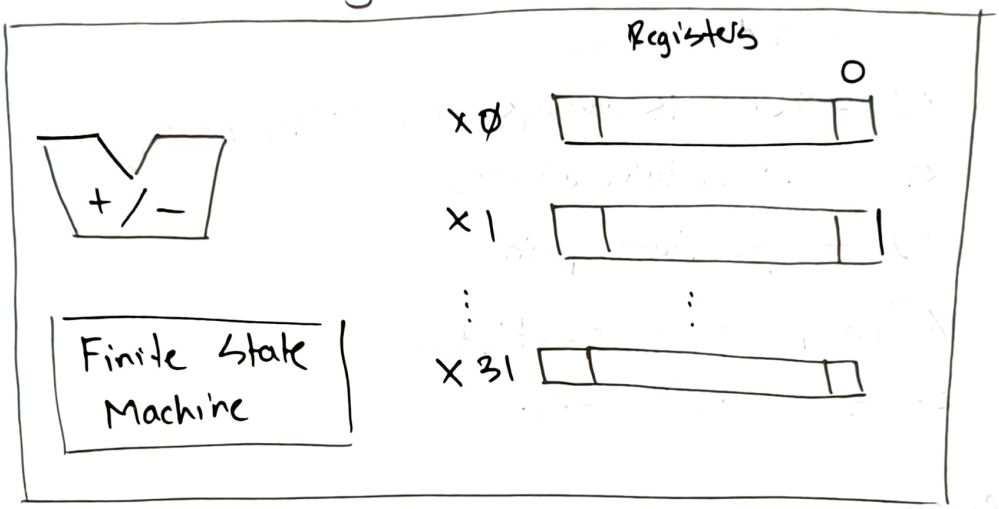
Week 1: Lecture 1 Jan 6, 2024

Today: Basic Computer Organization and Assembly Language Programming

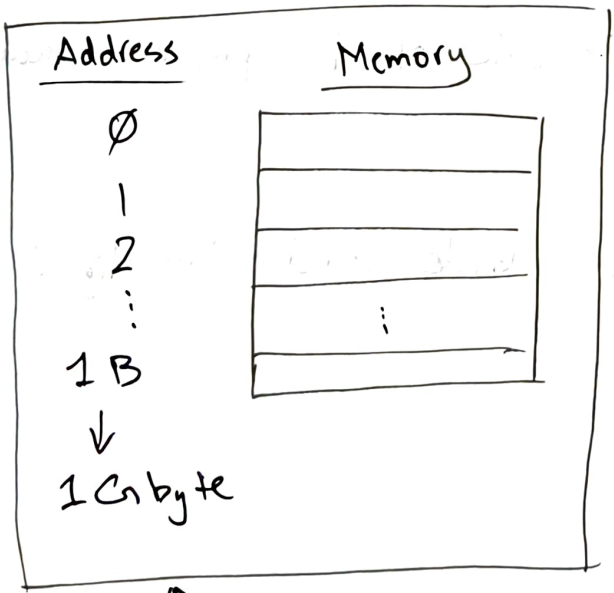
Computer Organization

* A computer has 3 main parts

Central Processing Unit (NIOS V/RISE V)



Memory



Input/Output Unit

- screen in/out
- mouse
- network/microphone
- camera
- keyboard
- digital hardware

System "Bus"

① CPU:

- ~~controls~~ performs computations
- makes decisions and controls flow of program
- using NIOS V in this course
- main computation is done in the 32 x 32 bit registers
- each register has a second, more useful name
  ↑ x0 → x31

e.g. x0 – zero : is always zero (0 is handy)
x1 — ra : return address
x2 – sp : stack pointer
x5, x6, x7 — t0, t1, t2
others — s0 → s8

② Memory:

- holds variables and data structures
- also holds the machine code of the program executing

③ Input / Output

- communicates with the outside world thru digital logic

Programming (C program magic)

```
int A, B, C, D;
```
①

- 4 × 32-bit numbers
- Let's put A, B, C, D into registers t0, t1, t2, t3

t0	1	A
t1	8	B
t2	9	C
t3	7	D

```
A = 1;      ②
B = 8;      ③
C = A + B;  ④
D = B - A;  ⑤
```

Assembly

① no assembly language

② li t0, 1

③ li t1, 8

④ add t2, t0, t1

⑤ sub t3, ~~t0, t1~~
t1, t0

Function

(nothing to do)

t0 ← 1

t1 ← 8

t2 ← t0 + t1

t3 ← ~~t1 - t0~~ ? might have it
t1 - t0 backwards ?

iloop: j iloop

- li means "load immediate"

- label called iloop, says jump to iloop ... an infin. ~~world~~ loop

Week 1: lecture 2 Jan 8, 2024

Today: Lab1 Loops and Conditional Branches in NIOS V Assembly Language.

Lab 1 Part 2: learn CPUlater, single stepping, breakpoints

Part 3: write a program

Part 4: learn about the monitor program

Part 5: run part 3 program

Conditional Branch

- to make a loop, you need a "conditional branch" assembly statement
- have an unconditional "branch" = "jump"

0x20 ↑ address

```
iloop: j iloop
```

label

- a conditional branch → transfers execution to an address/label if a specific condition is true (e.g. t2 = t3) otherwise execution goes to the next instruction in sequence
- an example: a loop that executes 4 times (and does nothing)

```
.global _start        ← label called _start
_start: li t0, 1      * t0 ← 1
        li t1, 4      * t1 ← 4
```

represents address where ... goes

t0 is one of 32 registers

```
myloop: addi  t0, t0, 1     # t0 ← t0 + 1
        ble   t0, t1, myloop
fin:    j fin
```

* "branch" less than or equal to... is $t0 \leq t1$?

* general form of conditional branch statement:

bXX rA, rB, DEST_LABEL

* rA and rB are the two registers being compared
* XX is the condition

~~eg. if~~ eq if rA = rB then branch to DEST_LABEL, else go to next statement

ne	"	!= "
ge	"	>= "
le	"	<= "
lt	"	< "
gt	"	> "

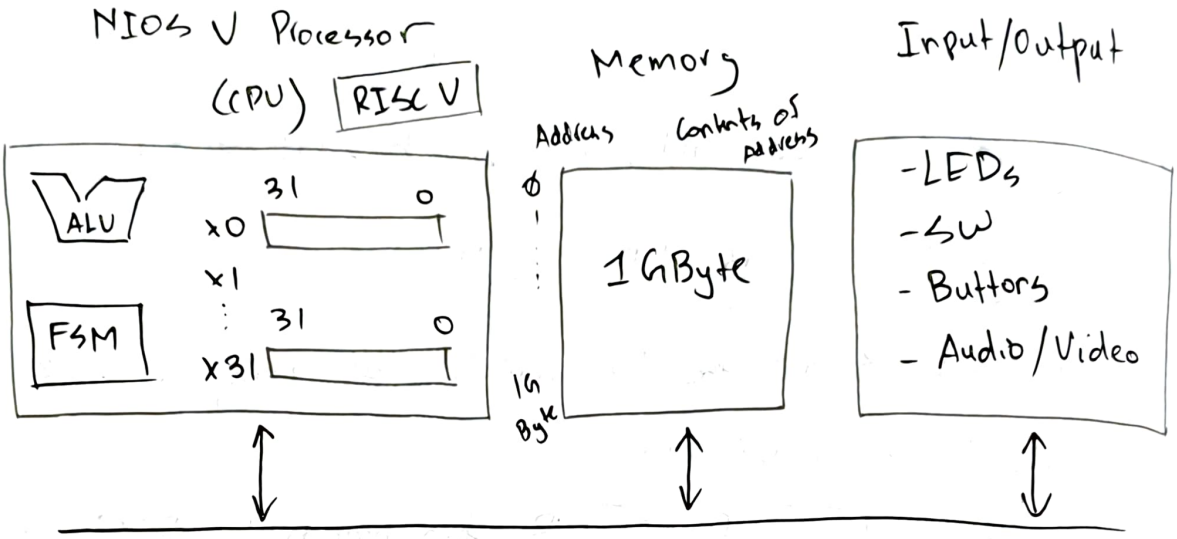
} you add "u" at the end (e.g. leu, ltu, gtu...) for unsigned

* program gets turned into 5 words of numbers in memory
* "_start" has the value that the assembler knows about
  * address where the instruction "li t0, 1" is stored
  * equals "00000000"
* "myloop" becomes "00000008", which is address that stores the "addi t0, t0, 1" instruction.

Week 1: Lecture 3 Jan 9, 2025

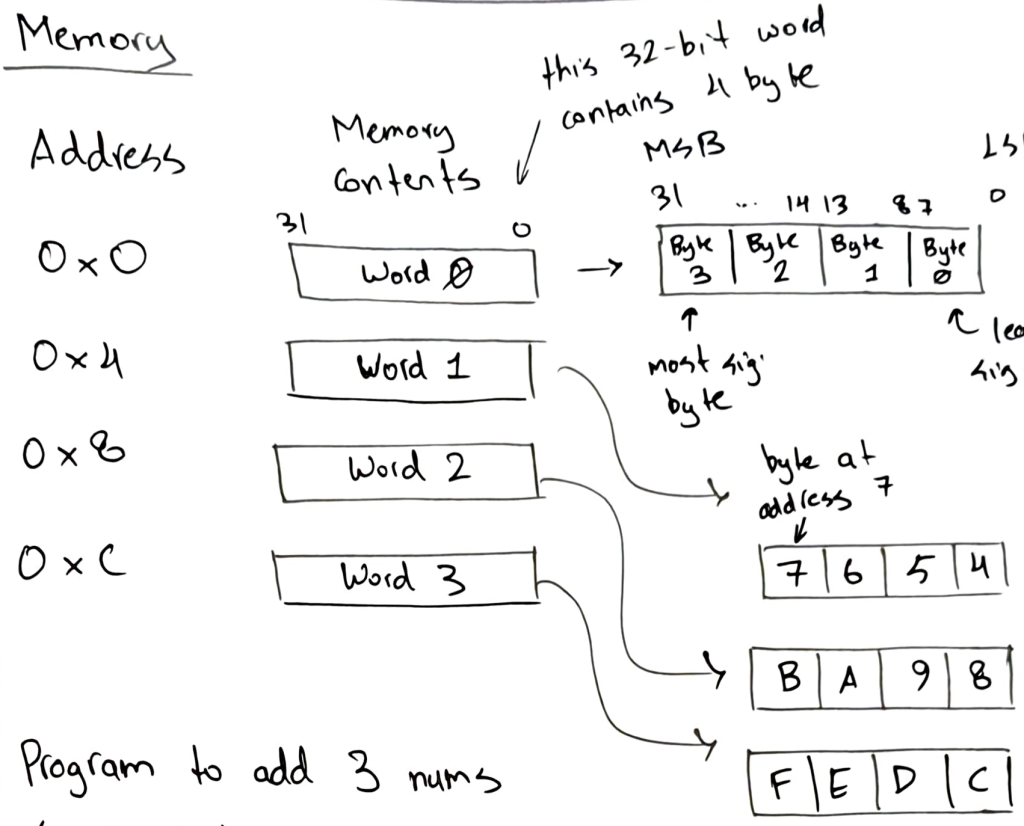
Today: Memory Organization, in NIOS V processor
↳ 32-bit

Recall Computer Organization



- max # of addresses ~~of~~ $= 2^{32}$
- NIOS V is a 32-bit processor
  - → registers are 32 bits wide
  - → number of addressable locations in memory $= 2^{32}$
  - → max amount of bits retrieved/sent to memory is 32
  - → "word size" of processor is 32 bits
  - → however, the memory is byte (8 bits) addressable
  - → every byte gets a separate address
  - → even though the processor can access 32 bits at a time = 4 bytes at 4 different addresses

Memory



- "x" or actually "0x" is zero and it's in hex
- ".word" is a command to assembler to put "10" into mem addr. 0x2c

Program to add 3 nums (10, 20, 30) in memory

- need to initialize memory like this in C

```
int list[3] = {10, 20, 30};
```

- skeleton of the program to start: and how it is laid out in memory

Address	Assembly Code	
0x0	la t0, list	← takes address of list (0x2c) and puts it into t0
0x8	lw t1, (t0)	← pointer dereferencing ... load ~~word~~ into t1 what's stored at the mem address sitting in t0
⋮		
0x28	done: j done	
0x2c	list: .word 10	↝ 0xa
0x30	.word 20	↝ 0x14
0x34	.word 30	↝ ~~0x1a~~ 0x1e

} base 16

Week 2: Lecture 1 Jan 13, 2025

Today: Add 3 numbers in full

/* Program to add 3 numbers in memory */

①
```
.global _start
_start la t0, list
```

- "la" means load an address (32 bits) into a register
- in this case, list = 0x2c
- now, t0 ← 0x2c

```
done: j done
list: .word 10
      .word 20
      .word 30
      .word 40
answer: .word 0
```

⑥

0x2c	10
0x30	20
0x34	30
0x38	0

② `lw t1, (t0)`

* "lw" means "load the word" from the memory address given in register t0
* t1 ← (t0) — just like dereferencing a pointer
* you should read the brackets ( ) as "contents of memory address given by what's inside ( )"
* ∴ t1 ← (0x2c) = 10

③ `lw t2, 4(t0)`

* similar, except we add 4 to t0 and then load from memory that address
* t2 ← (t0+4) = (0x2c +4) = (0x30) = 20

④ `lw t3, 8(t0)`

* t3 ← (t0+8) = (0x34) = 30

(5)

```
add  t2, t2, t1
add  t2, t2, t3
la   t0, answer
sw   t2, (t0)
```

\# t2 ← t2 + t1

\# t2 ← t2 + t3

* puts answer = 0x38 → t0

* "sw" means "store word" in register t2 into memory address given by la, t0

+ last 2 lines can be replaced by "sw t2, 12(t0)"

Week 2: lecture 2 Jan 15, 2025

Today: - Accessing smaller items in memory: bytes/half words
- a more complex program from lab 2

[int a,b,c;
a=b+c;] → compile → [li t0, a
⋮] → assembler → [0011001
0100111
⋮]

- an instruction like `lw t2, (t3)`
loading words so far = 32 bits / 4 bytes

→ NIOS V has byte-addressable memory

- we can also access a single byte from memory using the load byte (lb) instruction

eg: `lb s3, (s2)` # load the byte @ address given in reg s2 into reg s3
(s2 = 0x30)

- lb use sign extension
- the most significant bit, bit 7, is copied into bits 8 → 31
- if don't want sign extension → use lbu (↳ unsigned)

- similarly, lh loads half a word = 2 bytes from memory and sign extends from bit 15 to bits 16 → 31
- lhu is unsigned

Initializing Memory With Values

list .word 10

assembler directive

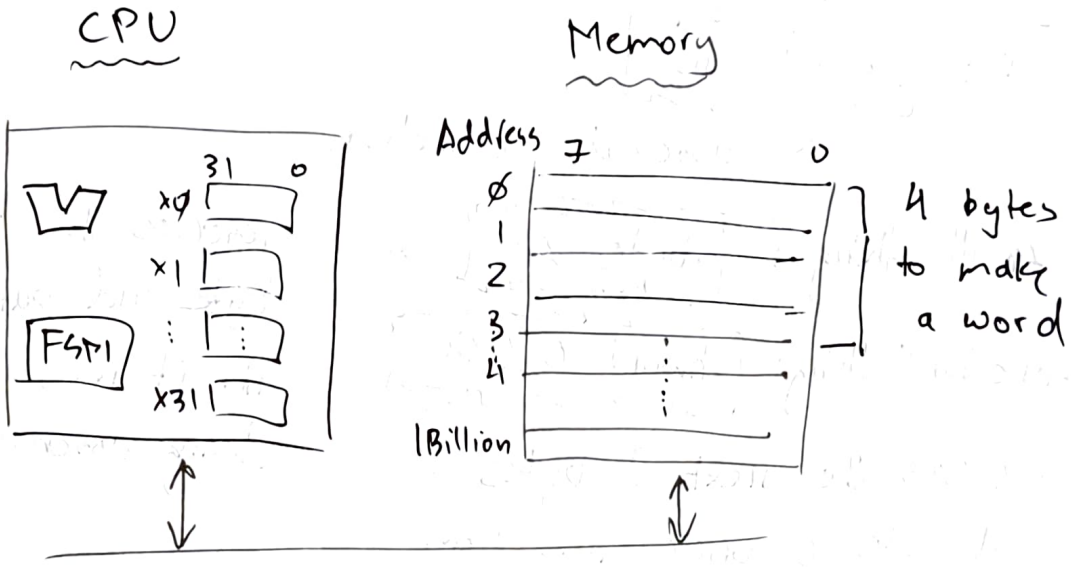
- small thing: .byte 50 ← reserves the "next" byte and put 50 in it in memory during assembly
- one half thing: .hword 0x6600

  reserves the next 2 bytes and puts 0x6600 into them
- bytes can be @ any address
- half-words must have an address on 2 byte boundaries ← address must be even
- words must have addresses that are divisibly by 4

Week 2: Lecture 3 Jan 16, 2025

Today: • more types and details of NIOS V instructions
+ move, add, load
+ logic and shift

Context of all this



General Forms of Instructions in NIOS V

→ eg: move instruction

eg: mv t3, t2 ** t3 ← t2

copies values in t2 into t3

general form: mv rD, rS ** rD ← rS

dest. register ↑ ↑ source register

→ * the "la" and "li" instructions are really more like move immediate instructions

eg: la a0, 0x20 ** a0 ← 0x20 (full 32 bits of hex 0x20)

- immediate constant
- encoded inside the instruction

+ may be implemented with an "Li"

→ the add and subtract instructions

registers only

`add rC, rA, rB` ** rC ← rA + rB

`addi rC, rA, imm` ** rC ← rA + imm

imm → a 12-bit number that is sign extended

~~`sub rA, rB, rC`~~

`sub rC, rA, rB` ** rC ← rA - rB

* sub doesn't not have "subi"

+ use addi but use a negative number

→ Recall: load word

lw t4, (a0) ** load into register t4 the word beginning at memory address given in a0 : t4 ← (a0)

General form

| lw rB, D(rA) |

** rB ← (rA + D)

* load into rB the word beginning at rA + D

→ Similarly, the store word

| sw rB, D(rA) |

** copies all of rB into the word in memory beginning at address rA + D

→ recall lb, lh, sb, sh

Logic and Shift Instructions

- need to isolate and access individual bits
- use bit-wise AND, OR, XOR → in C: && (and) || (or)
- these have two general forms

and / or / xor } | XXX rC, rA, rB |

** bit-wise operation of rA w/h rB result into rC

| XXXi rC, rA, imm |

** immediate form

eg: and t3, t1, t2 ✱ t3 ← t1 BIT wise AND t2

t1: ... 0 0 1 0 1

t2: ... 0 1 1 0 1

t3: ... 0 0 1 0 1

eg: ori t2, t1, 0x007

t1: ... 010101

imm: ... 000111

t2: ... 010111

sign extended (max 12 bits)

→ Shift Instruction

+also need to move bits within a register (shifting them left or right)

eg: srli t2, t1, 3 ✱ shift t1 3 bits to the "right", bring in 0's into higher order bits

T1 ~~eg~~: 1 1 0 1 ... 1 1 0

T2 : 0 0 0 1 1 0 ...

"shift right logical immediate"

∴ srli ← immediate (right)

srl ← not // (right)

slli ← (left) immediate

sll ← (left) not //

can replace this with a register, and "srl" will shift right by x and x is stored in that register

+shifting right by 1 divide by 2

+shifting left by 1 multiply by 2

Week 3: Lecture 1 Jan 20, 2025

Today: subroutines, linkage and parameter passing

Subroutines

- a subroutine is seperate code that is re-used in different parts of a program
- API (application programmer interface)
- key to well-structured code
- makes your code/software useable by someone else and vice-versa.

Ex: Sample C program w/h a function (subroutine)

```
int main(void){
    int x, z;
    x = my_sub(3);
    z = my_sub(4);
}
int my_sub(int p)
    return (p+p);
```

Q1: How does my_sub "know" where in the code* to go back to? *that called it

Q2: How do parameters (arguments) like 3 and 4 get sent to my_sub and the answer returned?

