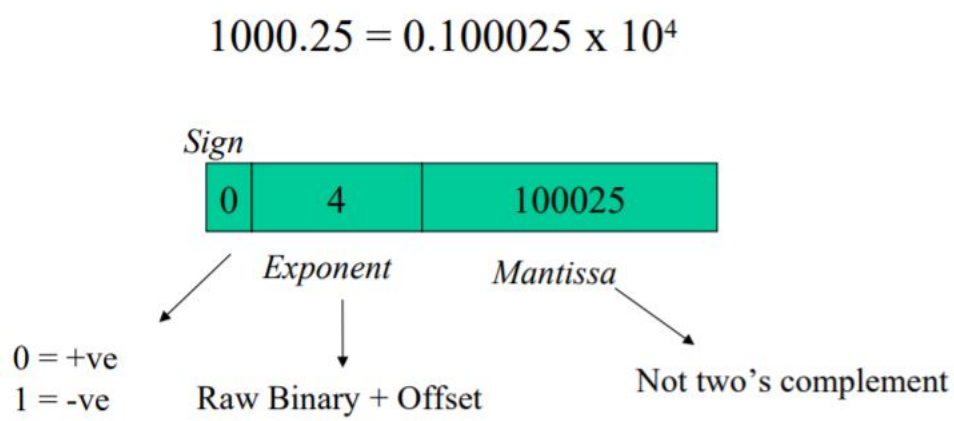
Performing multi-bit addition is just a matter of chaining together these full adders (like in assignment 2).

5.3 Floating Point Arithmetic

A floating point number can be broken down into a few pieces. First, recall that in scientific notation, we can make all numbers have the following form:

$$1.ax10^n$$

Well the same is true in binary. The way this is represented is:



Notice that there is no reason to keep the leading 1, and the decimal point, since they will be the same for any number. And there is also a sign bit for the exponent, since the exponent can be negative.

The way the exponent works is with a bias. For single precision, this bias is 127, for double it's 1023. Going below the bias gives negative exponents, going above gives positive. For example: 126 = -1, 127 =0, 128 = 1.

5.3.1 Converting Decimal To Binary Floating Point

To convert from decimal to binary floating point, we need to break up the number into two pieces, pre and post- decimal. The pre-decimal part can be converted however you choose. The post-decimal part needs some new techniques.

Its best shown through example.

Problem 1. *Convert -69.125 to binary.*

Solution. First, $(69)_10 = (1000101)_2$. Secondly, we 'll try to break down the part after the decimal to a fraction of a power of 2.

In particular:

$$0.125 = \frac{1}{8} = \frac{1}{2^3} = \frac{1}{(1000)_2} = 0.001_2$$

Now we have: 69.125 = 1000101.001, and to turn this into scientific notation, we need to multiply by the appropriate powers of 2. (Like in base 10 where we multiply by powers of 10).

$$= 1.000101001 * 2^6$$

So our exponent is 127 + 6 = 133 = 10000101

The sign bit is 1, and dropping the initial 1. all together we have:

$$11000010100010001010 01$$

which extended to 32 bits is:

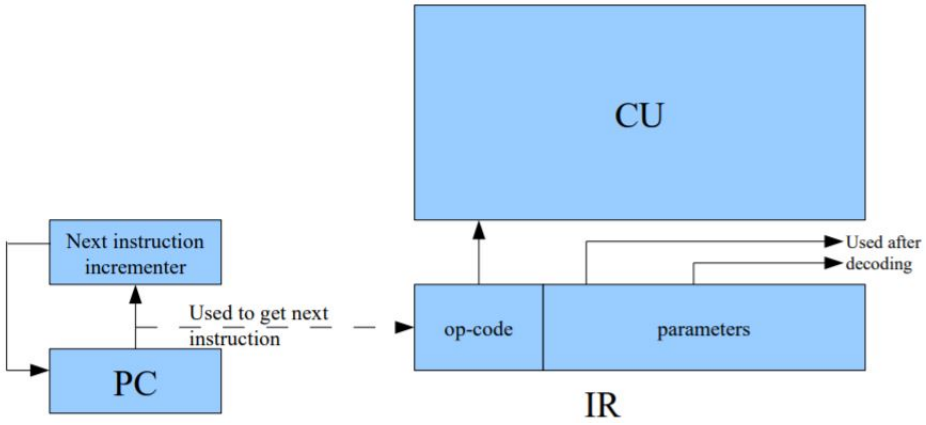
$$11000010100010001010010000000000$$

5.3.2 Addition and Multiplication of Real Numbers

For addition, we need to first convert to our IEEE format like above, then make the exponents match by shifting on of the numbers by powers of 2. Next we can add the mantissa's the usual way.

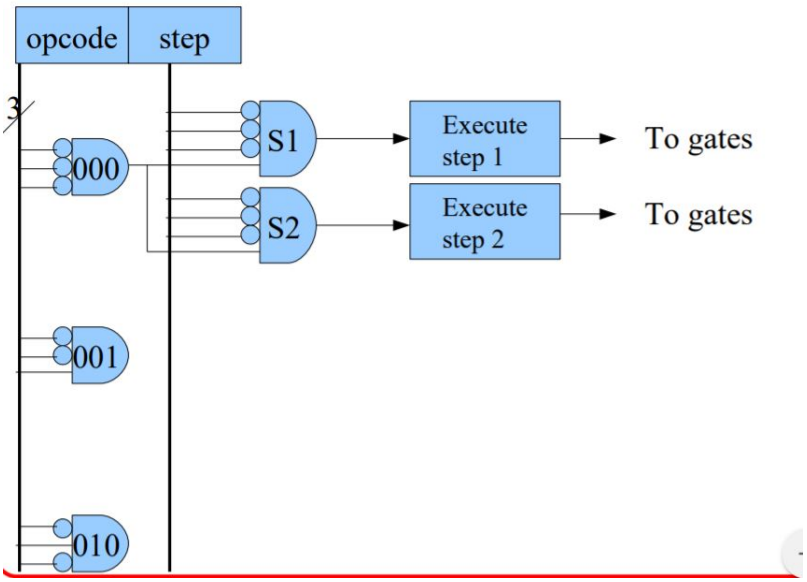
For multiplication, we need to add the exponents, and then multiply the non-exponent parts.

6 Control Unit



The CU or control unit is in charge of deciding the exact steps that will be executed for each instruction. The PC simply points to instructions, which are sent to the IR. The CU takes in an op-code and parameters, from the IR.

Once inside, the instructions are broken down into the actual steps.

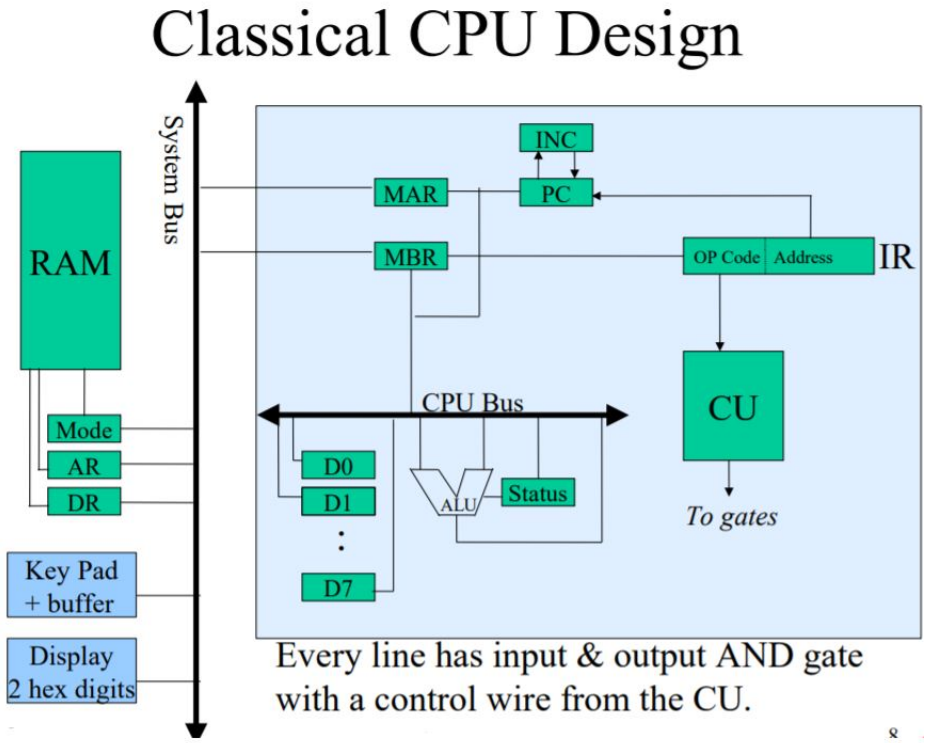


Here the step is just being incremented in a loop so that the micro-instructions (the pieces that make up the overall instruction, ie: "ADD A + B" is really: "load A into L, load B into R, ADD, store result" etc).

All micro-instructions are 1 tick, some of which might be "stalls". Stalls meaning do nothing. We might want to do nothing if our instruction only takes 3 ticks, since the longest instruction takes 4, and all need to be the same length for timing.

7 Classical & MIPS Pipeline CPU

7.1 Classical CPU



In this schematic, the execution cycle, like the pipeline is broken down into 4 stages.

First the fetch cycle:

- PC increments and goes into the MAR
- MAR goes into RAM[AR] to read instruction
- RAM[DR] gives response to MBR, then goes to IR

(Have to go to RAM every instruction! SLOW) Then decode:

- Opcode goes into CU and is broken down into micro-instructions

Next the CU triggers the necessary gates to execute the instruction.

Finally the store phase:

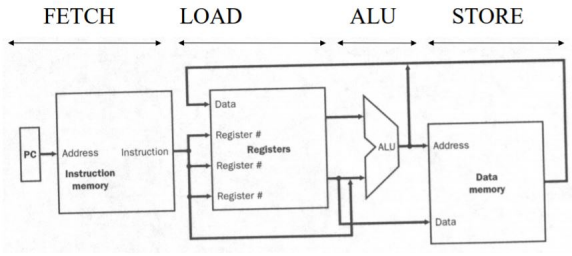
- MAR gets the address to be stored at
- Data goes to MBR
- Write signal sent to RAM AR and DR, data copied from MAR to AR, MBR to DR.

The cycle is an infinite loop.

7.2 MIPS Pipeline CPU

The main idea behind the MIPS pipeline CPU is that it is able to execute several instructions simultaneously. This is done by having several stages:

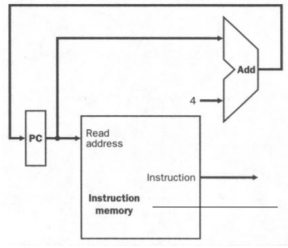
- Fetch: Get the instruction from memory (cache or otherwise)
- Load: Load the instructions into registers
- Execute: Run the instructions
- Store: Save the result either in data cache or in a register.



The stages are separated by their own instruction registers (and often own CU), so that each stage can operate independently. Note that the IR's have a buffer IR in front of them to protect the data from being overwritten, and are controlled by AND gates by the CU.

7.2.1 The Fetch Portion

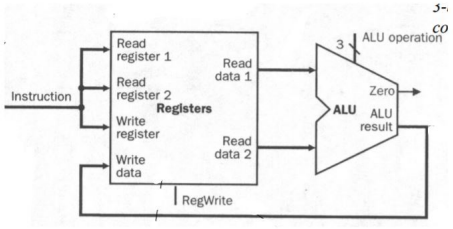
The PC is in a loop with itself, so that every tick it will increment by 4. The PC is then connected to the instruction cache, which will check if the required instruction is in cache. If not, it will need to fetch from RAM.