Q1:

one of the "argument" registers

* recall that each assembly instruction (eg: li a0, 3) is encoded as a number (eg: 0x00300513) by the assembler:

= 0x0 x00300513

program counter → 0x4 x05621ABC

↳

* the PC points to the next instruction to be executed

not one of the x0 → x31

* there is another register, called the program counter (PC) and it contains the address of the next instruction to be executed
* we need to save the program counter when calling a subroutine
* will save it in register x1, also known "ra" which means "return address"

Ex: Assembly Version

```
main:  li a0, 3          ** a0 ← 3
       call my-sub       ** ra ← pc, pc ← my-sub
next:  li a0, 4          ** a0 ← 4
       call my-sub       ** ra ← pc, pc ← my-sub
done:  j done
my-sub: add a0, a0, a0   ** a0 ← a0 + a0
       ret               ** pc ← ra
```

- we are not showing the saving of answers into x, z

Week 3: Lecture 2 Jan 22, 2025

Today: How the stack works and using it for subroutine "linkage"

Recall:

- the program counter always knows address of next instruction to be executed
- the "call" copies pc into ra : ra ← pc
- the "return" copies ra into pc : pc ← ra

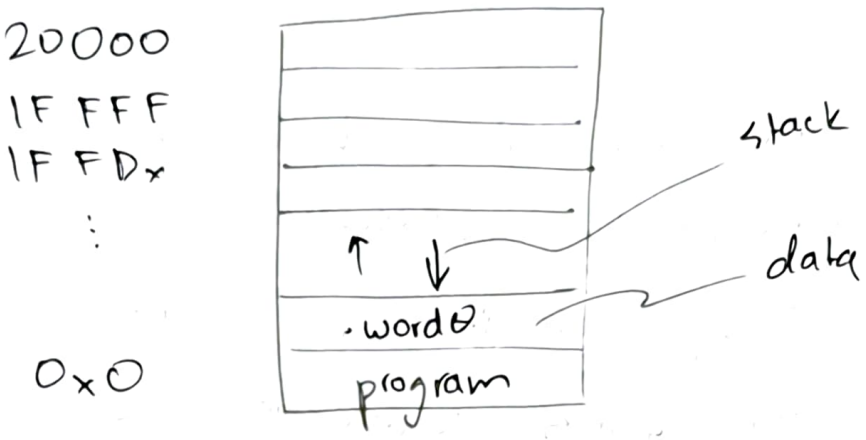
→ we need a way to save and restore ra appropriately as a subroutine calls another subroutine calls another...

→ a stack data structure → a ~~slot~~ chunk of memory

- can do 2 operations: ① "push" a new item onto the stack and ② "pop" an item off of the stack into a register
- in RiscV, register X2, also called "sp" (top of ~~point~~ stack pointer) → stack pointer for short.

ECE468

- stacks grow downwards in memory addresses



- we will initialize our stacks in the course/lab to begin @ address 0x20000 → grows downwards when pushing items more

→ Let's push 0x1234f678 onto stack

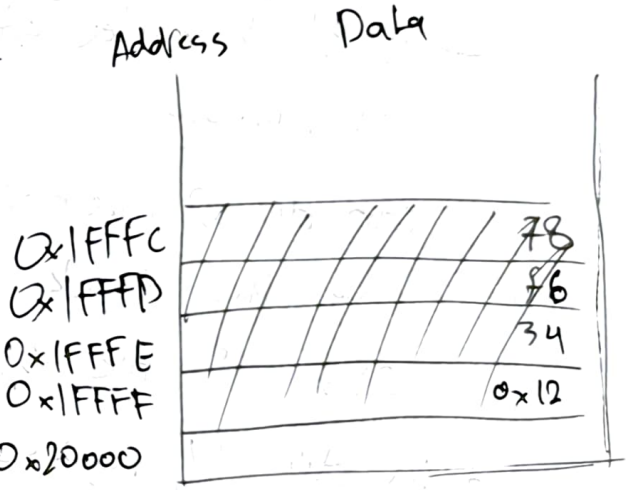
first initialize stack pointer

```
la sp, 0x20000
```

* sp ← 0x20000

```
li t0, 0x1234f678
```

* want to push onto the stack



- we want to push a 4 byte (word) onto stack

Step 1. subtract 4 from $sp

```
addi $sp, $sp, -4
```
** $sp ← $sp - 4

Step 2. store the item at the address

```
sw $t0, ($sp)
```
** memory @ $sp address ← $t0

- that's a push

- to pop an item off of the stack

```
lw $t1, ($sp)
addi $sp, $sp, 4
```
** $t1 ← ($sp)

** $sp ← $sp + 4

- now removed from stack, (but is in $t1)

Let's use the stack to correctly allow subroutines to call subroutines and save and restore correctly with the "last in first out" behaviour

```
la sp, 0x20000
li a0, 1
call sub_1      * ra ← pc, pc ← sub-1
done: j done

(*1) sub_1: addi sp, sp, -4   } push ra into
            sw ra, (sp)       } stack
            :

(*2) →      call sub_2
            :
            lw ra, (sp)       } pop ra off
            addi sp, sp, 4    } stack
            ret

sub_2: addi sp, sp, -4        } push ra
       sw ra, (sp)            }
       :
       call sub_3
(*3) →  :
       lw ra, (sp)            } pop ra
       addi sp, sp, 4         }
       ret
```

Week 3: Lecture # 3 Jan 23, 2025

Today: • Introduction to Memory-mapped Input/Output
- → connection between the virtual (inside computer) and the outside, real, physical world
- Nios V/ Risc V calling convention for subroutines

Memory-Mapped I/O

From ECE 241 and this course, we use DE1-SoC Board
- it has an FPGA on it →?
- it also has

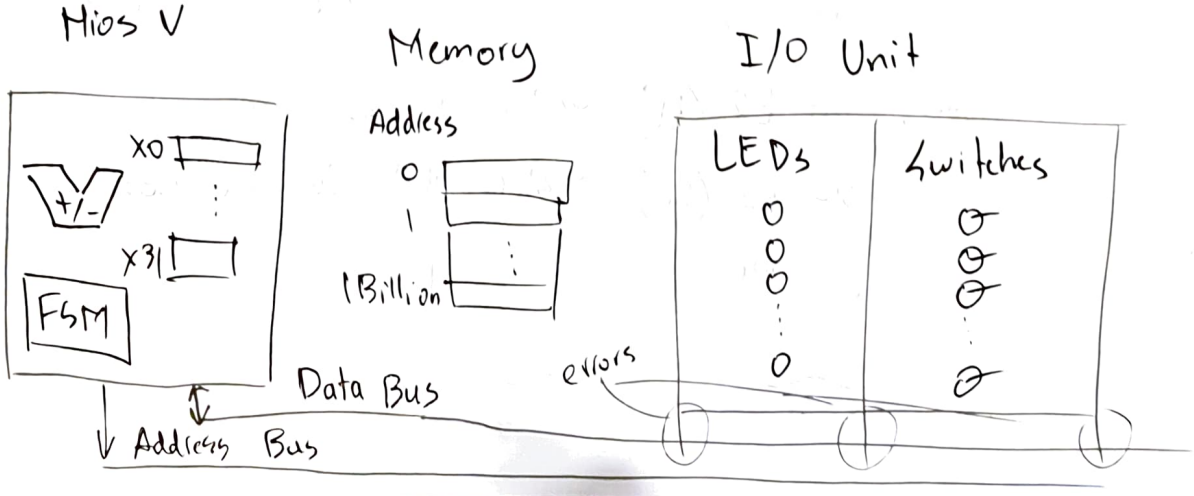
10 LEDs (LEDR 9:0)

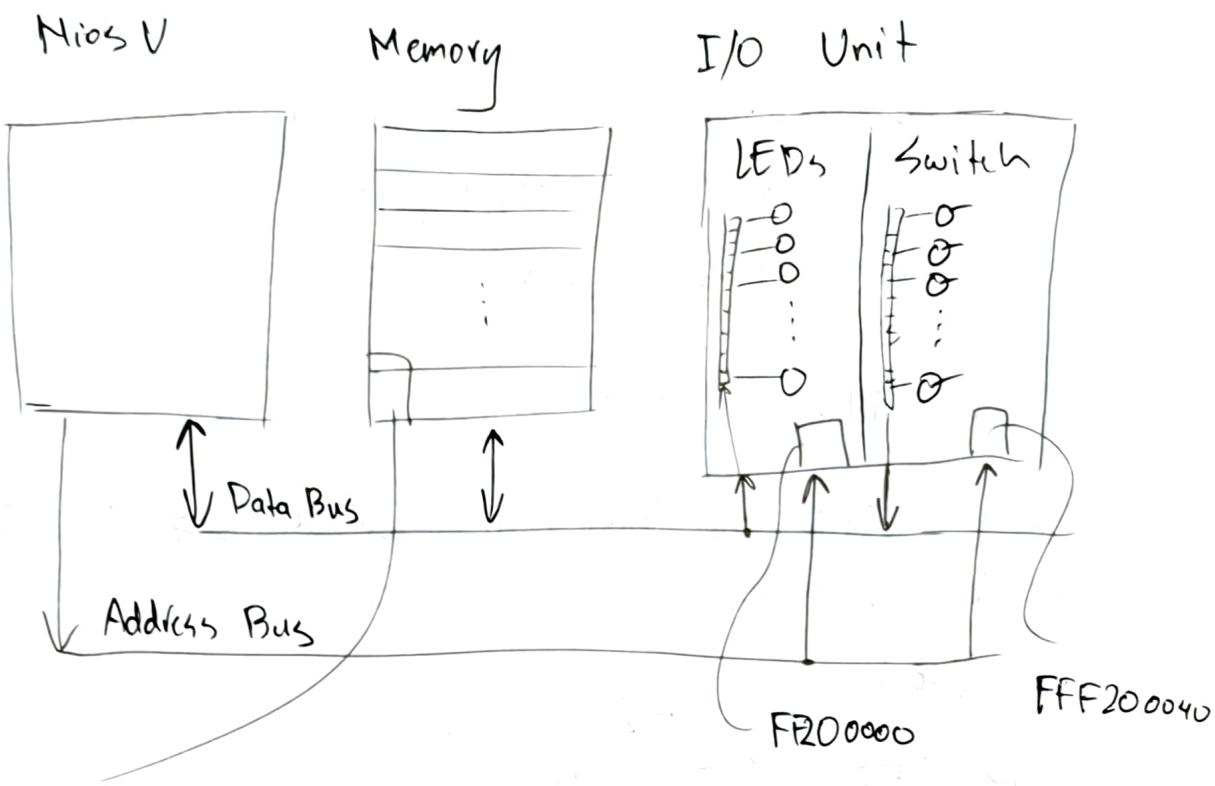
10 Slider Switches (SW 9:0)

4 Buttons (Key 3:0)

6 Seven-Segment Displays (SEG7 5:0)

- these are also available to your program running on the Nios V
- Recall the computer's organization





Digital ckt to decide if address is in the range 0 → 1B

* here is code to read the switches and write that number to LEDs:

```
.equ LEDs, 0xFF200000
.equ sw, 0xFF200040

_start: la t0, LEDs
        la t1, sw

loop:   lw t2, (t1)   * loads the switches
        sw t2, (t0)   * stores that value into LEDs
        j loop
```

these set the labels to be equal to those addresses for program readability

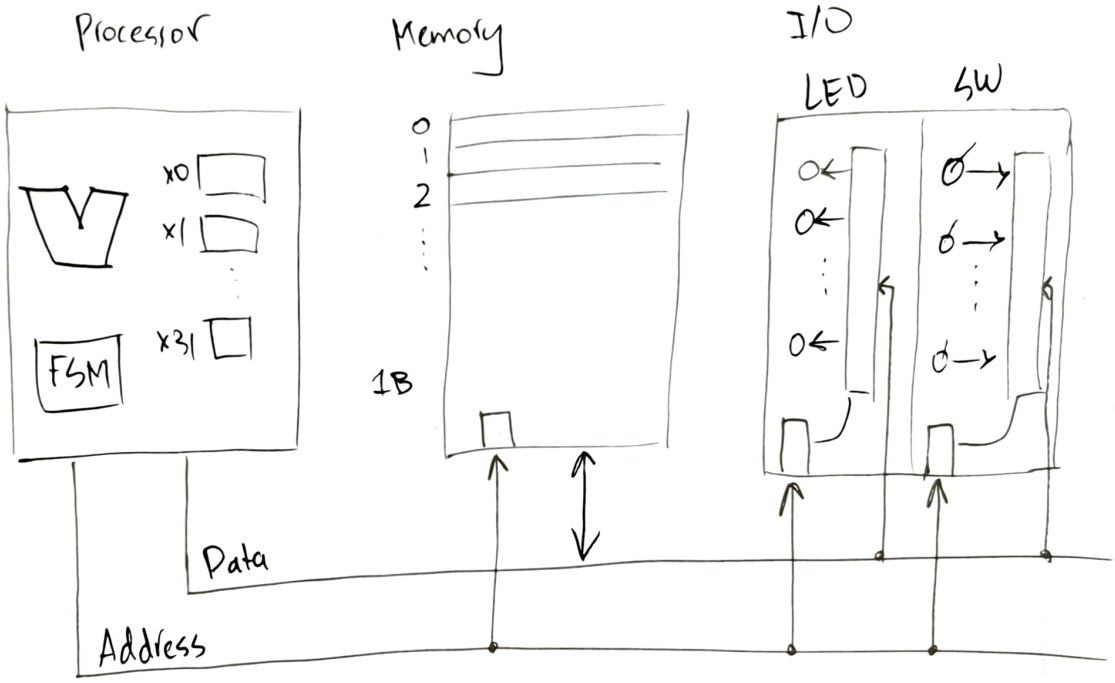
* t0 ← 0xFF200000

Week 4: Lecture 1 Jan 27, 2025

Today: I/O: parallel ports and The Polling Method of synchronization

Recall Memory Mapped I/O

lab 4
check if the button got released

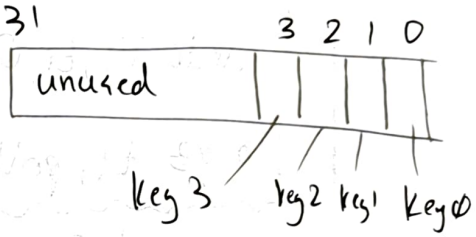


- the I/O devices have their own registers to hold and synchronize the data transfers
- these are distinct from the processor registers
- these are typically more registers that have a variety of functions
- these are collected into a parallel port.

- These weeks, we're covering synchronization b/n computer and I/O devices
  - ↳ Polling method (ask a lot)
  - ↳ Interrupt Method
- Let's illustrate polling using the pushbuttons (KEY0-3)
- they are connected to a memory-mapped parallel port

Memory Mapped Address Register

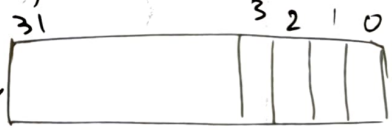
→ 0xFF200050 Data Register →



- bit i goes to 1 when key i pressed
- goes back to 0 when key i released

→ 0xFF200054, Director Register (ignore)

→ 0xFF200058, Interrupt Mask Register →



→ 0xFF20005C, Edge Capture Register

- bit i is set to 1 when key i is released
- bit i is reset to 0 if a 1 is stored into it (sent to it)
- bit i ~~is~~ stays the same if 0 is stored into it

- Let's write a program ~~that~~ to use the data register and poll it to see if key1 is pressed.

```
.equ KEY_BASE, 0xFF200050
la t0, KEY_BASE
* polling loop
poll: lw t1, (t0)      * load data register ↑ to and
                          to find if Bit #1 is 1
andi t1, t1, 0x2
beqz t1, poll
```

Data Register	B3	B2	B1	B0
AND	0	0	1	0
			B1	

→ zeroes out all bits except bit 1

→ if word of t1 = 0, button is not pressed

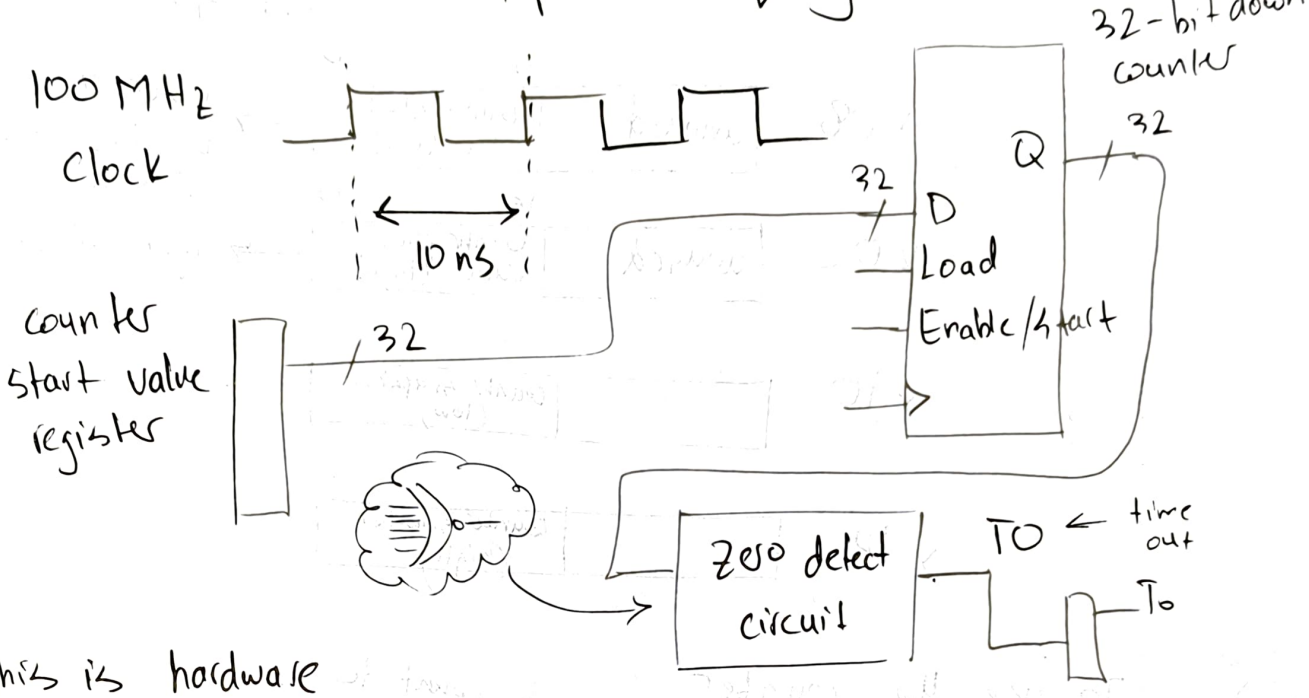
* arrive here when button is pressed

Week 4: Lecture 2 Jan 29, 2025

Today: The timer I/O Device
How software knows what time it is

Hardware Based Timer

* first need a consistent, precise frequency clock

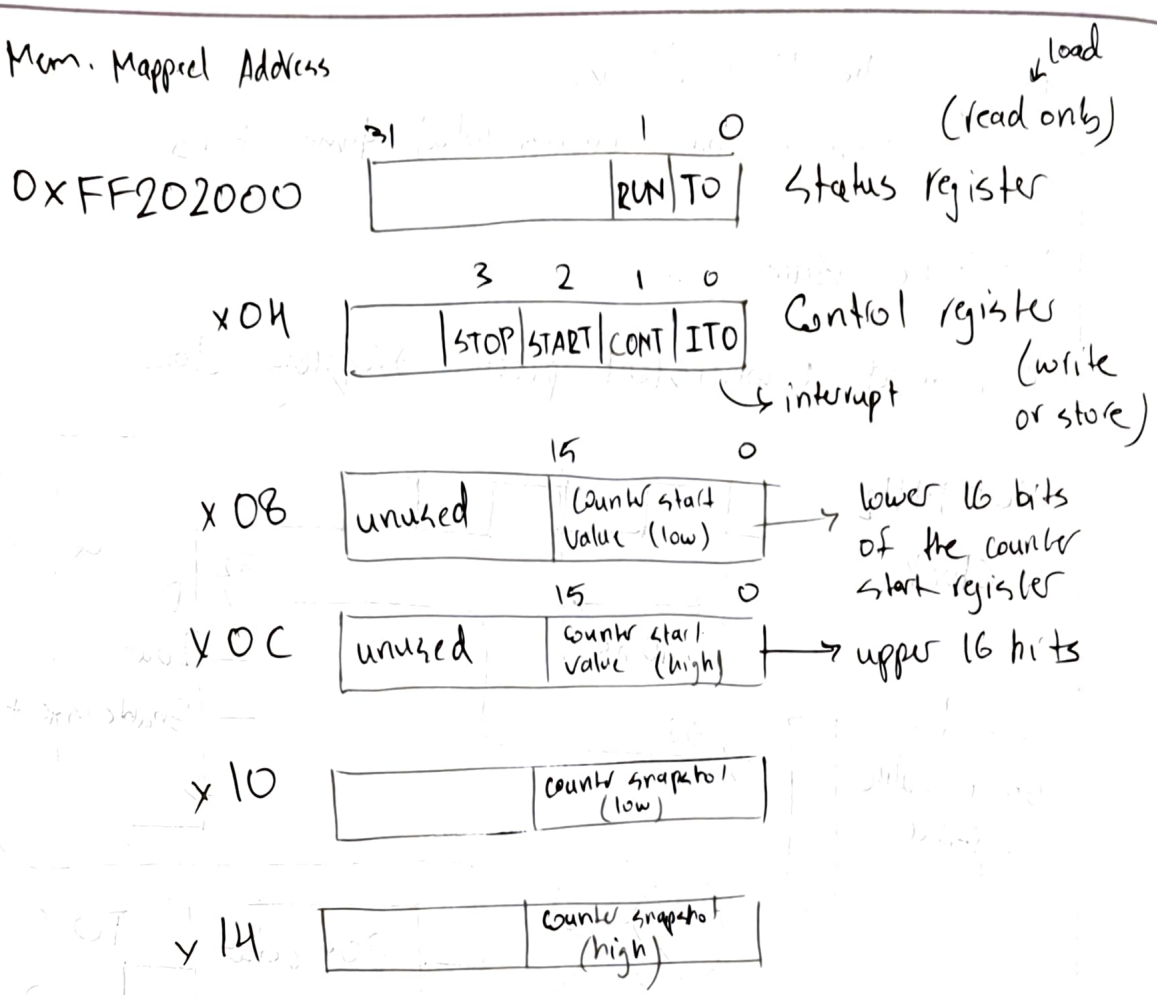


- this is hardware
- the timer I/O device in our DE1-SoC is just like this
- but with memory mapped registers that connect to this ckt

Here is the Interval Timer

- has 6 memory mapped registers

Mem. Mapped Address



write

To use the counter

1) Put the number you want (to count to) into the counter start val register
→ to wait 1 second, 32-bit number needs to be 100 million

2) To count down once, turn on start bit only in control register
to count down continuously, turn on start bit and cont bit

3) To determine if the most recent count down cycle has been reached, must "poll" the TO (time out) bit in status register.
if TO == 1, it means the count down to zero has been reached.