7.2.2 The Load Portion

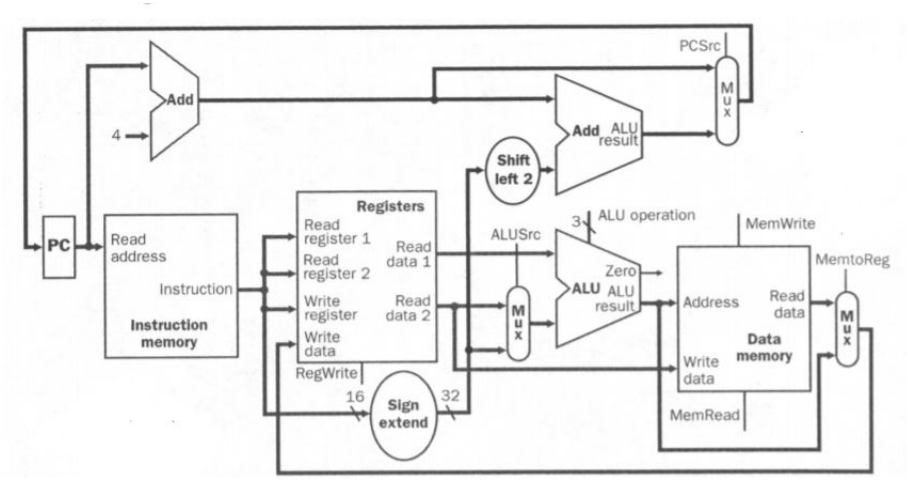
The registers are accessed by an id number (see the MIPS instruction formats). The instructions leave the instruction cache and enter the appropriate register. The data is then pushed into the ALU.

The registers can also read input from either the ALU or data cache at this point.



7.3 Overview of MIPS CPU

I skipped the ALU and store portions since I'll talk about them here.



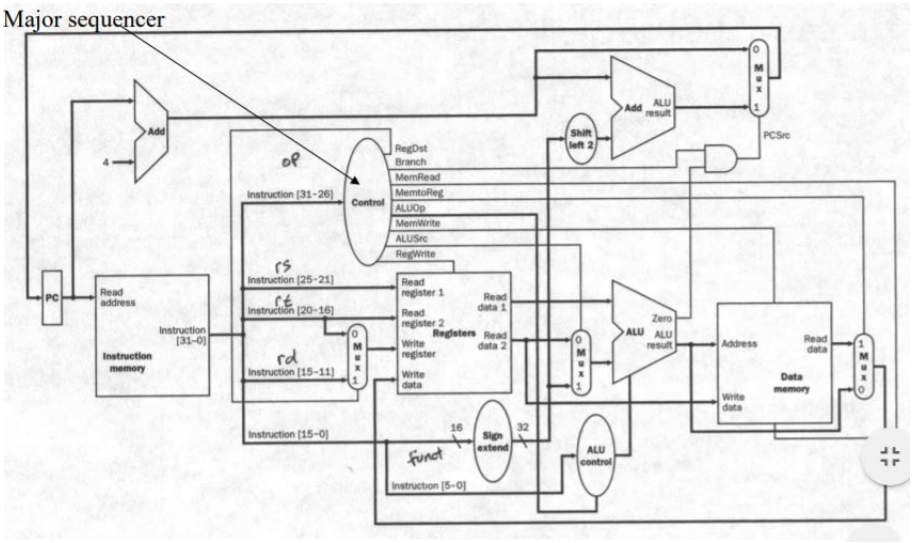
The sign extend box is there to deal with instructions with constants. Normally constants are 16 bits, (from instructions such as add immediate, jump, etc) and thus need to be extended to 32 bits to be compatible with the ALU. The multiplexer in front of the ALU decides from where it will be accepting input (controlled by the CU).

The shift-left-2 box and the adder above the ALU is for branching. When we want to branch, the address we jump to needs to be multiplied by 4, so the shift-left-2 will move all the bits of the constant two to the left (equivalent to multiplying by 4!). Then this constant is added to the current value of the PC, which outputs the new PC address. This is then fed into the multiplexer which decides whether or not we are branching (controlled by CU). The output of this multiplexer is fed back into the PC to continue incrementing. One thing to note is that when branches happen, the next instruction already begins to get processed, regardless of whether the condition was true or false, so we need to disregard the staged instructions.

The result of the ALU can either be stored in data cache, or be piped over to a multi-

plexer which decides whether data is being read from the data cache or the ALU before being saved into a register.

Not shown is the circuitry which handles the timing of it all. There is a CU (or multiple) and multiple instruction registers which control all of the multiplexers and AND gates which enable/disable the pieces of the pipeline. In particular, each time we want to shift to the next phase, the CU needs to allow this, so that data is not lost and everything is synchronized.



This picture shows the path of the different pieces of the instruction code. (See assembler instruction formats in section 10)

Part III
MIPS Assembler Programming

8 Assembling, Compiling, Linking

In compiled languages, the compiler verifies syntax and makes the conversion into assembly code. Next, the assembler verifies the assembly code syntax, and makes the conversion to machine code (binary/hexadecimal). (This is the .o files in C) Next the linker checks all the places where you made library calls and 1.) makes sure they exist, and 2.) merges the library code into your program. Finally an executable is created.

Your program will only execute when the code version is the same as the compiler, the machine code output is of the correct format for your CPU, and loader version is the same as the OS API on your computer. (I don't think any of that is relevant for the final but doesn't hurt to cover it.)

8.1 Two-Pass Assembly

On pass 1, the assembler builds the symbol table. (Identifies all labels, links them to a PC address or place in RAM).

This is done by scanning the source from top to bottom, counting the number of instructions from the base address.

Then on pass 2, the machine language program is built, by making a 1:1 mapping from the assembler instructions to binary.

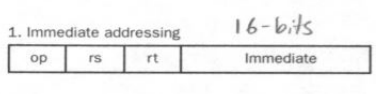
9 Basic Assembler

I won't be covering the basics of assembler programming as the slides are mostly self explanatory, and there's no replacement for practice! Plus there's only a few days until the final, so instead I'll focus on the structure of assembler code which was not as well described in the slides.

9.1 Addressing Modes

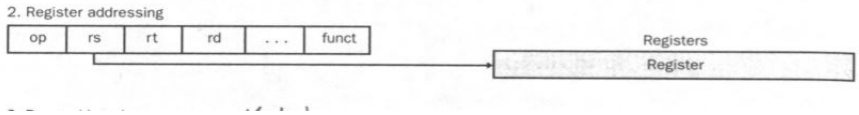
There are different types of instructions in assembler, each using different addressing modes. These are:

- Register: All arguments are registers ie add $s1, $s2, $s3
- Base or Displacement: Register + offset ie: lw $s1, 100($s2)
- Immediate: 16-Bit constant used directly, ie: addi $s1, $s2, 100
- PC-Relative: PC + offset is where the program continues execution from. ie: j 2500 will jump 2500 away from where the PC is currently pointing
- Pseudo-Direct: The 6 most significant (highest order) bits of the PC will be concatenated with the 26 bit number you specify.

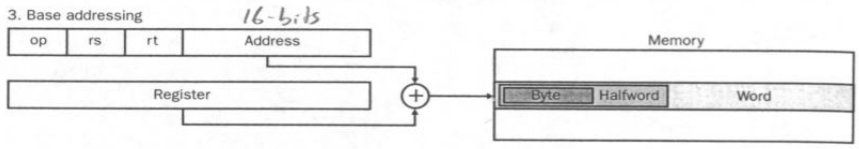


So here, the instruction looks like: OP-code, (what type of instruction it is), then rs refers to the first register source (if any), then rt refers to the second register source, (if any), and

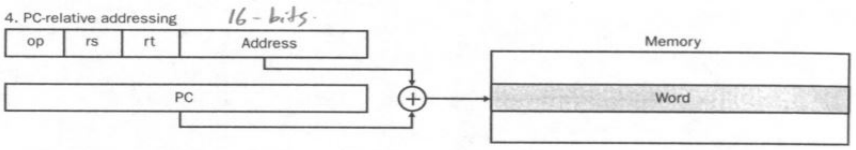
immediate is the constant you put there.



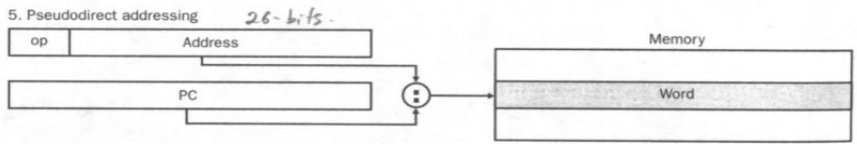
In this case, all 3 arguments are registers.



In the case of base addressing, the register value gets added to the constant you specify. The right-hand side is meant to illustrate that you can refer to bytes, halfwords and words.



Here, the PC is being added to the constant address you specify.



Finally, here the first 6 bits of the PC are concatenated with the 26 bit address you specify.

9.2 Instruction Formats

There are 3 instruction formats:

- R: Register format (only registers are used) so for these, rs, rt and rd are all register addresses.
- I: Immediate format (mix of registers with a constant)
- J: Jump format (just a constant target address)

In the slides (and on your formula sheet) this table sits dauntingly:

Name	Format	Example						Comments
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
add	R	0	2	3	1	0	32	add $1,$2,$3
sub	R	0	2	3	1	0	34	sub $1,$2,$3
addi	I	8	2	1	100			addi $1,$2,100
addu	R	0	2	3	1	0	33	addu $1,$2,$3
subu	R	0	2	3	1	0	35	subu $1,$2,$3
addiu	I	9	2	1	100			addiu $1,$2,100
mfc0	R	16	0	1	14	0	0	mfc0 $1,$epc
mult	R	0	2	3	0	0	24	mult $2,$3
multu	R	0	2	3	0	0	25	multu $2,$3
div	R	0	2	3	0	0	26	div $2,$3
divu	R	0	2	3	0	0	27	divu $2,$3
mfhi	R	0	0	0	1	0	16	mfhi $1
mflo	R	0	0	0	1	0	18	mflo $1
and	R	0	2	3	1	0	36	and $1,$2,$3
or	R	0	2	3	1	0	37	or $1,$2,$3
andi	I	12	2	1	100			andi $1,$2,100
ori	I	13	2	1	100			ori $1,$2,100
sll	R	0	0	2	1	10	0	sll $1,$2,10
srl	R	0	0	2	1	10	2	srl $1,$2,10
lw	I	35	2	1	100			lw $1,100($2)
sw	I	43	2	1	100			sw $1,100($2)
lui	I	15	0	1	100			lui $1,100
beq	I	4	1	2	25			beq $1,$2,100
bne	I	5	1	2	25			bne $1,$2,100
slt	R	0	2	3	1	0	42	slt $1,$2,$3
slti	I	10	2	1	100			slti $1,$2,100
sltu	R	0	2	3	1	0	43	sltu $1,$2,$3
sltiu	I	11	2	1	100			sltiu $1,$2,100
j	J	2	2500					j 10000
jr	R	0	31	0	0	0	8	jr $31
jal	J	3	2500					jal 10000

Lets break it down:

At the top, we have the number of bits each piece of the instruction is allowed to be. The format is always: 6-bit op-code, 5-bit source register rs, 5-bit source register rt, 5-bit destination register rd, 5-bit shift amount shmt, and 6-bit function code, funct (more specific op-code).

Notice that the order of the registers in the format is different than how you'd actually write the code. If you wanted to store the result of adding $t1 and $t2 into $t3, you'd write `add $t3, $t1, $t2`. But the way you'd write the format is: op — 1 — 2 — 3—.