Program to turn LED0 on and off every second

```
.equ TIMER_BASE, 0xFF202000
.equ COUNTER_DELAY, 100,000000   * the number of cycles of 100MHz clock in one second
.equ LED, 0xFF200000
        * set up counter
_start
    la t5, TIMER_BASE
    sw zero, 0(t5)   * clearing the TO bit in the status register in case it's on

    li t0, COUNTE_DELAY

    sw t0, 0x8(t5)   * low 16 bits of starting count

    srli t1, t0, 16   * shift upper 16 bits of the delay into the lowr 16 bits of t1

    sw t1, 0xC(t5)   * put that into the counter start high register

* set up control bits → stop 2, turn on cont and start
    li t0, 0b 0110
                 ↳ cont
                ↳ start
    sw t0, 4(t5)   control register   * store 0110 in to the timer controll register
                                      * causes ~~them to~~ start and go into continuous mode
```

write holes

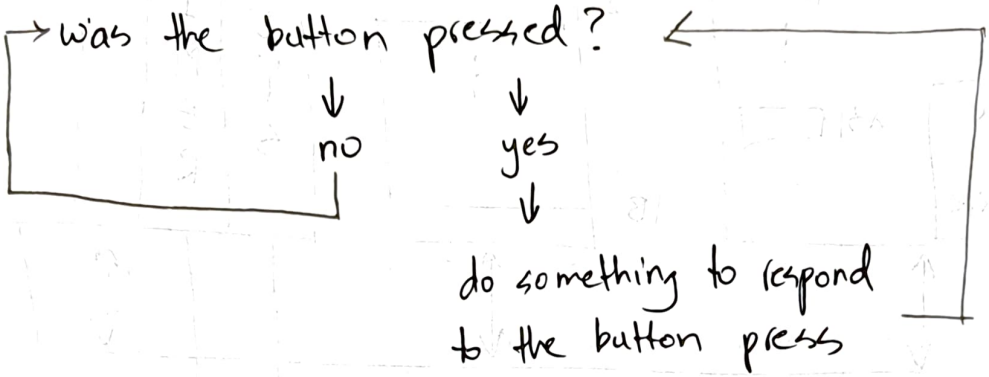
```
la  t6, LED     } * used to toggle LED on/off
li  t2, 1

tloop: sw t2(t6)      * store the current value of light
                        into LED 0 → "set LED0"
       xori t2, t2, 1 * invert bit t2 → turns off LED0
                        ~~nextime~~ next time

* now wait for timer to count down the next time in
* a polling loop

ploop: lw t0, (t5)  * read the time status register AKA
                      * load timer status register
       andi t0, t0, 1 * isolate bit 0 → "TO bit"
       beqz t0, ploop * if TO bit is 0, keep asking/checking it
       sw zero, (t5) * reset to zero (clear the TO bit)
       j tloop
```

Week 4: Lecture 3

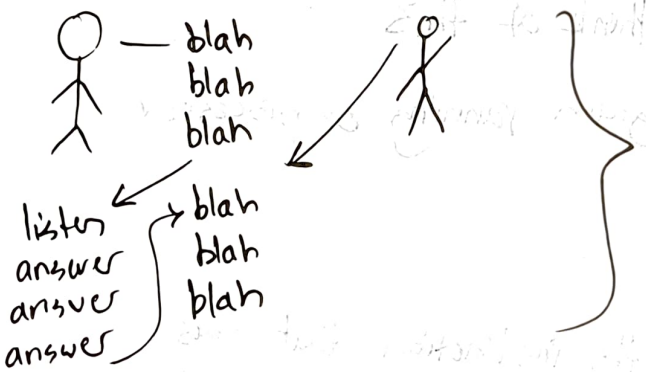
Jan 29, 2024

Today: Interrupt-driven I/O - part 1 intro

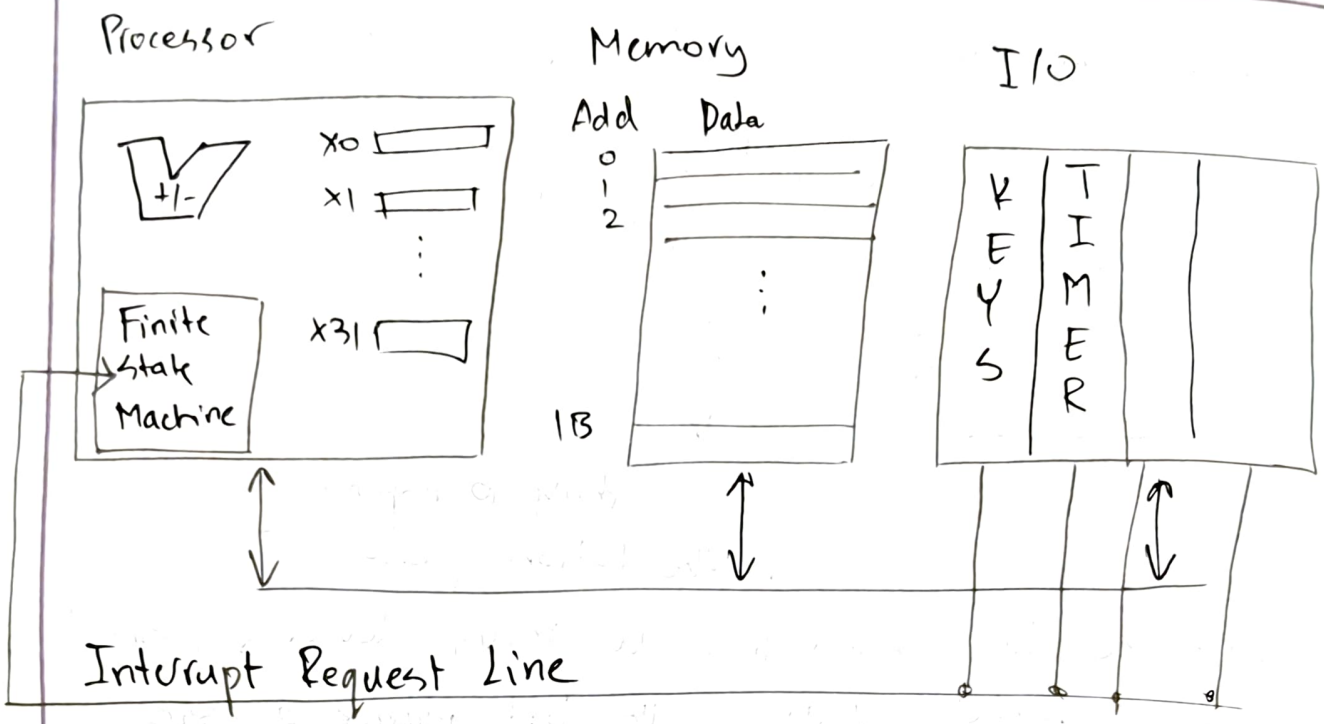
- Polling, though, is a mindless loop:



- can't use this when there are many devices connected to the computer + dealing with bad address accesses
- need a better mechanism
- interrupt driven I/O
- like in a class, me teaching



how does this happen in computer



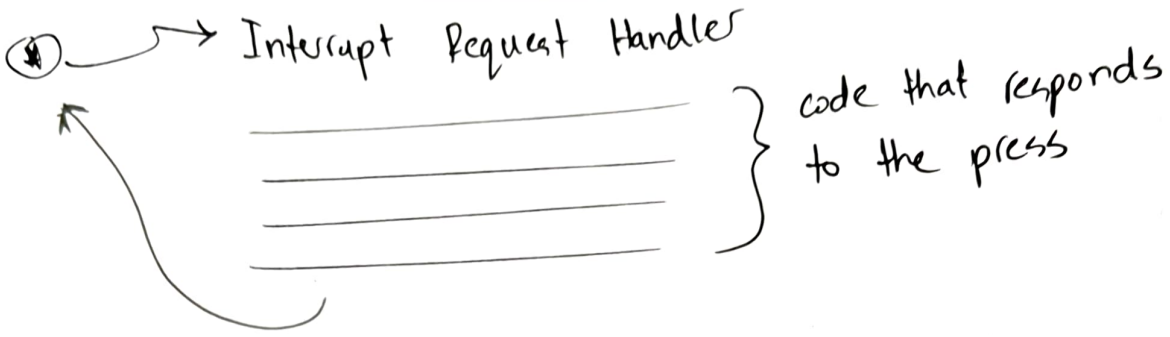
- with interrupts, when a device needs attention (i.e the timer timed out : TO, or button press) — it activates the interrupt request line
- when interrupt happens, think of this
- there is a "main" program running on processor when interrupt happens

Main

```
loop: li t3, 1
      add ...
      ---- : (*)
      ----
      ----
      j loop
```

(*) is the instruction that was executing when, say, a button that is interrupting, is pressed

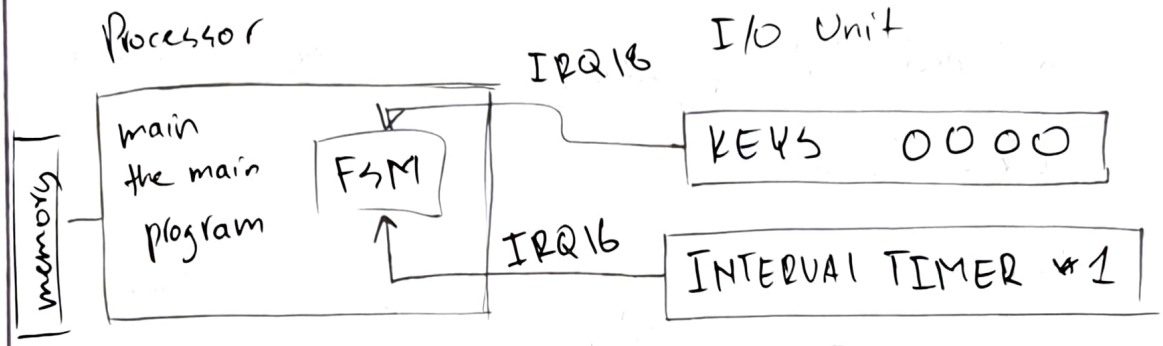
→ Interrupt Request Handler



- when an interrupt happens, the code being executed stops
  → by the FSM in the processor
- → execution is transfered to the interrupt request handler, which is code that you write.
- → that code is executed
- → execution returns to where it was stopped in the interrupted main program AS IF NOTHING HAPPENED.

Week 5 lecture 1 Feb 3, 2025

Today: Interrupts part 2:
- need to synchronize external devices/signals with the software running on the processor



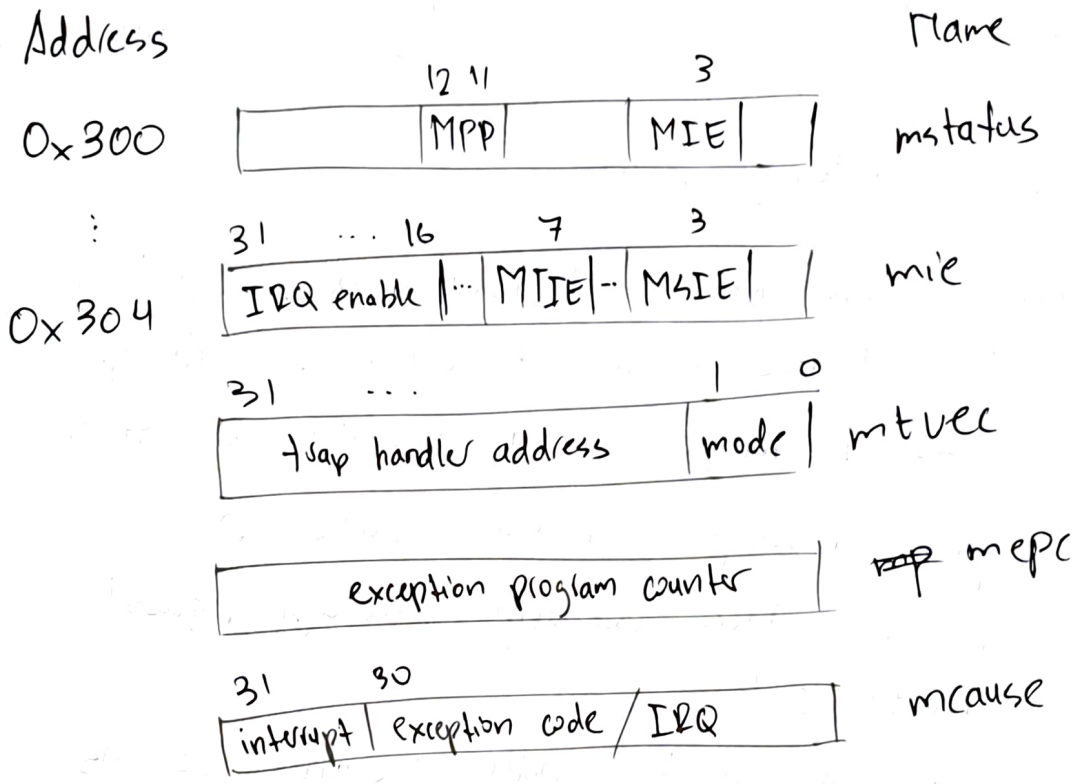
IRQ ≡ interrupt request

- when someone pushes and releases a key that you want (and have set up) to cause an interrupt, let's say it happens at instruction (*), then
  - processor stops (doesn't execute (*))
  - processor starts executing interrupt request handler code
  - when finished the handler, execution returns to (*) with none of its registers "disturbed"

* To make that happen and control it properly, there are two sets of registers —— ① in the processor
② in the devices themselves (memory mapped registers)

- Here are processor control and status registers:

(look at figure 5



- mstatus reg: MIE bit = 1 processors are enabled
= 0 " " disabled
→ global interrupt

- mie reg: IRQ enable bits 16 → 31
→ each bit individually enables/disables a specific IRQ line

- when an interrupt occurs, the mcause register lower bits encode the number of the IRQ line that is interrupting

  (eg: if it was the timer #1, the number 16 would be in mcause)

- here is how you can modify or read these controll and status registers in the processor
- whole set of instructions on pg. 15-16 of <u>Hios_V_Intro</u>
- eg: to set the interrupt handler address into mtvec

```
la t0, interrupt_handler   * t0 ← ...
csrw mtvec, t0             * mtvec ← t0
     ⋮

interrupt_handler : add sp, sp, -12
```

"csrw" control status register write instruction

- general form of control status

```
csrw csr, rs1
```

- control status registers
- which of the above registers
- regular processor register

- can also use an immediate constant

- there is also csrr ← <u>read</u>
- also set individual bits
- " clear " "
- now: write, read, set, clear
  rw rr

- there are other devices (pg. 11 of DE1-SoC-manual) manual from lab 7
  - → interval time : 16
  - → Pushbutton KEY : 18
  - → mouse / keyboard : 22, 23
- finally, recall the registers for KEYS parallel port:

→ ~~X~~ Turn on Interrupt Mask Register bit i to have edge capture register bit i request an interrupt after key i is pressed and released

→ to turn off the request, you must turn off the edge capture bit in the usual way.

Week 5 : Lecture 2 Feb 5, 2025

Today: Interrupts Part 2

Example program to ~~turn~~ toggle LED3 when Key 1 is pushed and released with interrupts

```
.global _start
.equ LEDs, 0xFF200000
.equ KEY_BASE, 0xFF200050

_start: la sp, 0x20000      * init stack — use for saving and restoring registers in the interrupt handler
        csrw mstatus, zero  * mstatus ← 0 ... really sets MIE (bit 3) to zero, which disables all interrupts to processor (in case it was on)
```

* tell the KEYs (key 1) to request an interrupt when pressed and released

* use the KEY parallel port interrupt mask register

```
la t1, KEY_BASE
li t0, 0b0010
sw t0, 8(t1)      * sets bit 1 of interrupt mask reg
sw t0, 12(t1)     * turn off any previous edge capture on key 1
```

2nd level enables

* enable specific IRQ line within the processors

```
li t0, 0x40000    * this turns bit 18 on
csrs mie, t0      * turns bit 18 of mie on → enabling specific line IRQ18
```

↳ csrs: set specific bits, don't change the other

* tell the processor where the handler is in memory

```
la t0, interrupt_handler
csrw mtvec, t0      * mtvec ← t0
```

* finally, enable all processor interrupts by turning on the MIE (bit 3) of the mstatus register

```
li t0, 0b1000
csrs mstatus, t0
```

```
main_loop: li t0, 1
           li t1, 2
           j main_loop
```

* here is where execution "goes" upon an interrupt

↳ before this, mstatus MIE (bit 3) is set to 0 — preventing further interrupts

```
interrupt_handler :
    addi sp, sp, -12
    sw t0, 0(sp)
    sw t1, 4(sp)
    sw ra, 8(sp)
```

} push the registers used in the handler onto the stack

* check to see if the cause of the interrupt was what we expected

(next page)

```
li t0, 0x7FFFFFFF
csrr t1, mcause      * mcause encodes the number of the
                       IRQ line that caused the interrupt
and t1, t1, t0       * zeroes out bit 31 → used for something
li t0,18
bne t1, t0, end_interrupt   * if not equal, somethings
                              wrong, return from interrupt
call KEY_ISR         * KEY interrupt service routine
end_interrupt: lw t0, 0(sp)
               lw t1, 4(sp)
               lw ra, 8(sp)
               add sp, sp, 12
               mret   * return from interrupt: pc ← mepc
                        MIE (bit 3) is
                        set back to 1
```

* check prof notes handout for full note (the one that talks about subroutine KEY_ISR)

Week 5: Lecture 3 Feb 6, 2025

Today: Connecting the C language to Assembly Language

Consider the compilation of a C-language program

C program file.c

```
int main(){
    while ~
    ~~~~
    ~~~~
}
```

→ gcc file.c →

file.s

```
.global _start
_start:
    li ...
    ~
    ~
    ~
main: ~
    ~
```

assembly language statement

file.o

- encoded instructions
- space in memory for data (full contents of memory prior to execution of program

assemble using (gnu as assemble)