Also notice that similar instructions have the same op-code, but with different funct values. For example, add and sub have the same op code of 0, but different func values 32 and 34.

J and I-type instructions have many of their fields merged, since they aren't necessary. See the right hand column for a code example, and see which fields become the new source, destination etc.

Using the tables in the formula sheet, see if you can translate assembler code into binary. (You'll need to on the final!)

10 Complex Data

10.1 Sizes of Data

Data can be categorized by size. It's size affects how we'll handle it in assembler.

Small data is things like:

- .byte use lb (load byte)
- .word use lw (load word)
- .float use lw (load word)
- .double see below.

Doubles are also small data, but need to be handled differently. Need to load address, then load the first half into one register, and then load the second half into another.

```
la $s0, D
lw $t0, 0($s0)
lw $t1, 4($s0)
```

Medium data is things like strings and arrays.

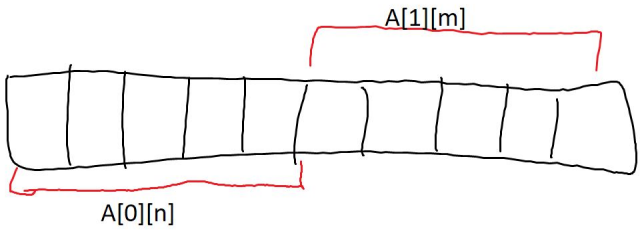
```
#Strings + Arrays

.text
la $t0, S       #load address of the string
lb $t1, 0($t0)  #access first char
lb $t1, 4($t0)  #access second char (overwrote what was in $t1)

la $t2, Arr     #load address of array
lw $t3, 0($t2)  #access first index of array (like the string)

.data
S: .asciiz "cool guide bro"
Arr: .word 1,2,3,4 #makes an array of 4 words
#or alternatively:
Arr: .space 40     #gives 40 bytes of space to be used as desired
```

For large data, we have things like 2D arrays and structs (basically collections of data (kind of like objects)).

```
#making a 2D array
.text

la $t0, arr     #load address of the 2D array
li $s0, 4       #load the size of each cell
li $s1, 40      #load the size of each row
```

```
add $t1, $t1, $s1 #get offest for second row, 3rd column
muli $t2, $s0, 3
add $t1, $t1, $t2

add $t2, $t1, $t0 #arr[2][3] = arr + offset
lw $t1, ($t2)   #load arr[2][3]

.data
arr: .space 80  #makes a 10x2 array, two rows, 10 columns
```

It's worth noting that 2D arrays are an imaginary construct. They don't exist inside the computer. They are really just a collection of arrays one after the other in one big array.



So here the first group of 5 cells is the first row, and the second group is the second row.

A struct is very similar. It's just a long array of contiguous data, but the cell size is not fixed. So say you want a struct of an int and a double:

```
.text
   la $t0, struct
   li $s0, 0  #offset to get to the int
   li $s1, 4  #offset to get to the double

.data
struct: .space 12 #4 for the int, 8 for the double
```

10.2 Passing Large Data as Parameters

There's a limited number of registers we can use to pass information into a function, so to pass large data we need a way to encapsulate all the information in one small package.

Easy solution, pointers! Simply pass a pointer to the address of the object you want to use inside the function.

Similar to Java and C, the function will change the original data! (call by reference).

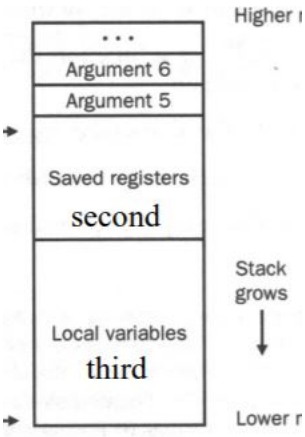
You can also pass the data using the stack to do call-by-value. This requires writing each individual item of the struct to the stack, before entering the function, and then popping the stack repeatedly to access the variables once inside the function.

11 Stack & Recursion

11.1 Stack and Frame Pointer

The `$sp` register is reserved for the stack pointer, which points to the top of the stack. (I assume you know what a stack is, if not check out my COMP250 guide!)

The `$fp` register points to the start of the stack in RAM. You probably won't ever need to move $fp, just make sure no variables are addressed beyond it.

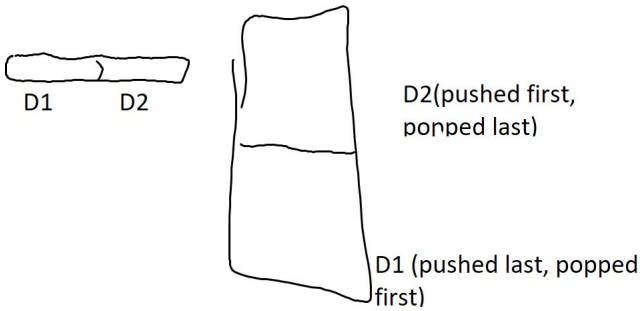


The stack grows DOWN. So if the bottom of the stack is address 100, then the stack addresses go down, 99, 98, 97...

So to pass our struct from the last section using call by value, we need to do:

```
la $t0, struct  #load base address
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t3, 8($t0)     #load the pieces of the struct (the int and two pieces
    of the double)

subi $sp, $sp, 12 #make space on the stack
sw $t1, 12($sp)
sw $t2, 8($sp)
sw $t3, 4($sp)  #careful to place in correct order
```

Careful when placing and removing doubles on the stack because order matters!



11.2 Local Variables Using Stack

Suppose you have a function:

```
int main(){
   int x = 0
   func(5)
}
void func(int x){
   x = 6
}

#in assembler:
.text
   main:

      li $s0,
      subi $sp, $sp, 8 #make room on stack
      sw, $s0, 0($sp)    #save the local variable x

      sw, $t0, 4($sp)    #save argument to stack
      jal func      #call func(5)

   func:
      lw $t1, ($sp)
      addi $sp, $sp, 4 #pop the stack (get 5, the value for x in main is
          untouched)
      #do stuff
      jr $ra          #return
```

Note that the above code is not complete, but thats roughly how you'd pass in parameters using the stack. Whatever happens inside the function (unless call-by-reference) will not affect what was outside. If the values were not saved to the stack before entering the function, and were only living inside registers, they are at danger of being overwritten. (Including the $ra, so take care during recursion!)

11.3 Recursion

Recursion is tricky, especially if you're not already comfortable with using the stack.

We need to be careful with what happens to the $ra register, since the return address will be updated several times in the recursion before we actually start returning. So we need to save it to the stack at every level of the recursion, and load it into $ra when we want to return.

I'll do my best to explain the factorial example given in class, but it's probably better if you try it yourself.

```
.text
.globl main
main:
        li $a0, 5
        jal fact
        move $a0, $v0
        li $v0, 1
        syscall                         #print the result
        li $v0, 10
        syscall                         #end
fact:
        subi $sp, $sp, 8
        sw $ra, ($sp)                   #save return address to on the stack
        sw $a0, 4($sp)                  #save the argument
        bne $a0, $zero, fact2           #if n<1
        addi $sp, $sp, 8                #pop stack on the way up
        li $v0, 1               #return 1
        jr $ra
fact2:
        subi $a0, $a0, 1                #n-1
        jal fact                        #recursive call
        #########                       $ra brings us back here
        lw $ra, ($sp)
        lw $a0, 4($sp)
        addi $sp, $sp, 8                #get argument and return address from stack
        mul $v0, $a0, $v0               # n* fact(n-1)
        jr $ra
```

The code above will actually run and print fact(5) = 120 if you throw it in MARS.

I really recommend drawing the stack as you go along if you try to follow this.

When fact is first called, we need to save the return address of main. We also need to save the input to the function (remember $a0 is used for parameters, and $v0 for return values). So we make 8 spaces on the stack and add them. Next we check for our base-case. Well right now n=5, so we go down to fact2. We subtract 1 from n, so now n=4. We then recursively fall fact. Now the $ra is going to point to fact2, right below the line the jal was on. Again we save the return address, and argument (now 4) to the stack. Repeat the process until the base case, n=0.

Now that we're at the base case, we don't jump to fact2. We instead "pop" the stack by moving the stack pointer up 8. We return 1 (our base case) and return. This brings us down to fact2 below the jal. We load the argument, and return address from the stack, and pop. We then multiply the current argument value with the previously returned, and return

the result. The return brings us back to fact2, and we repeat until the return from the stack brings us all the way to main.

Again it will make a lot more sense if you draw the stack frames.

12 Floating Point Numbers

Floating point numbers need to be treated specially in MIPS. In particular, to load and save floats/doubles, we use: `lwc1` and `swc1`. When performing add, subtract, multiply and divide with floats, need to use: `add.s`, `sub.s` etc. And for doubles, need to use `add.d`, `sub.d` etc.

There is also special commands for branching with floating points: `c.1t.s` will compare floats, and `c.1t.d` will compare doubles. Both result in 1 for true, 0 for false.

You should always use $f registers when using floating point numbers. And when it comes to doubles, need to use two registers.

When multiplying and dividing doubles two special registers are used.

The $hi register stores the high-order bits in multiplication, or the remainder in division. The $lo register stores the lower order bits in multiplication, or the quotient in division.

See the slides and appendix of textbook for the full list of commands.

12.0.1 Co-Processors & RAM

Along with the main processor, there is also several co-processors that work in-parallel with the main. Co-0 deals with exceptions and interrupts, and Co-1 deals with floating point operations (since it requires different circuitry than integer operations).

Both the main processor and the co-processors are directly connected to RAM.

13 The Heap

The heap is where dynamic memory lives in RAM. This is where you can "malloc" (memory allocate in C), at runtime. The heap is managed by the OS. And contrary to the stack, the heap starts near the "bottom" of RAM, and addresses upward. If the stack and heap touch, then the program has used all of the available space in RAM.

To ask the OS for space on the heap, need to use syscall 9. Where you place the amount of bytes you want into $a0. After the operation, a pointer to the beginning of the block of space will be in $v0.

Part IV
Advanced Circuitry

14 Polling & Interrupts

14.1 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.

14.2 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.

14.3 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.

14.4 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.

14.5 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).

14.5.1 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!

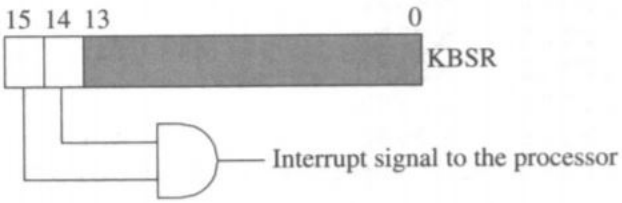
14.6 Interrupts

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.