- if there are multiple source (code) files → file1.c, file2.c, file3.c

Using Memory Mapped I/O Devices in C

- recall the first memory-mapped program (I/O) from Lec 9
  - copy the switches into the LED

```
.equ LEDs, 0xFF200000
.equ SW, 0xFF200020

_start: la, t0, LEDs
        la, t1, SW
// infinite loop to copy over, over
loop:   lw t2, (t1)   (a)
        sw t2, (t0)   (b)
        ; loop
```

```
int main(void) {
 volatile int *LED_ptr =
        ⋮
```

(a) ⟷ value = *SW_ptr;

(b) ⟷ *LED_ptr = value;

- volatile means, to the compiler, "don't do fancy optimizations on this variable that would break the memory mapped I/O → (x)

```
int main (void) {
  volatile int *LED_ptr = 0xFF20000;
  volatile int *SW_ptr = 0xFF200020;
  int value;
  while (1) {
    value = *SW_ptr; // dereferencing a pointer, does what lw statement (a)
    *LED_ptr = value; // same as the sw statement (b)
  }
}
```

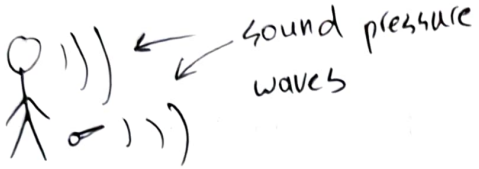
(x)

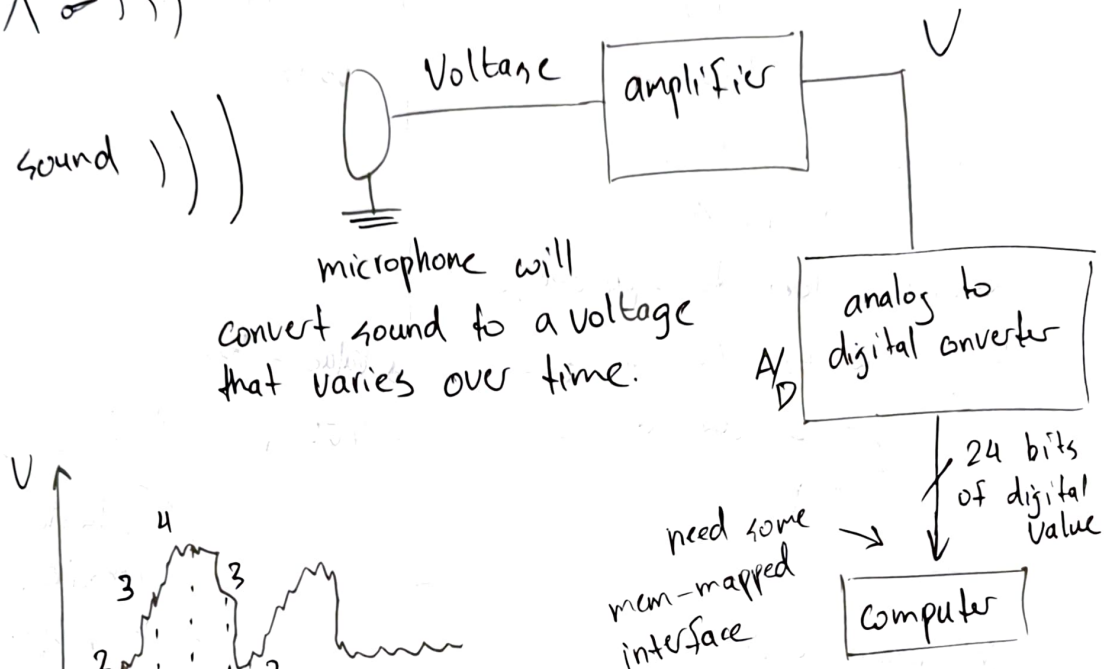
- an optimization that puts it in a register
- TBD: put it in cache

Week 6: lecture 1 Feb 10, 2025

Today: Audio (sound) I/O - building on this course + others

Music / Audio





100s of microseconds
= 10kHz sampling rate

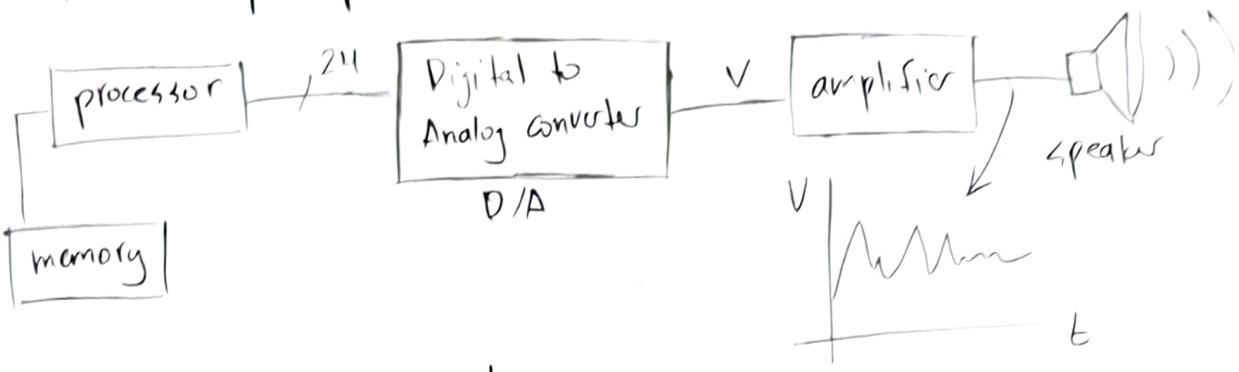
- the ADC converts the time varying voltage into a sequence of digital (binary) values
  → sampled at some rate: eg 10kHz (10k samples/second)

Coming Out of ADC

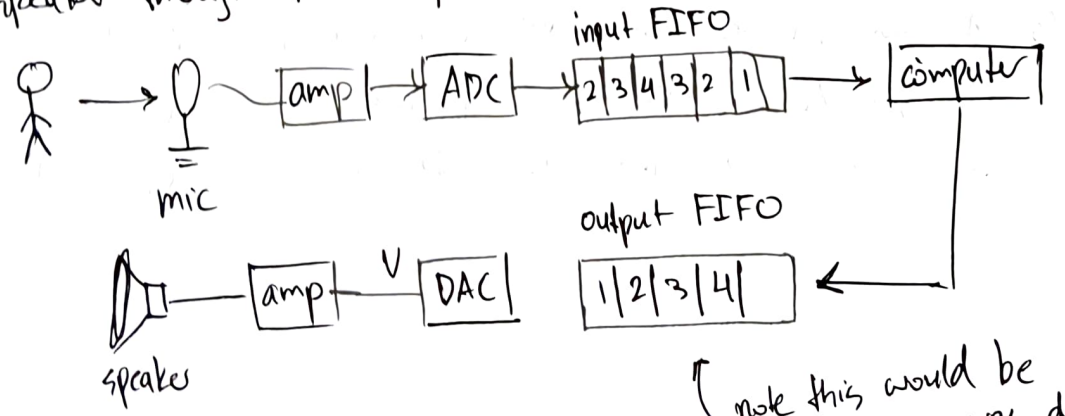
time	Value
100 µs	1
200 µs	2
300 µs	3
400 µs	4
500 µs	3

numbers encoded in binary → can store these in memory → permanent memory

- the sound output proceeds in the other direction



- the DE1-SoC has a microphone input jack and speaker output jack
- there is a CODEC chip connected to the jack that performs all the amplification, and A/D, D/A
- it samples sound input at 8kHz (125 μs period)
- the output samples also come out @ 8kHz
- → the processor might be busy and miss an input sample (messing up sound) or be late sending an output sample to the D/A
- → for both these reasons, the input and output need to be buffered with hardware memory that acts like a queue → a first-in-first-out structure (FIFO)
- → called a hardware FIFO
- consider the system that simply connect the microphone to the speaker through the computer

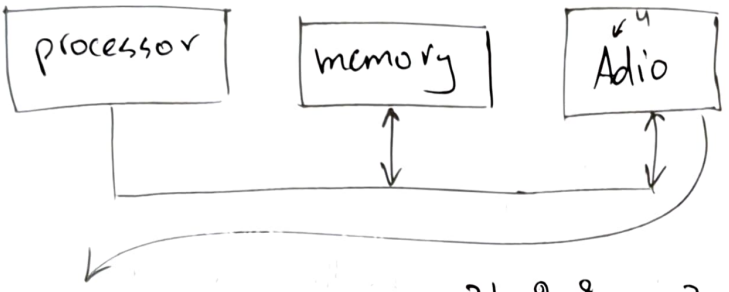


(note this would be mirrored if diagram was drawn in 1 line

Week 6: lecture 2 Feb 12, 2025

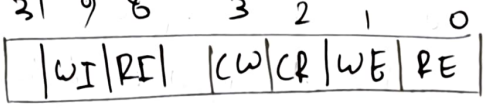
Today: Audio Part 2: register level control and 'sound' data flow

- Computer connects to the audio input/output unit through a memory-mapped interface

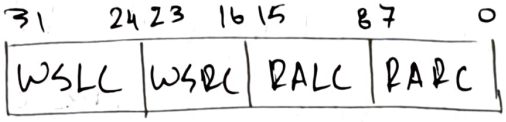


here are the specific memory-mapped registers that connect to the input and output FIFO's of the audio unit

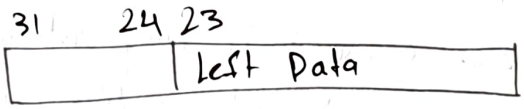
@ address 0xFF203040



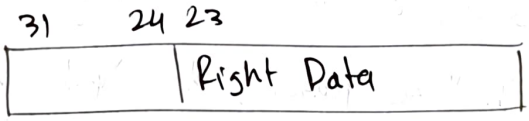
control/status



4H fifo space



48



4C

Two channels,
Two output FIFO's
Two input FIFO's

1) Left Data and Right Data

- if you load/read from these registers, it gets the next sample from the associated (left or right) FIFO; when that happens, that sample is removed from the input FIFO (so that you get the next sample when you next read)
- if you store/write into these registers, this sends that value as a sample into the left or right output FIFO

2) FIFOspace 4 fields tell you how occupied the 4 fifo's are

* RALC is how many of the left input fifo's entries are occupied
* RARC " " " " " right → (of 128)
* WSLC " " " " " left output fifo's are empty
* WSRC " " " " " right " " " "

3) The Control/Status register bits work as follows

* control: • write a "1" into CR causes the whole (both) input fifo's to be cleared/emptied
  • write a "0" into CW " " " " output " ...
* Status: RI goes to 1 if the input FIFO's are ≥ 75% occupied
  WI " " " " " output FIFO's are ≤ 25% occupied
* Control: RE = 1 means request interrupt when RI = 1
  WE = 1 means request an interrupt when WI = 1

Program in C to connect mic to speaker

```
#define AUDIO_BASE 0xFF203040
int main(void) {
    volatile int* audio_ptr = (int *) AUDIO_BASE
    int left, right, fifospace;
    while (1) {
        fifospace = *(audio_ptr + 1)
```

← adds 4 with pointer arithmetic

⋮

cont'd on next page

```
if((fifospace & 0x000000FF)>0){      // right input fifo is not empty
    // load the two input channels
    left = *(audio_ptr+2);           } same as lw t0, 8(t1)
    right = *(audio_ptr+3);          } 12(t1) contains 0xFF203040
    *(audio_ptr+2) = left;           }
    *(audio_ptr+3) = right;          } store into left and right data → output FIFO
}
}
}
```

* can check just the right one (or left) since they're filled in at ~~the~~ the same rate

Week 6 Lecture 3 Feb 17, 2025

Today: Lab 6 → Making Square Waves + Echo Box + C structure

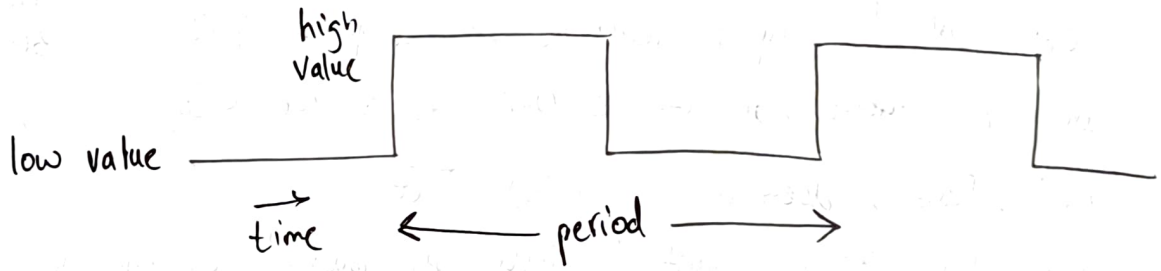
Lab 6:

Part 1: * get used to C in memory-mapped I/O
* dereferencing pointers + pointer arithmetic
+1 integer pointer is +4 bytes/addresses

Part 2: * microphone → computer → speaker (code given)

Part 3: * synthesize sound: square wave whose frequency is adjustable with the switches from 100 Hz to 2 kHz (cycles/second)

What is a Square Wave?



low: 0
high: 1 1 1 1 1 . . . } 24-bits you figure it out

* need to know the output sample rate to know how many samples to generate in each half period
→ sample rate is 8 kHz
→ period = 125 μs
* how to know when it's okay to send the next sample to the output FIFO → look at WSRC or WSLC in the fifospace register

Part 4: *make an echo box

Echo → to both output the current input but also the input that came in some time ago
→ suggest 0.4 seconds ago

→ time is measured in samples → each sample comes in every 125 μs
eg: 1000 samples ≡ 125 ms = 0.125 s

→ when echo "comes back" after the delay it is quieter

→ we say it is "damped" by a damping factor D, where $0 < D < 1$

Let's call the input sequence of samples $I(t)$ and the output samples we want to compute $O(t)$

1st $I(0)$
2nd $I(1)$
3rd $I(2)$
⋮
100,000 $I(?)$

Aside: Part 2 does this : $O(t) = I(t)$

Now with echo, we want to have the output to include the input + the damped sequence from N samples ago

$$O(t) = I(t) + D \times I(t-N)$$

N → # of samples to get 0.4 s delay

$I(t-N)$ → must store the previous N samples in an array

to get the damped echo N, 2N, 3N... samples ago, store the output $O(t) = I(t) + D \times O(t-N)$

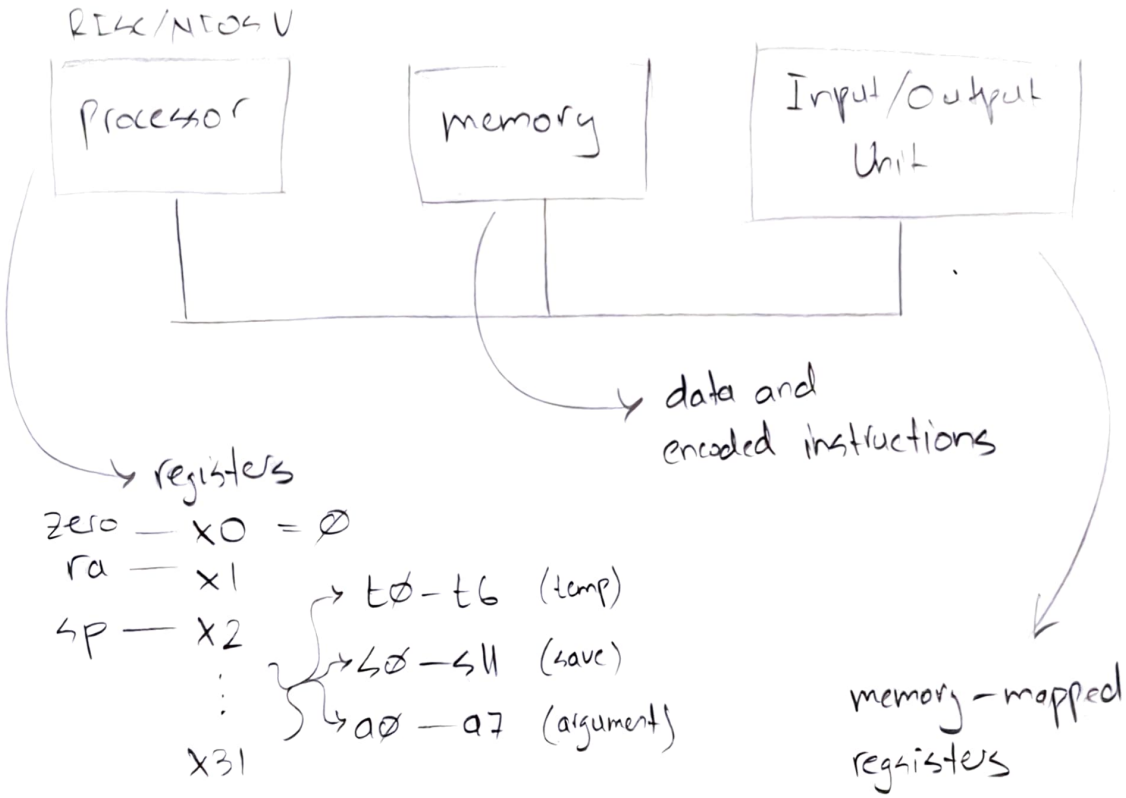
- only need to store N values of

Week 7: Lecture 1 — Feb 24, 2025

- labs 3, 4, 5, 6 → labs 1/2 pretty basic

Review

- Computer Architecture



Memory Organization

- byte addressability
- words, half-words, bytes
- order of bytes within a word (little Endian)
- translate binary to hexadecimal

Instructions for RISC/NIOS V

- Loads (read) and stores (write)
  - lw, lh, lb
  - sw, sh, sb
- indexed addressing
  - eg: lw s0, (s1)
- indexed with offset
  - eg: lb t0, 3(t1)

  - cannot access word at address not divisible by 4
  - cannot acces half word @ address not divisible by 2

- mv, li, la → can turn into 2 instructions depending on size of constant
- add, addi, sub ⟶ no "subi"
- shift instructions: srl, srli, sll, slli
- conditional branches
  - eg: bge x4, zero, loop
    - bxx t4, t3, elsewhere
      - xx is the condition
- unconditional jump: "j"
- call } subroutine
- ret }

- if you jump to a subroutine instead of call, it won't know where to return

Assembler Directives

* .word .byte .hword .skip (.text .data)

(setup things in memory)

{ .know (well

Subroutines

* how call and ret work
* how to transmit and recieve information to/from a subroutine ⟶ MIOS V calling convention.

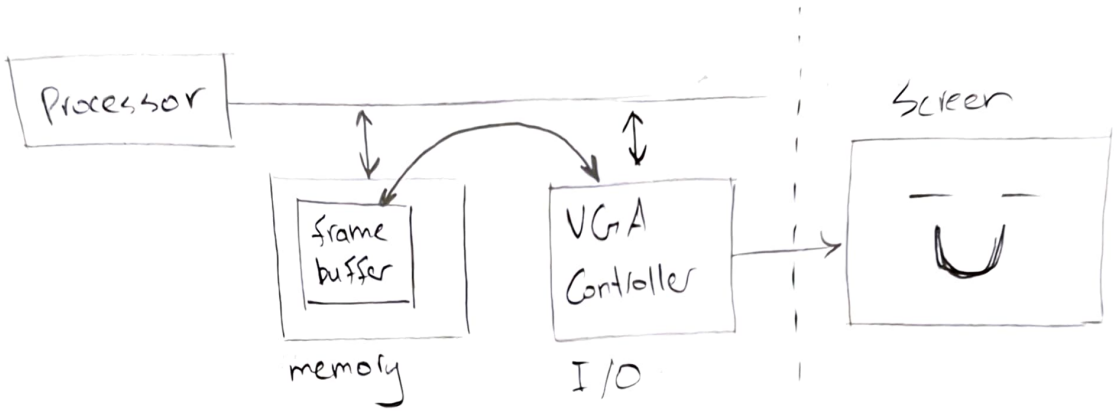
Convert this program to assembly (obey MIOS V calling conv.)

```
main() {
    int N = 10;
    int answer = findsum(N);
}
int findsum(int N) {
    int sum = 0;
    while (N != 0) {
        sum = sum + N;
        N = N - 1;
    }
    return (sum);
}
```

```
-start: la sp, 0x20000  ** need stack
        la s0, N
        lw a0, (s0)  ** a0 holds the 1st argument
        call findsum
        sw a0, 4(s0)
end: j end

findsum: addi sp, sp, -8  ** push ra & s0 onto stack
         sw ra, 4(sp)
         sw s0, 0(sp)
         li s0, 0  ** sum=0

while: beqz a0, endloop
       add s0, s0, a0  ** sum=sum+N
       addi a0, a0, -1  ** N=N-1
       j while

endloop: mv a0, s0  ** ans goes into a0
         lw ra, 4(sp)
         lw s0, 0(sp)
         addi sp, sp, 8
         ret

N .word 10
Answer .word 0
```

Graphics

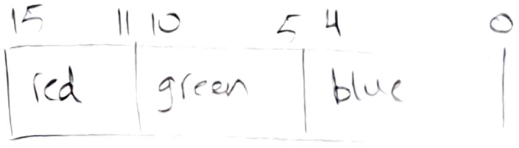
- drawing on the screen
- here is the DE1-SoC graphics system



- software draws the picture you want into the frame buffer, which is specific set of memory locations
- the VGA controller reads the frame buffer and renders it onto the screen
- To use this system you need to know

1) what is a picture made of?
   - pixels - picture elements
   - many, all across the screen at different (x,y) location
   - each pixel's colour is determined by a mixture of red, green, and blue (RGB) - diff. amounts of each get you diff. colours
   - in the DE1-SoC graphics, each pixel is 16 bits

Blue - 5 bits
Green - 6 bits
Red - 5 bits
} organized like this



white - all 1's

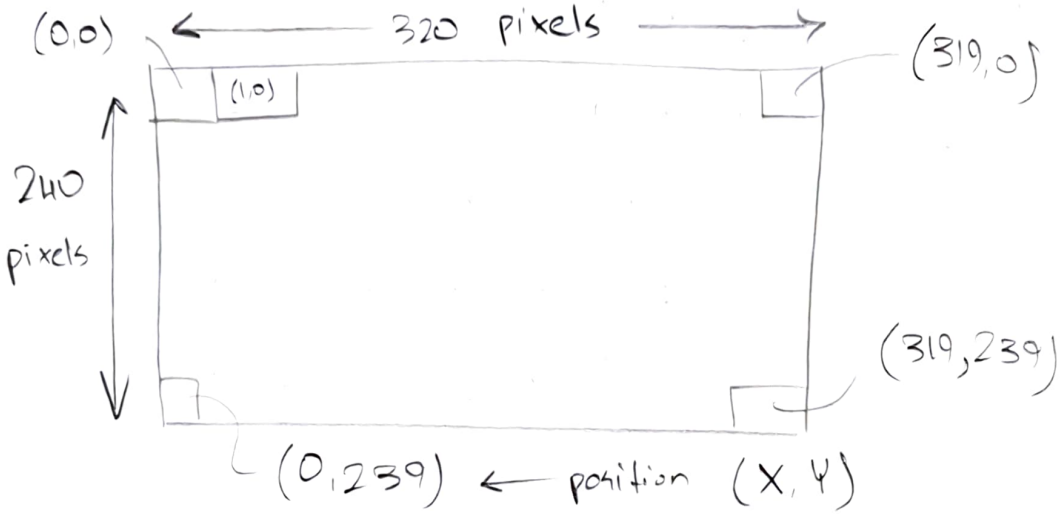
black - all 0's

green - 0x07E0

blue - 0x001F

red - 0xF800

2) Coordinate system of the screen



3) Need to map (X, Y) location (2D) of a pixel to a specific (1D) address in the frame buffer

* Given the (X, Y), find address

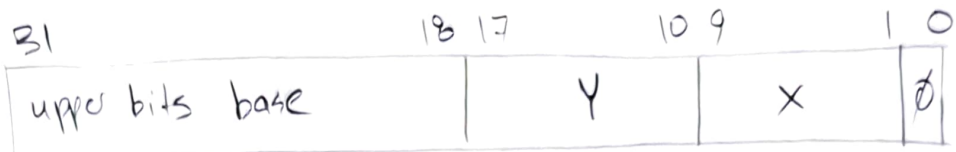
  $0 \le x \le 319$ $0 \le y \le 239$

* need to know where frame buffer begins (called <u>base</u> address)

* DE1-SoC default base: 0x08000000

* to convert (X, Y) to the address, as follows

short int, 16 bits



* can compute address given X and Y and base :

```
address = base + (y<<10) + (x<<1);
```

eg: (X,Y)
(0,0) → @ 0x08000000

(1,0) → @ base + (0<<10) + (1<<1) = base + 0 + 2
= 0x08000002

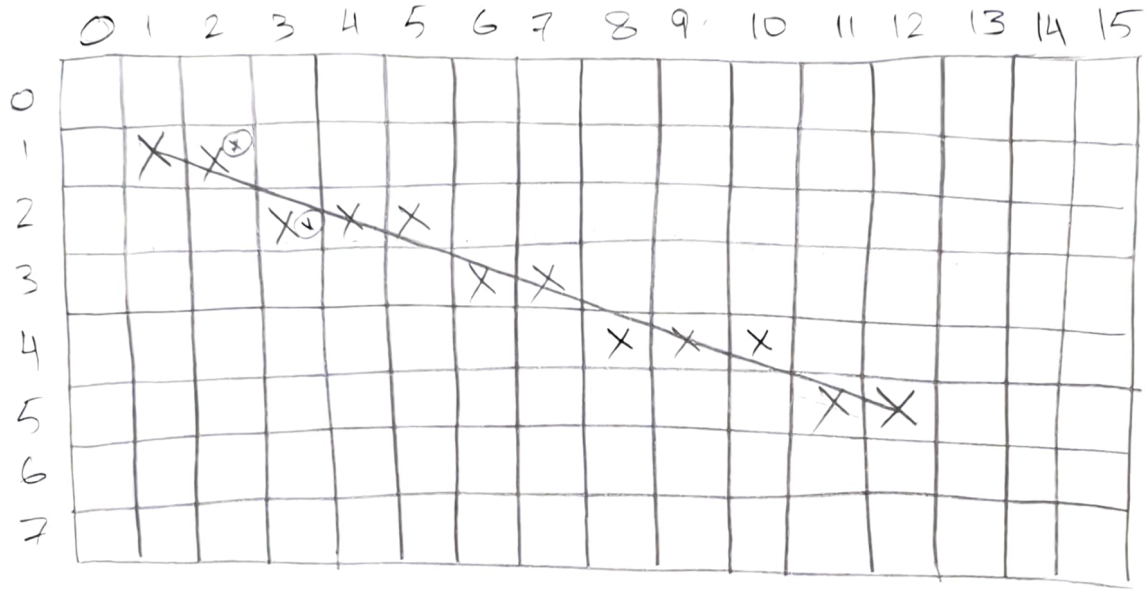
(1,1) → .... (1<<10) + (1<<1)
= base + 0x400 + 2 = 0x08000402

* showed the conversion of an (X, Y) location on screen to address in the frame buffer where that pixel's 16-bit colour is stored.

Week 7: Lecture 3 Feb 27, 2025

Drawing A Line

- is harder than you think
- because of discrete resolution of the screen
- consider a line from $(X_1, Y_1) = (1,1)$ to $(X_2, Y_2) = (12,5)$



- must choose pixels along the line to turn on
  - ↳ the closest ones
  - ↳ first pixel is given $(1,1)$ — turn on
  - ↳ next pixel: need to determine the slope of the line

    from $(X_1, Y_1) \rightarrow (X_2, Y_2)$: slope $= \frac{Y_2 - Y_1}{X_2 - X_1}$

    slope $= \frac{5-1}{12-1} = \frac{4}{11}$ for our line

  - ↳ $X = X_1 = 1$, $Y = Y_1 = 1$ } turn on pixel $(1,1)$

↳ $X = 2$ ("walking" one pixel to the right)

$Y = Y_1 + \text{slope}(X - X_1) = 1 + \frac{4}{11}(2-1) = 1\frac{4}{11}$

⇒ round down $1\frac{4}{11}$ to 1 ⇒ $X = 2$, $Y = 1$ ✱

↳ $X = 3$

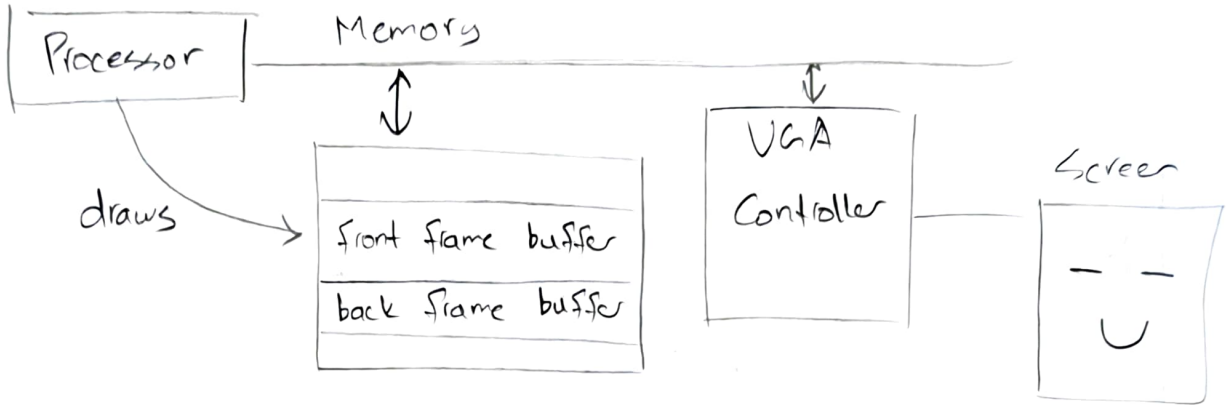
$Y = Y_1 + \text{slope}(X - X_1) = 1 + \frac{4}{11}(3-1) = 1\frac{8}{11}$

⇒ round up $1\frac{8}{11}$ to 2 ⇒ $X = 3$, $Y = 2$ ✱

* there's a problem: steep lines will not be filled in
  ↳ switch to walking/stepping with Y
  ↳ called Bresenhan's Algorithm

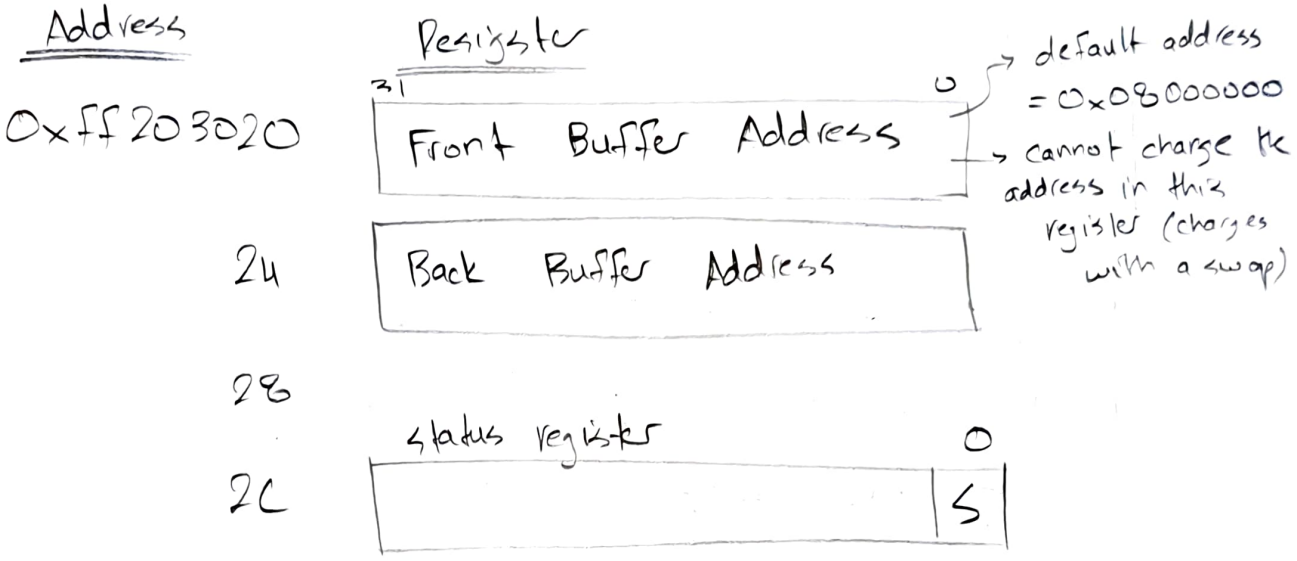
Lab 7 Part 2: Animation

* horizontal "moving" down, up, down, up...
* to animate the line
  ↳ draw white on the line
  ↳ wait for some ? time
  ↳ erase line (draw black on it)
  ↳ draw white line in a new location
* the VGA controller renders the frame buffer every $\frac{1}{60}^{th}$ second
  ↳ that line should be moved 1 pixel down/up every 1/60 second
  ↳ we'll wait for this timing to happen to draw the next line
  ↳ VGA standard when the rendering is finished a "vertical sync" happens



* animation is several frames/pictures in sequence
* software draws picture/frame into frame buffer
* VGA controller renders from frame buffer on to screen
* with just one frame buffer, the rendering is disturbed by the drawing
* per last lecture, we need two frame buffers
  - One that is being used to do the rendering of the current frame → front buffer
  - One that is being used to draw the next frame → back buffer
* Once the rendering is done and the drawing is done, we swap the front buffer and the back buffer
  - then render the front, then draw the back

The processor talks to the VGA controller through these mem-mapped registers



the front and back registers both start = 0x08000000

↳ single buffering

to use the second buffer, you must put a different address into the back buffer register.

↳ address must be in range 0x0 → 0x03ffffff

↳ other addresses are not accessible to VGA controller

KEY Point: the swap back ↔ front is just the swap of the addresses in these two registers

↳ swap done by VGA controller itself when the time comes

the front buffer is rendered every 1/60 of a second

* to cause a swap you do following:

1) Software writes a 1 into the front buffer register
   ↳ this tells the controller that you want to do the swap process
   ↳ causes the S bit in status register (bit 0) to be set to 1

2) swap doesn't happen right away; controller waits until rendering done
   ↳ that point in time is called vertical sync (Vsync)

3) Your code must poll the S bit of status reg to wait for it to turn into 0
   ↳ Vsinc render done

* function to wait for the Vsinc

```
void wait_for_vsync(){
    volatile int* fbuf = (int*) 0xff203020   # base of VGA controller
    int status;
    *fbuf = 1;  // start swap cycle per 1)
    status = *(fbuf+3)  // read status register
```

```
while((status & 0x01) != 0)
    status = *(fbuf+3)
}
```

polling loop waiting for status bit = 0

Lab 7 Part 2 – single buffer — use to know when to change single frame

3 – double buffering — " " " " draw new frame

Week 8: Lecture 2 March 5, 2025

Lab 7 Part 3 Goals:

* draw N = 8 boxes of size 2×2 pixels in random initial positions on screen
* each box has random colour
* then: → make the boxes move in a random direction and bounce off edges of screen
  → each box should be joined to another (linked in a chain)

Guidance

1) Use arrays for everything associated with a box
   ↳ location on screen: int x_box[N], y_box[N];
   ↳ colour of each box: short int colour_box[N];
   ↳ direction and amount box moves each frame:
   * int dx[N], int dy[N];
   * only use diagonals
   * only move one pixel per frame in x and y
   * dx ∈ {-1, +1} dy ∈ {-1, +1}
   * 2×2 = 4 directions, randomly choose -1 or +1 for each dx and dy

8 directions
↳ 4 directions

   ↳ use rand() to make all the "random" decision
   * part of stdlib → ** include <stdlib.h>

↳ rand() returns a number 0 and $2^{32} - 1$

$$dx = \Big(\big(\underbrace{(rand() \,\%\, 2)}_{0 \text{ or } 1} \big) \times 2 \Big)_{0 \text{ or } 2} - 1$$

-1 or 1

↳ random initial x and y

↳ random colour

* choose carefully

Animation Loop

* recall: always "draw" (aka set the colour of specific x, y pixels) into the back buffer

```
// Major outer loop
while () {
    draw(); // draw the frame into the back buffer
    wait_for_vsync();
    back_buffer = *(fbuf + 1); // gets the value in back buffer
                               register = address of back buffer
}
```

draw() {

// erase the previous content of back buffer

① Easy but slow : erase every pixel 320 × 240

② Harder but faster : erase only what was drawn into this buffer 2 frames ago

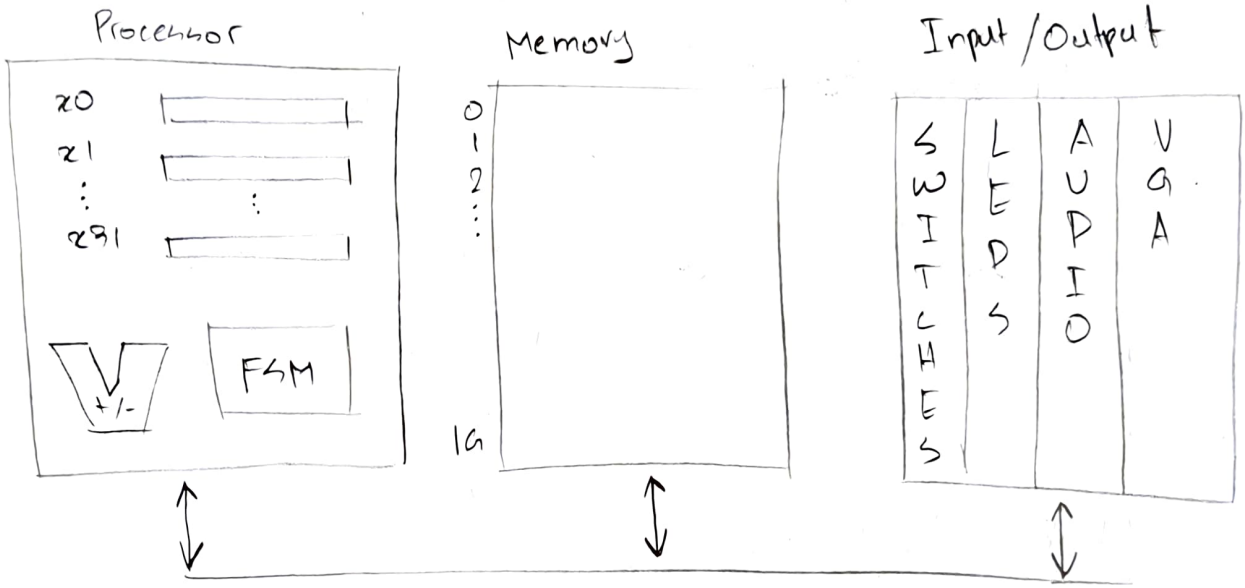
for each box i

```
draw_box(i);
draw_line(i, (i+1) % N);
// update box position for next frame
X_box[i] += dx;
Y_box[i] += dy;
// plus check for the screen edges
```

Week 8: Lecture 3 Mar 6, 2025

Digital Design of a Processor

- dive into the processor
- then jump out and talk about some aspects of computer architecture
- computer architecture so far



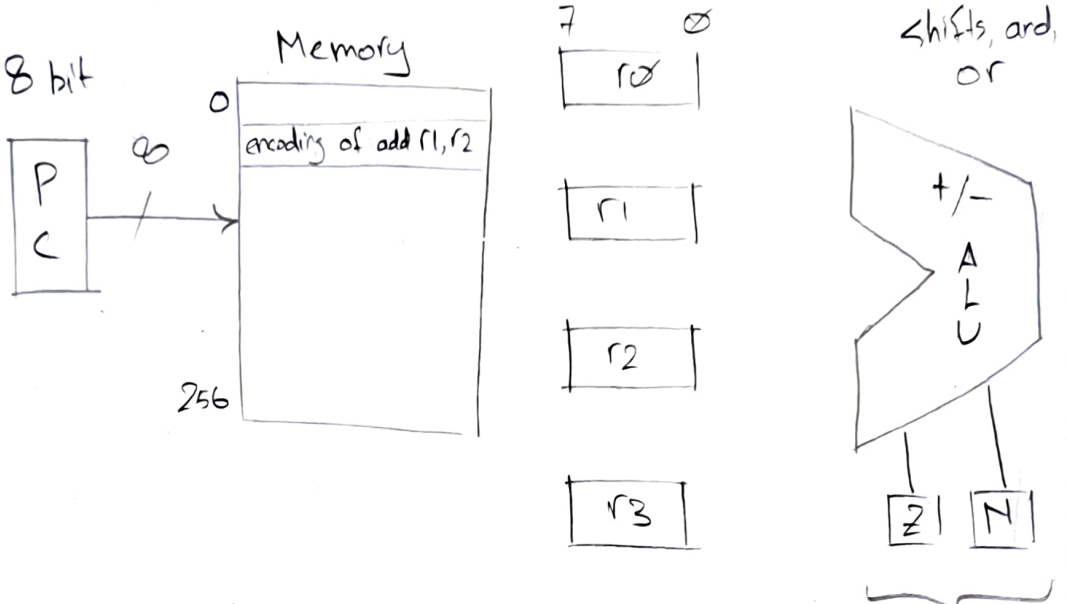
what
why
how
ECE334
software guardian
physics guardian

- discuss digital design of simple processor
  - has 4 registers - r0, r1, r2, r3 (V.S MIPS, 32)
  - each register is 8 bits wide (word size is 8) (V.S. MIPS, 32)
  - has only 256 bytes of memory (V.S. ≥ 1 Gbyte MIPS V)
  - the program counter is 8 bits because $2^8 = 256$ ← all the memory
  - 10 instructions: will show the encoding
  - SimP instructions have 2 operands
  - Z bit = 1 if the result of most recent instruction = 0
    = 0 " " " " " " " ≠ 0
  - N bit = 1 if result is negative
    = 0 " " ≥ 0

Z and N bit are used in conditional branch instructions

Consider SimP's add instruction

* eg: add r1, r2 ⇒ r1 ← r1 + r2
* how to build that?