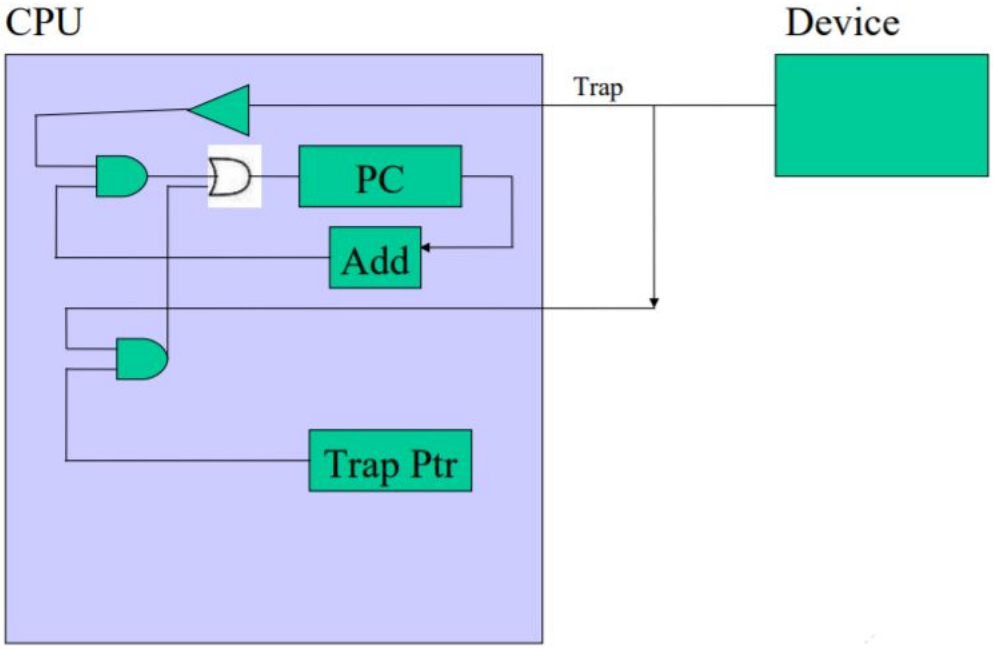
This is a picture of what an interrupt would look like on the register's end:



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.

14.6.1 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.

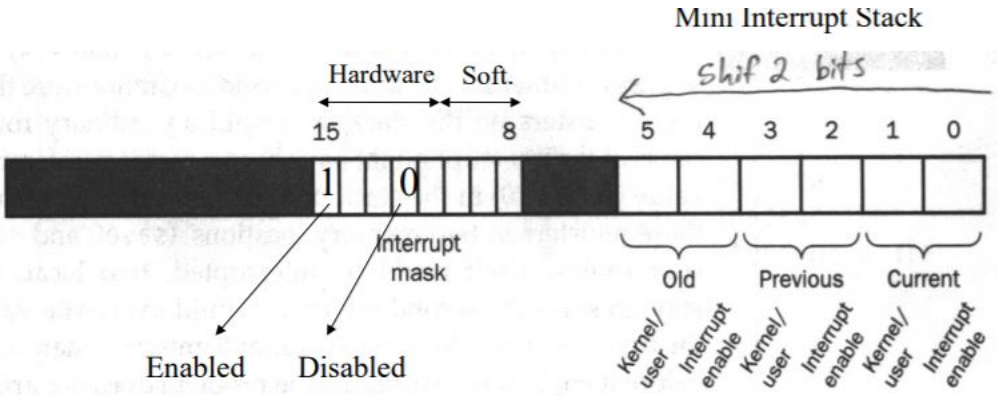
14.7 The Exception Co-Processor

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:

14.7.1 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.

14.8 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.

15 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.

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

15.1 Cache-Loading

15.1.1 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.

15.1.2 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.

15.2 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)

15.3 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 $(1011)_2$ mod $(100)_2 = (11)_2$, 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:

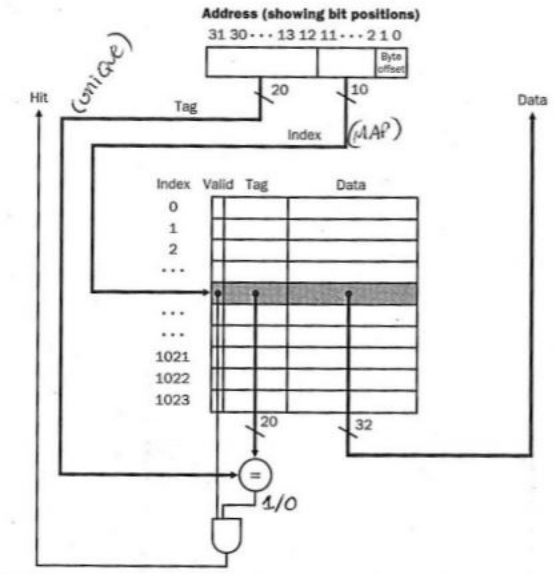
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory(10110_{two})
111	N		

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

address.

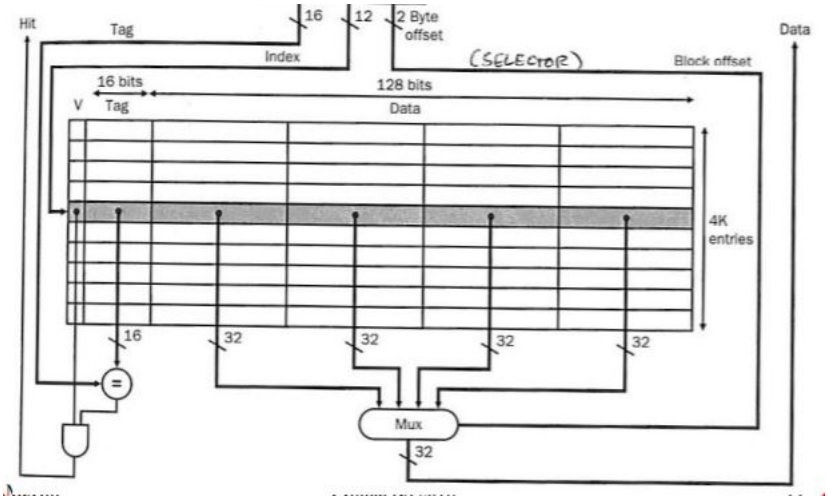
The column V keeps track of whether that table entry is valid, meaning whether it has any data in it or not. (or else how would we distinguish 00000 (not initialized), from 00000 (the number zero)?).

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.

15.4 Performance

15.4.1 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.

15.4.2 Examples

Problem 2. *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 3. *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..

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

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

Problem 4. *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!

16 Virtual Memory & Performance

16.1 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)

16.2 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.

16.3 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.