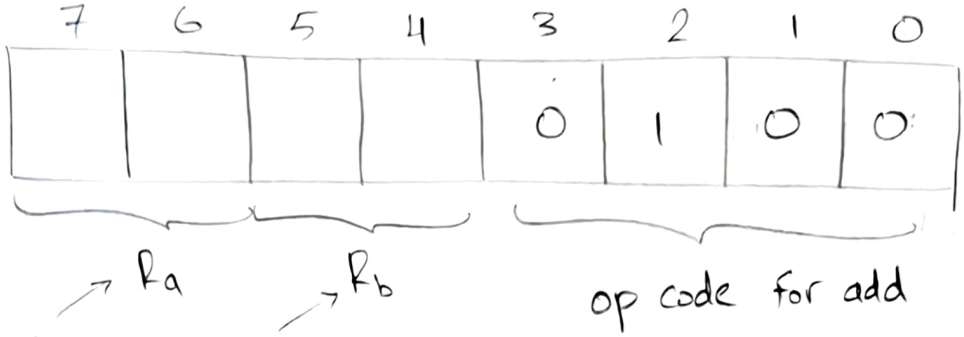


Bak to Add

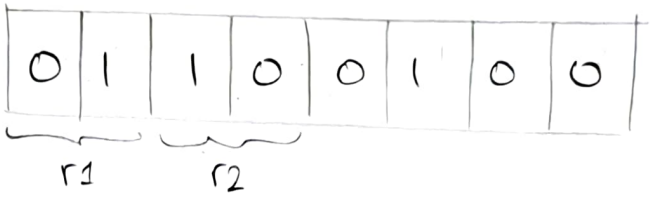
* general form: add Ra, Rb ⇒ Ra ← Ra + Rb
* both Ra & Rb can be any of r0, r1, r2, r3

1) Fetch instruction from memory
   ↑ the encoded
2) Figure out what it's supposed to do (in this case, add)
3) Compute Ra ← Ra + Rb
4) if (Ra == 0) Z = 1 else Z = 0
5) if (R < 0) N = 1 else N = 0
6) add 1 to PC: PC ← PC + 1

Instruction Encoding of add R_a, R_b into 8 bits



- eg: add r1, r2 ⟶

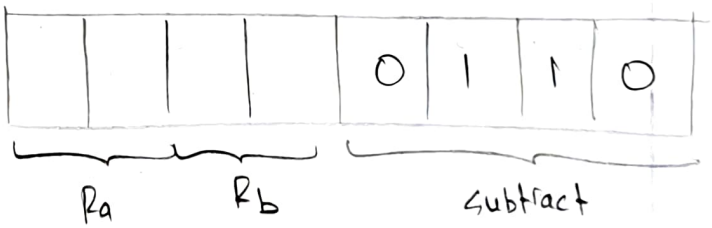


⇒ 0x64 (hex)

Subtract

- sub R_a, R_b ⇒ $R_a \leftarrow R_a - R_b$ Encoding

1) Fetches instruction
2) Computes $R_a \leftarrow R_a - R_b$
3) Set Z and N
4) $PC \leftarrow PC + 1$



- eg: sub r3, r2 ⟶

⇒ 0xe6 (hex)

NAND Instruction

* similar to add, sub

nand Ra, Rb where $Ra, Rb \in \{R_1, R_2, R_3, R_4\}$

* Execution of this

1) Fetch of instruction from memory
2) Compute: $Ra \leftarrow (Ra \barwedge Rb)$ ~ bitwise NAND
3) condition codes are set: $Z = 1$ if res $= 0$
   $M = 1$ if res < 0
4) $PC \leftarrow PC + 1$

* The encoding of NAND is

7	6	5	4	3			0
R	a	R	b	1	0	0	0

which of the 4 registers are Ra, Rb — opcode

eg: nand R2, R1

1	0	0	1	1	0	0	0

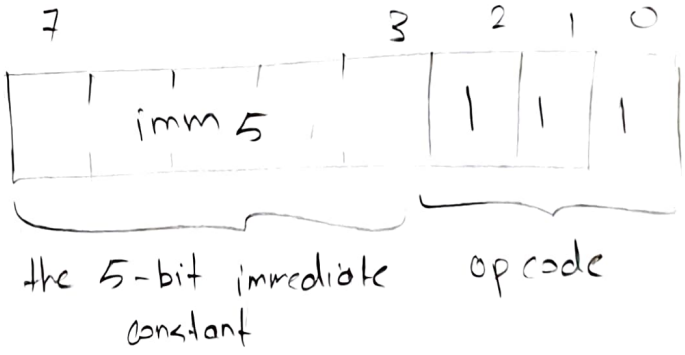
$= 0x98$ (hex)

The OR immediate Instruction

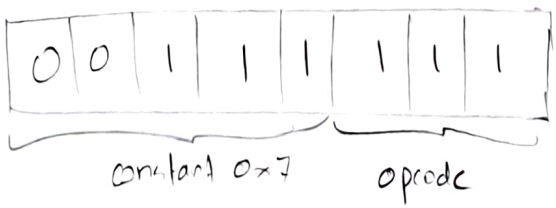
- recall that immediate operands / constants are encoded in the actual instruction
- general form: `ori imm5` ← stranger, no room to specify reg, so we use only R1 by definition
- Execution:
  1) Fetch
  2) R1 ← R1 logical OR ZE(imm5)
  3) Set Z and N
  4) PC ← PC + 1

ZE means zero-extend... add 3 zeroes to the left of imm5

- Encoding of ori:



- eg: ori 0x7



= 0x 3F hex

Memory Instructions

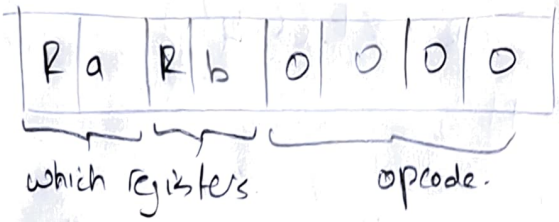
load Ra, (Rb) ← similar to lw Ra, (Rb)

Execution:

1) Ra ← Mem(Rb) # Ra gets contents of memory address given by Rb
2) PC = PC + 1

⇒ Z and N are not set

Encoding:

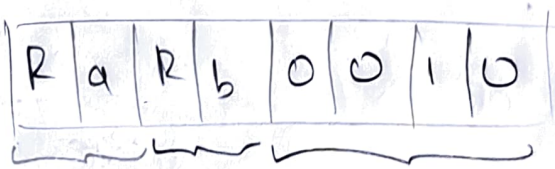


store Ra, (Rb) ← similar to sw Ra, (Rb)

Execution:

1) Mem(Rb) ← Ra
2) PC = PC + 1

Encoding:



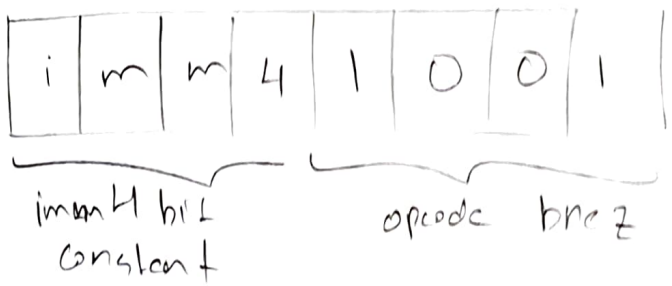
proj: use of a proc.

Conditional Branch Instructions

- Recall in MIOS V bnc x5, x0, loop
- SimProc: "branch if not zero" (bnz)
- "goes" to a diff. address if "" is not zero
  - ↳ the result of the previous operation
  - ↳ (know this if z = 0)
- general form: | bnz imm4 |
- Execution:

SE means sign extend the 4 bit imm4 constant to 8 bits

1) if (z == 0) PC ← PC + SE(imm4)
   else PC ← PC + 1



- bpz opcode is 1101 (branch if positive)

assembler figures out what imm4 should be

```
1
2
3
4
5
6
7
8   bnez 2
```

$\leftarrow pc = 8, \Rightarrow 8 + imm4 = 2$

$imm4 = -6$

$PC_{new} = PC_{old} + SE(imm)$

$0 = 3 + SE(imm4)$

$SE(imm4) \Rightarrow -3$

$(imm4) = -3 = 1101 \Rightarrow SE(imm4) = 1111$

$3 \rightarrow -3 \rightarrow SE(-3)$

$0011 \rightarrow 1101 \rightarrow 1111\boxed{1101}$

Week 9: Lecture 2 Mar 12, 2025

Let's Write + Assemble Small Program

assume R0 and R1 are and mem are set already

subtracts 2 numbers in memory and branches back to beginning if result of sub is not zero

Address		Instruction	Encoding Ra Rb opcode
0	loop:	load R2, (R0)	10 00 0000 = 0x80
1		load R3, (R1)	11 01 0000 = 0xD0
2		sub R2, R3	10 11 0110 = 0x....
3		bnez loop	11 01 1001 = 0xD9

Last Instruction: Shift Left / Right

shift L/R Ra imm2

Execution

1) if (L/R == 1) Ra = Ra << imm2 ← shift left

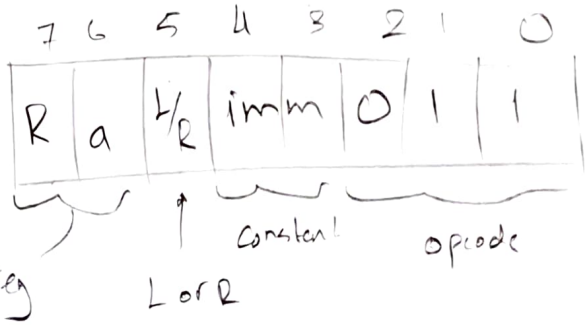
   else Ra = Ra >> imm2

2) set Z, N as usual

3) PC ← PC+1

L/R = 1 shift left

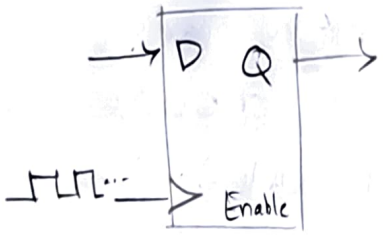
= 0 " right



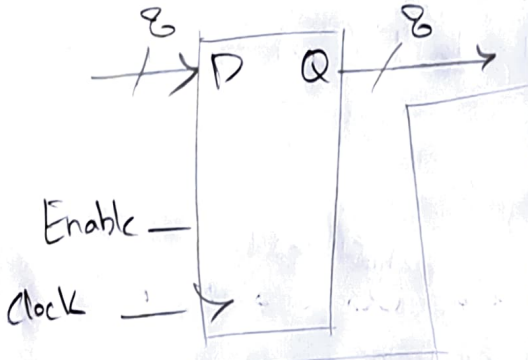
Recall from ECE241

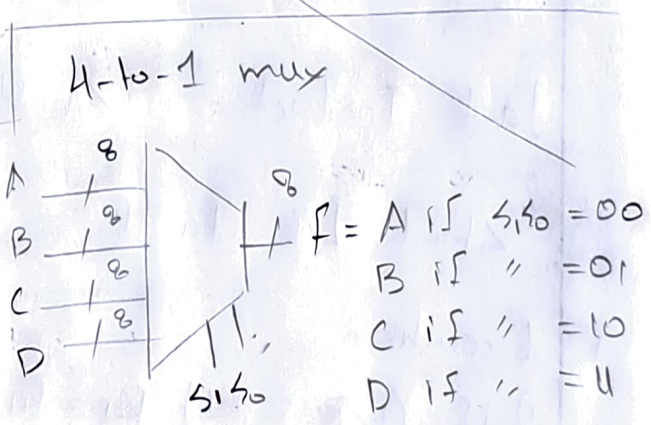
1) Recall the flip-flop : if Enable = 1, on a posedge of clock, Q ← D

if Enable = 0, Q doesn't change



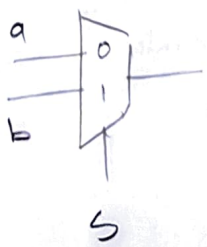
- Can make an 8-bit register with 8 of these and common Clock and Enable





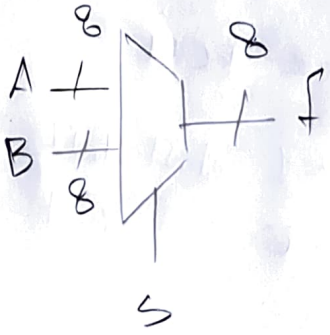
2) Recall Multiplexer → mux

1-bit 2-to-1 mux



f = a if s = 0
= b if s = 1
= $\bar{s}a + sb$

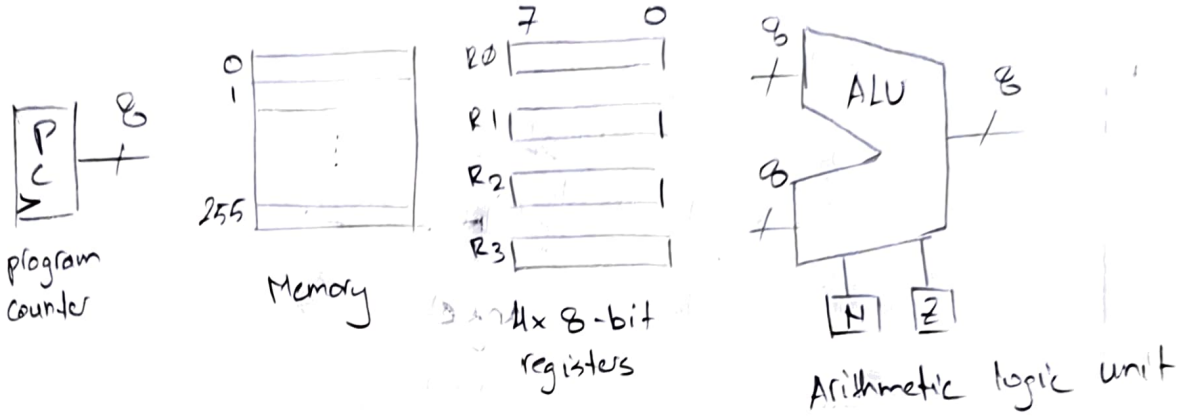
8-bit 2-to-1 mux



- all 8 wires on 8 2-to-1 1 bit muxes selected by one signal s

Week 9: Lecture 3 Mar 13, 2025

Recall "elements" of Processor

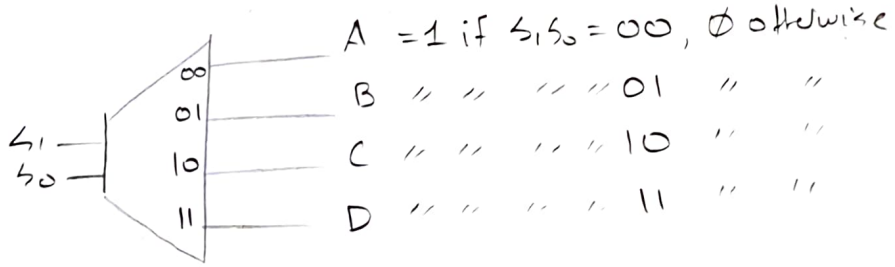


Instructions :

- add Ra, Rb
- sub " "
- nand " "
- ori imm5
- bnz imm4
- shift Ra, imm2

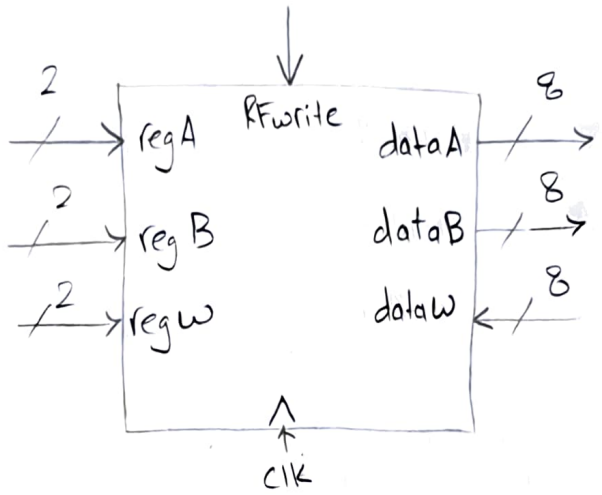
Decoder Ckt

- here is a 2-to-4 decoder



- Let's build more detail for the elements
- begin with registers.

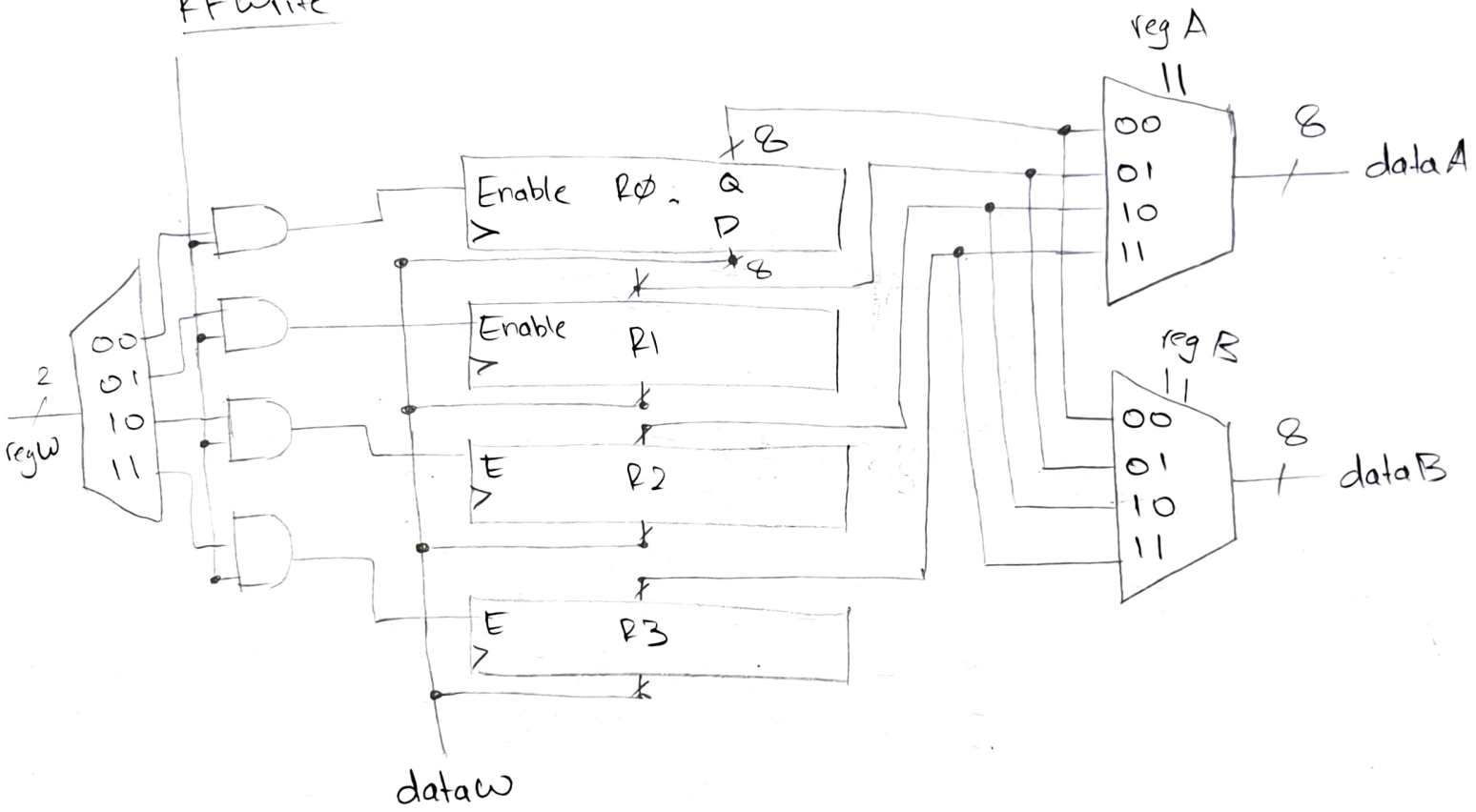
1) Registers → Register File (RF)



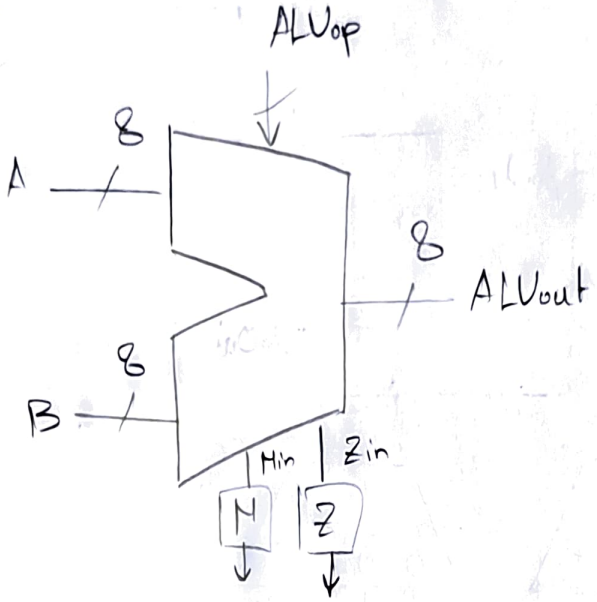
- regA tells register file which register's data to "put" on dataA
- regB " " " " " " " " on dataB
- regW " " " " register to "change" based on dataW only when RFwrite = 1
- be careful not to change registers accidentally

Building CKt

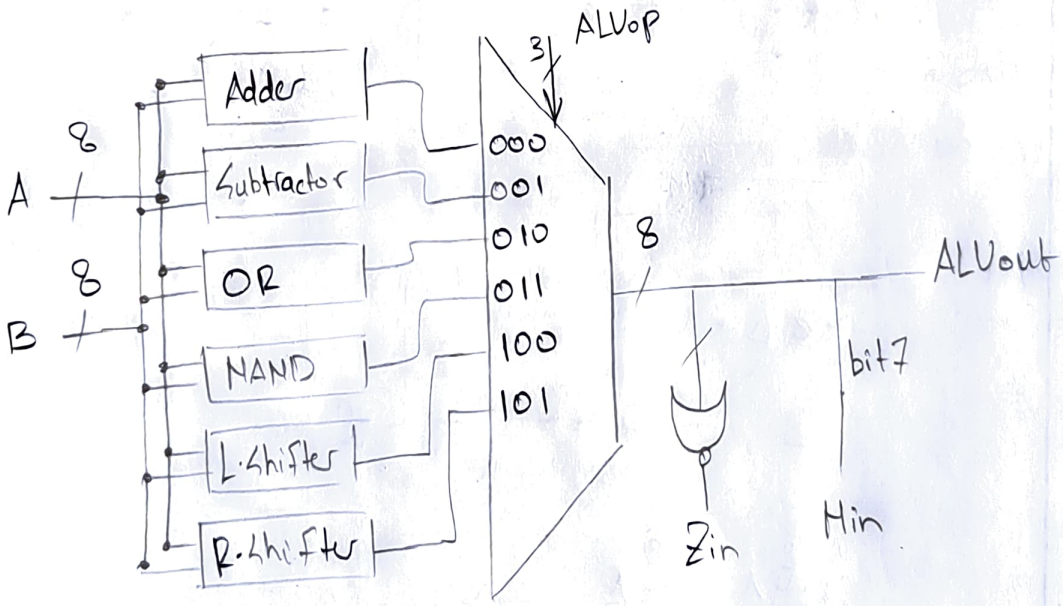
RF Write


reg A
2
00
01
10
11
8
data A
reg B
data B
Enable
R0
Q
D
R1
R2
R3
E
2
regW
dataW

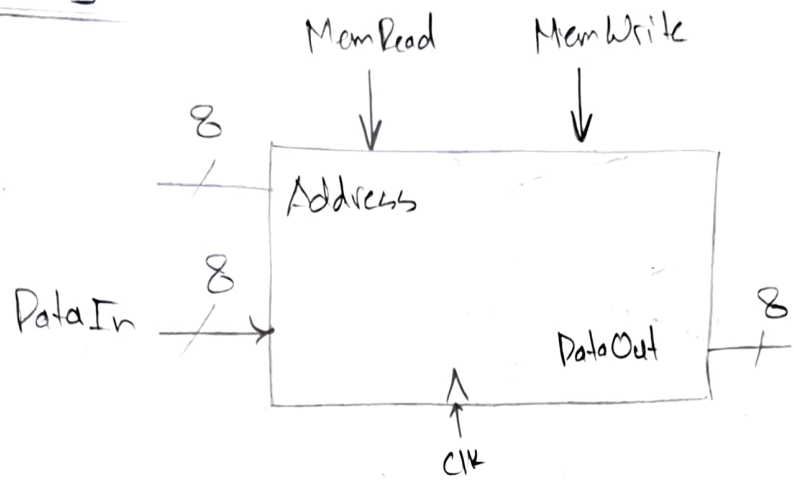
2) ALU : one view



ALUop	ALUout
000	A + B
001	A - B
010	A or B
011	A nand B
100	A << B
101	A >> B

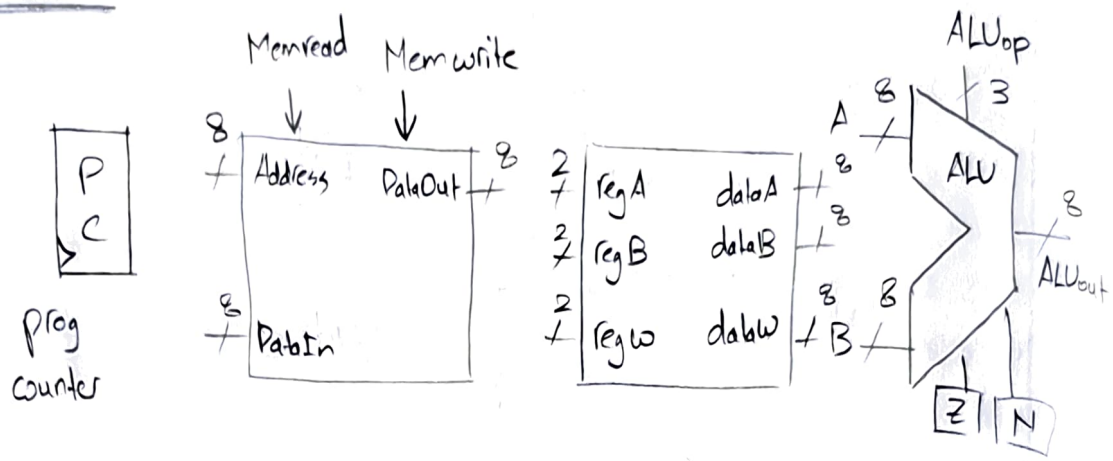


3) Memory



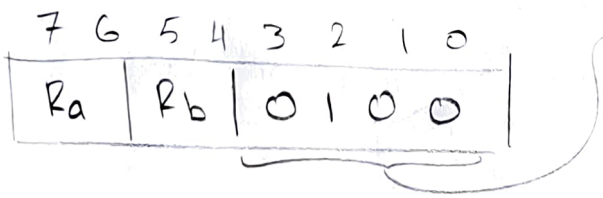
- if MemRead = 1 : • set address
  • on next cycle DataOut gives contents @ that address
- if MemWrite = 1 : • set address, and by end of a cycle
  • contents @ address in memory changes to DataIn

Data Path



Recall add instruction

- add R_a, R_b ⇒ $R_a \leftarrow R_a + R_b$



- here is what happens to do the full execution of this instruction

1) fetch the instruction (encoding) from memory
   - put this into the Instruction Register (IR)
2) Decode (new word): figure out what the instruction is supposed to do.
3) Read R_a and R_b (whichever they are) from the register file → encoded in the instruction

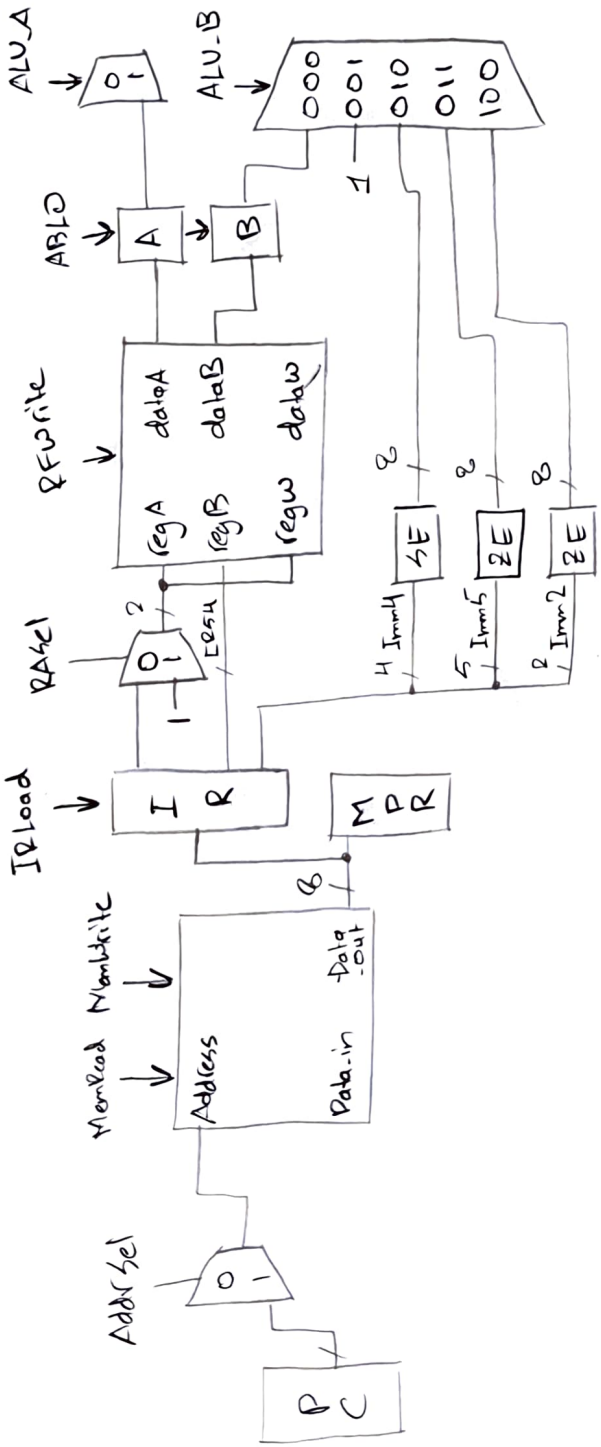
4) Compute ALUout ← Ra + Rb (ALUout is a register)

5) Write ALUout back into Ra

6) Set Z and N based on ALUout

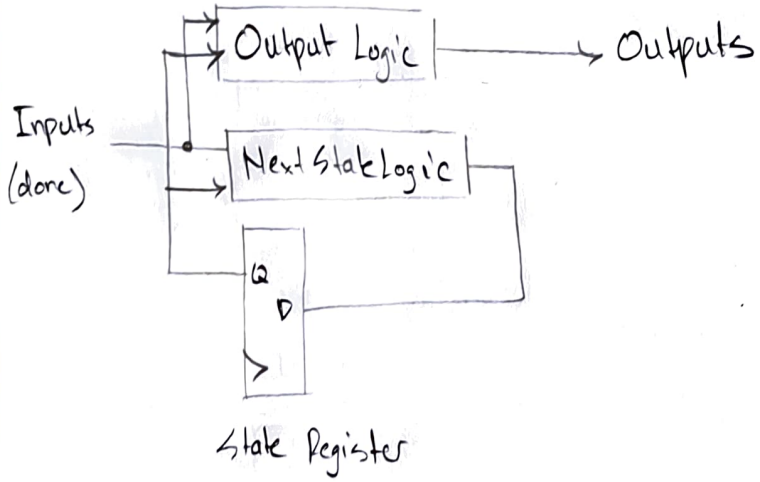
7) Compute PC ← PC + 1

incomplete processor
(didn't get time
to fully copy
from board)



Finite State Machine Control of Simple Processor

- The states of the state machine are just called:
  → Cycle 1, Cycle 2, Cycle 3, Cycle 4, Cycle 5
- This means the next state logic is simple: just proceeds 1 → 2 → 3 → ... until Done
- General FSM



Encoding of add

add Ra, Rb

7 6	5 4	3 2 1 0
Ra	Rb	0100

- Outputs of FSM:
  - every enable on a register (eg. PCwrite)
  - every mux select
  - every memory control + ALUop
- Inputs to FSM:
  - instruction opcode → IR 3,2,1,0
  - the condition codes Z, N (to make branch decision)

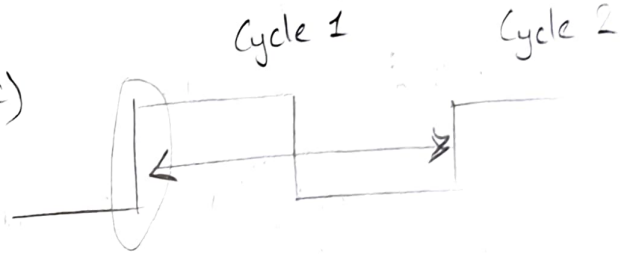
* only show enables that we care about
* In foregoing : only state value of outputs that "are activated"

FSM Control of add, sub, nand

* During cycle 1: fetch IR ← Mem(PC) and PC ← PC + 1

- AddrSel = 1
- MemRead = 1
- IRLoad = 1

} IR ← Mem(PC)



- ALU_A = 0
- ALU_B = 001
- ALU op = 000
- PCWrite = 1

} PC ← PC + 1

* During cycle 2: Decode – there is nothing to do (but we can do the Ra Rb read from RF → A, B because most instructions do this and it will save a whole cycle of time for them)

- RASel = 0
- ABLD = 1

} selects IR 7-6 → reg A

↖ ABLD

* During Cycle 3: (assume Add, Sub, Nand)

- ALU_A = 1 } selects A
- ALU_B = 000 } selects B
- ALUop = Add or Sub or Nand } depending on instruction
- ALUout_LD = 1 } captures ALUoutput
- FlagWrite = 1 } sets Z and N

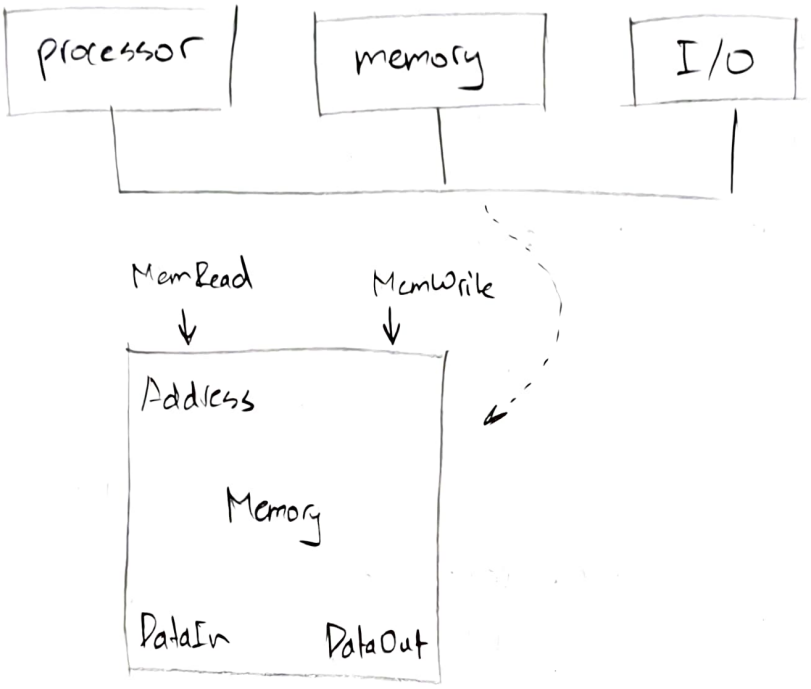
* During Cycle 4: (for Add/Sub/Nand)

- RegIn = 0 } connects ALUout to dataw
- RFWrite = 1 } change/write into RF
- done = 1

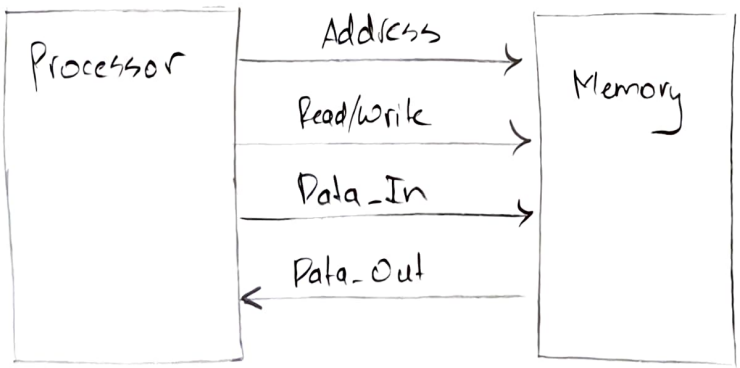
Week 11 : Lecture 1 Mar 24, 2025

Today: Intro to Cache Memory

* memory so far: lw t1, (t0) t1 ← (t0)
  sw t2, (t3) (t3) ← t2



* Stepping back, the processor connection to memory is this:



- processor makes 2 kinds of requests to memory

① Read — load instruction or instruction fetch

- Processor makes 2 kinds of requests to memory
  1. Read — load instruction or instruction fetch
  2. Write — store

  → both called a "memory access"
- we cannot tell from the memory side if a read is a load or an instruction fetch.
- now, some Physics

→ Larger memories (with more bits) are slower ↔ smaller memories are faster (but big memories needed)

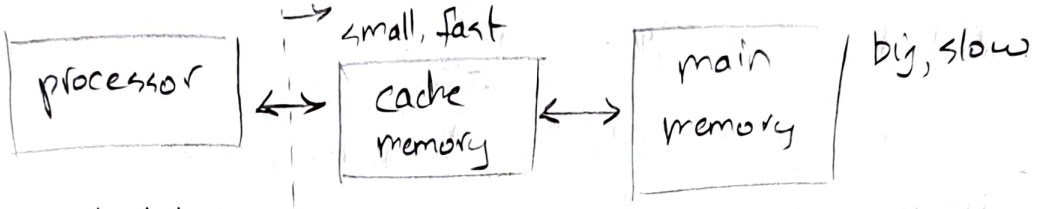
→ why? ∴ longer wires ≡ more resistance, capacitance
  more R×C → more delay
  • also limited by speed of light
  • decoder time for address will get slower

Processors

- 2-4 GHz clock → 0.5 ns – 0.25 ns
- 16 GB memory system takes ≈ 300 cycles to respond to a memory access request

How to solve of fast processor but slow big memory

* put a small fast memory in between the processor and main memory and keep the (hopefully, much fewer) active memory locations in the small fast memory
* which is called a Cache Memory



* need digital "smarts" to manage what is in the small memory
* works well if memory accesses have "locality of reference" (use the same memory locations multiple times before moving on) → both instructions and data.

eg: consider instructions in loop:

```
for (i=1; i<=100; i++){
    ≈≈≈
}
```

} these instructions will be executed 100 times

2 kinds

* spatial (addresses are near to each other)
* temporal (addresses are the same over a time period)

Operation of a Memory Read using Cache

1) Processor requests to read data from address A

2) Cache (somehow) looks inside to see if it has a copy of the data in main memory address A

   ↳ if it doesn't: called <u>cache miss</u>

      ↳ cache requests data @ A from memory

      ↳ also requests A+1, A+2, A+3 ... (block size)

      ↳ memory responds slowly and when A's data arrives, cache grabs it and sends on to the processor