16.3.1 Page vs Frame

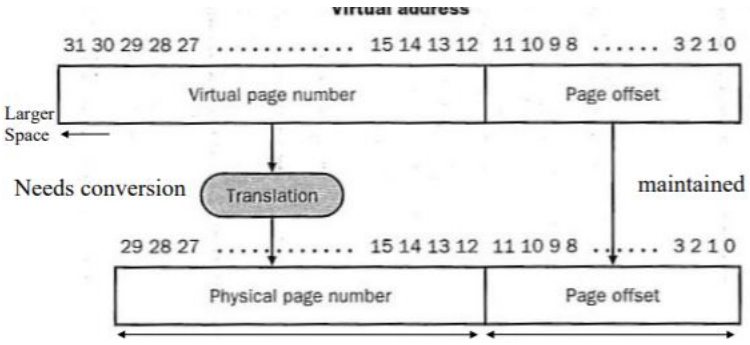
A **page** is the abstract idea of a "functional unit". It has varying size (like functions, while loops etc).

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.

16.3.2 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.

16.3.3 Types of VM

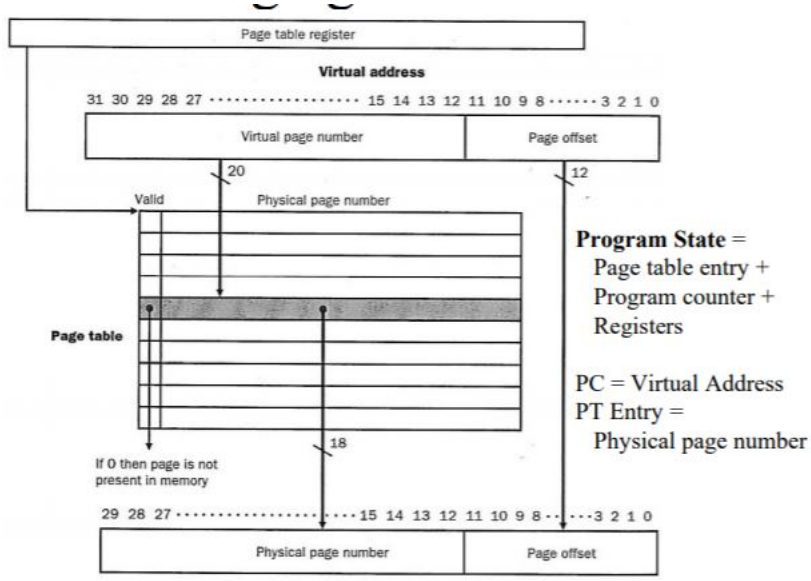
Paging: the overlays have the same size, overlay matches the framesize, and the addressing is done simply by concatenating the page number with the offset:

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)

16.4 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.

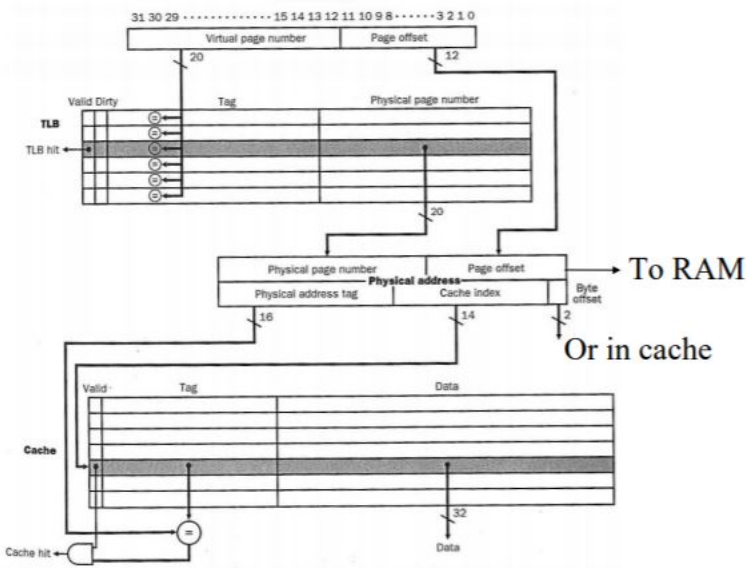
Of course, going to RAM constantly is slow, so we need a faster way to do the conversion.

16.5 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.

16.6 VM Performance

Let CPU time = program execution + memoryStall

Let memoryStall = readStall + writeStall

Let readStall = programReads*readMissRate*readMissPenalty

Where the programReads is the amount of times the program needs to read in a page from the disk to RAM, readmissRate is the probability of this occurring, and readMissPenalty is the number of ticks it takes to perform the read operation.

Let writeStall = (programWrites * writeMissRate * writeMissPenalty) + writeBufferTime

Where the terms are symmetrical to readStall, + the time it takes to actually write. (Feels incorrect to me, if anyone knows what the writeBufferTime really means, please let me know via e-mail or facebook message!)

16.6.1 Examples

Problem 5. *Suppose the instruction cache miss rate is 2%, the data cache miss rate is 4%, each instruction takes 2 cycles to complete, and the miss penalty is 40 cycles. The amount of instructions that perform load and store operations is 36%. What is the speed improvement*

if we have a perfect cache with zero misses? (Can put entire program into cache)

Solution. No formulas, just unit conversions and problem solving.

The time taken for missing instructions (in the instruction cache) is:

$$n(0.02)(40) = 0.8n$$

where n is the input size.

The time taken for missing data (for the data cache) is:

$$n(0.04)(0.36)(40) = 0.58n$$

Then the total stalls is $0.8n + 0.58n = 1.38n$. We multiplied the 36% here since only load and store operations use the data cache.

So the total amount of time (including good instructions) is:

$$2n + 1.38n = 3.38n$$

. and the ratio compared to a perfect CPU is thus:

$$\frac{3.38n}{2} = 1.69$$

The other calculations in the sides are done similarly. But note that when doubling the clock speed, from the point of view of the CPU, the miss penalty doubles. (we could have done twice as many things while waiting for the stall).