   ↳ if it did contain the data @ A ⇒ called <u>cache hit</u>

      ↳ can quickly send it to the processor in just a few cycles → good ☺

3) If processor then requests A+1, it will come quickly after A because of fetching more than A from memory in a <u>block</u>

• hit ratio

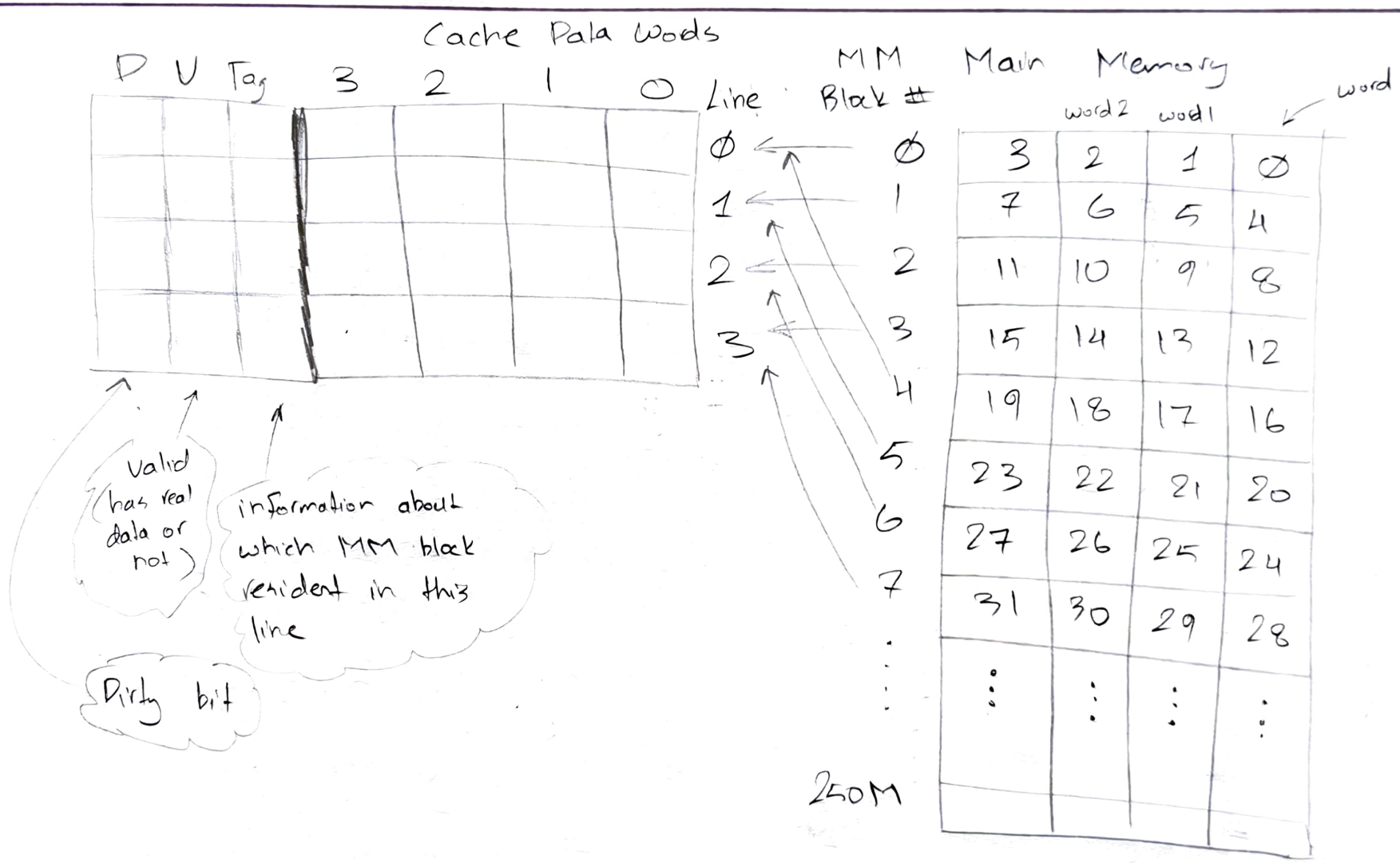
- when a request from the processor (of an address) is in the cache, called a hit

Bookkeeping and Logistics of Caching

- first - because the cache asks for more than one memory location at a time, we will think of main memory as organized into blocks of (eg) 4 words (can be any power of 2)
- aside: for this material, assume every word has an address (v.s. MIPS V - byte addressable)

The Direct-Mapped Cache

- cache has rows and columns like MM.
- each row is called a cache line
- each line size = MM block size because we will copy MM blocks into the cache lines
- consider a (very small) cache with 4 lines and 4 words per line


Cache Data Words
D
V
Tag
3
2
1
0
Line
MM Block #
0
1
2
3
0
1
2
3
4
5
6
7
250M
Main Memory
word2
word1
word
3 2 1 0
7 6 5 4
11 10 9 8
15 14 13 12
19 18 17 16
23 22 21 20
27 26 25 24
31 30 29 28
Valid (has real data or not)
information about which MM block resident in this line
Dirty bit

Direct-Mapped Cache Cont'd

- now consider when an address (say 9) is requested by the processor → MM block 2
  - ↳ where can MM blocks go in cache? (just 1 line? all lines? some lines?)
- 1 line → "Direct Mapped Cache"
- good → only have to look at one tag/line to see if ∃ a hit or miss
- bad → 2 or more MM blocks that use the same line will conflict and evict each other from the cache
- which line does MM block j "map" into? ($0 \leq j \leq 250$M)
  - ↳ $j \bmod 4$ (ie: $j \bmod$ # lines in cache)
    - ↳ remainder after division by 4
- how to detect hit/miss?
  - ↳ compare tag of cache line (if valid) with tag of the requested address's block
- example: assume 32-bit address

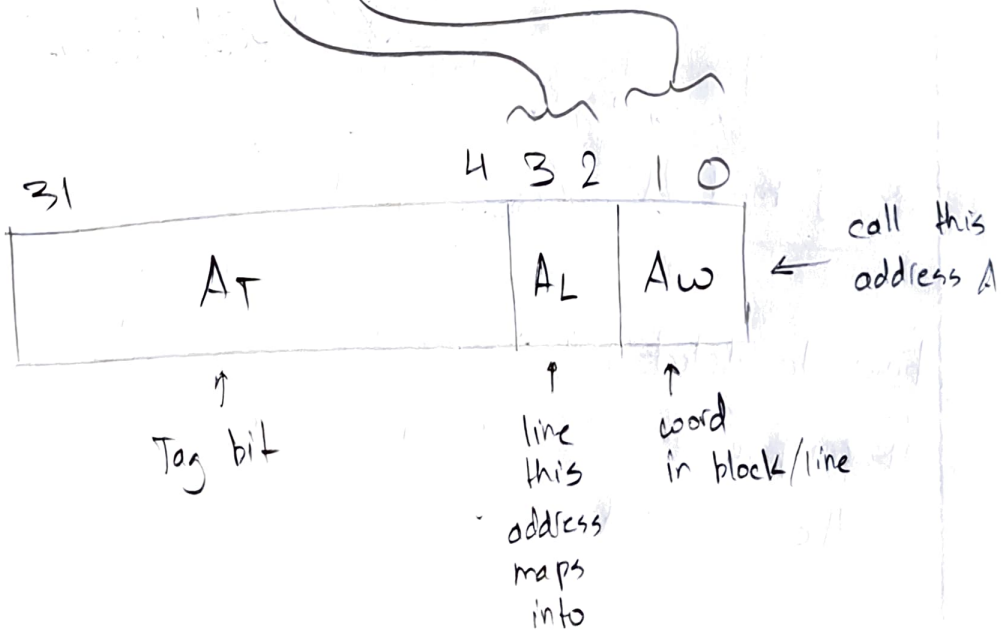
MM block j → cache line $j \bmod 4$

(4 = # of lines in cache)

* Consider these possible mem. access Request addresses of the first 9 blocks in MM

Block #	Bits of Addr. 31 30 ... 5 4 3 2 1 0
0	0 0 ... 0 0 0 0 x x
1	0 0 ... 0 0 0 1 x x
2	0 0 ... 0 0 1 0 x x
3	0 0 ... 0 0 1 1 x x
4	0 0 ... 0 1 0 0 x x
5	0 0 ... 0 1 0 1 x x
6	0 0 ... 0 1 1 0 x x
7	0 0 ... 0 1 1 1 x x
8	0 0 ... 1 0 0 0 x x

00, 01, 10, 11 } 4 words in each block



Week 11: Lecture 3 Mar 27, 2025

Direct-Mapped Cache

- Tag tells us which MM block is in a cache line
- Valid bit tells us if a cache is actually useful
- D - Dirty bit
- A_L: the line in the cache that this word / block would reside in if it's in cache
- Tag: that would be in cache Tag field - Tag(A_L) if this block was in the cache
- Consider what happens when the processor makes a read request to the cache at memory address A
  - ↳ cache looks inside to see if Tag(A_L) = A_T
    - ↳ if Tag(A_L) = A_T AND $V(A_L) = 1$ (valid), then → hit (cache sends that word quickly to the processor)
    - ↳ otherwise, → miss
      - ↳ if the line in A_L is valid, it must be evicted because direct-mapped cache only lets a MM block be in one place in cache
      - ↳ the new block is retrieved (slowly) from MM and copied into cache line A_L
      - ↳ the Tag(A_L) is set to become A_T
      - ↳ set valid bit = 1 — $V(A_L) = 1$
      - ↳ set dirty bit = 0 — $D(A_L) = 0$ (not yet dirty)

Eviction of Cache Line

→ **Method 1: Write-Back Cache**

- what we've been talking about
- if there was a write/store to any word in the line, its dirty bit will be set $D(A_L) = 1$
- when $D(A_L) = 1$, upon eviction, the line must be "written back" to MM so that MM is correct.
- multiple writes to a line only results in 1 write to MM upon eviction → goal ☺

→ **Method 2: Write-Through Cache** (aside)

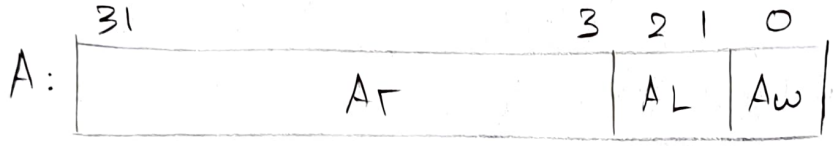
- does not have a dirty bit
- all writes to cache are also sent to MM
- mostly bad because you do more traffic to MM than needed → slow

Example Sequence of Reads for Slightly Diff. Cache

- 4 lines → 2 words/line
- 32 bit word addressable
- cache contains, already

V	Tag	word 1	word 0	line #
1	00 00	0xCA	0xFE	0
1	00 01	0x0E	0xAD	1
1	00 00	0xBE	0xEF	2
0	00 ... 01	0xFE	0xED	3

- processor read address request



different because 2 words/lines, words/block

- Binary read address

1) AT: 0.....01 (2), AL: 01 (1), Aw: 0 (3) → tag match in line $A_L = 01$,
$V(01) = 1$ → hit → get 0xAD (word 0 in line 01)

2) AT: 0....01, AL: 11, Aw: 0 → tag match in line $A_L = 11$,
but $V(11) = 0$ (not valid) → miss

3) AT: 0....00, AL: 00, Aw: 1 → tag match in line $A_L = 00$,
$V(00) = 1$ → hit → get 0xCA

4) AT: 0.... 11, AL: 01, Aw: 1 → tag mismatch → miss

Week 12: Lecture 1 Mar 31, 2025

Recall: Direct Mapped Cache

- each MM block can only reside in 1 cache line
- Good: fast to determine if specific block in cache
- Bad: if blocks want some line at same time → conflict
  - ↳ will evict each other

New Way: Fully Associative Cache

- every MM block can map into every cache line
- good: solves the conflict problem in the best way possible
- Bad: need a lot of digital circuits (comparators) to simultaneously check all tags of all lines

Compromise Between These Two

- each main memory block can reside in a small set of cache lines
  - ↳ say "M" cache lines
  - ↳ called an M-way Set-Associative Cache
  - ↳ typical values of M are 2-16 (there are typically 1000's of cache lines)

Ex 1: 2-Way Set-Associative Cache

* allows a main memory to reside in 2 different cache lines → those 2 lines are called a set

Ex 2: NIOS-V Cache

* more realistic
* for a NIOS-V processor — with byte addressibility
* cache size is 32 Kbytes → 32,768 bytes (power of 2)
* cache has 32 bytes per line = 8 words per line
* it is an M = 4-way set-associative cache
* To determine the set associative cache mapping / hit or miss, need to compute:

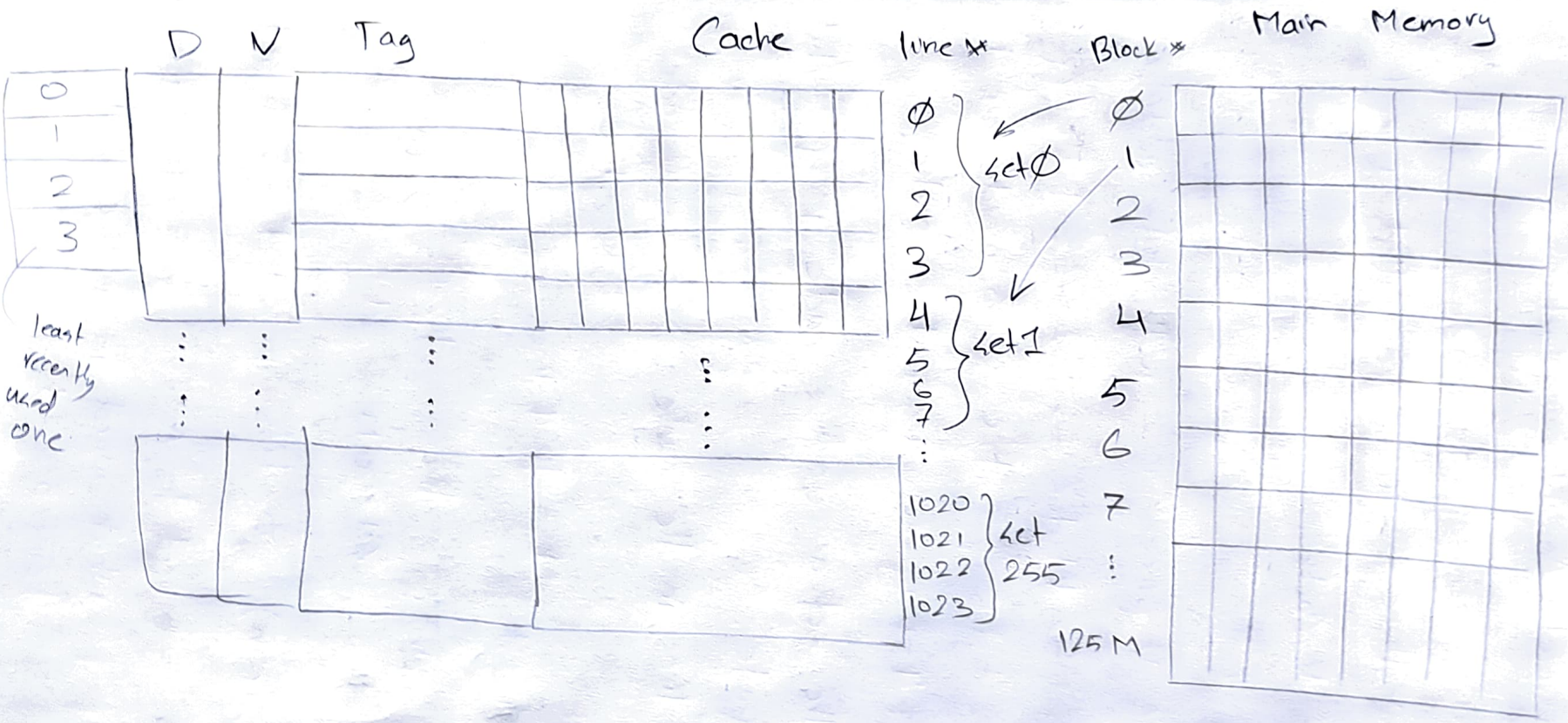
↳ Number of lines in the cache: $\frac{\text{\# bytes in cache}}{\text{\# bytes / line}}$

$$\Rightarrow \frac{32 \text{ K bytes}}{32 \text{ bytes/line}} = 1\text{K} = 1024 \text{ lines}$$

↳ Number of sets: $\frac{\text{\# lines}}{\text{set size M}} = \frac{1024}{4} = 256$

↳ need $\log_2(256) = 8$ bits to select which set

↳ 8 words/line ⇒ need $\log_2(8) = 3$ bits to determine which word in a line/block is being accessed.

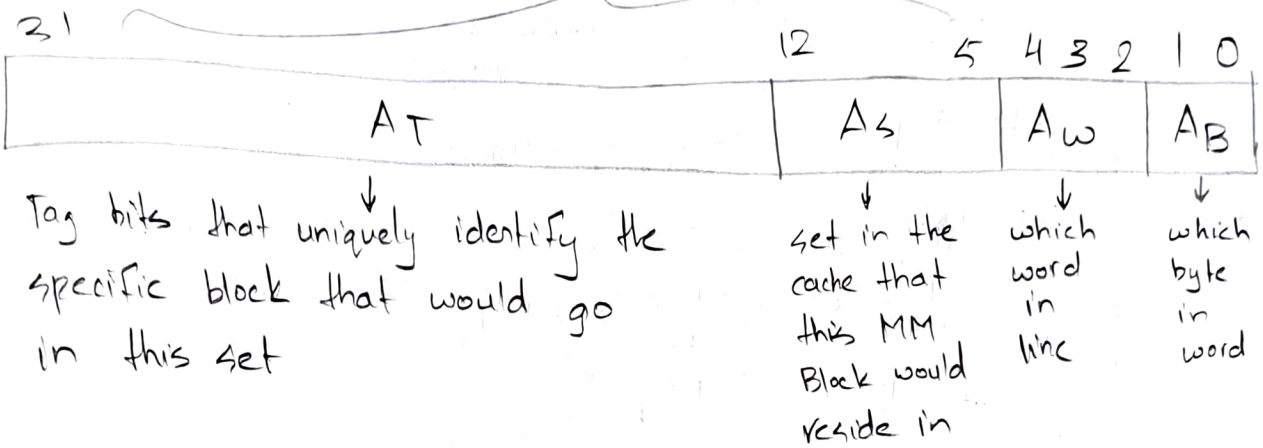


* block 0 → line 0 → block j → j mod 256
  1 → " 1

* random is almost as good

Ex 2 Cont'd

* this is in context of the processor requesting to access a memory address A
* a 32-bit address



* MM block j can reside in any line in set j mod 256
* So, an access of address A to MM works like this:
  - ↳ the cache looks at all 4 tags in set (A_S) to see if any match A_T
    - ↳ if match & V=1 for that line (ie valid), then => hit
    - ↳ else its a miss and must go get the block from MM
* when a new block is brought in → put it in any line that has V=0 (not valid)
  - ↳ if all valid? → least recently used line (this is one of many methods)
    - ↳ keep a 2-bit counter for each line { each line accessed has its counter set to 0; other lines are incremented; if set is full, the lines will have all counters different; evict line with counter = 3 if it is the least recently used one)

Analyzing a Program's Cache Behaviour

* Consider some contrived code example w.r.t their cache behaviour

Example 1: Conflicting accesses

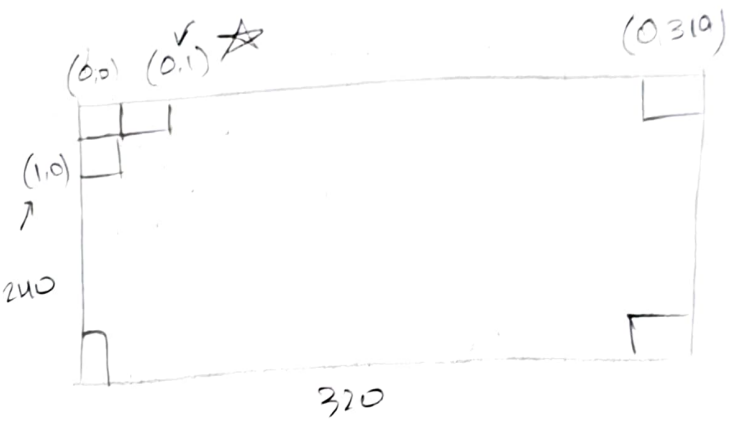
```
int A[1000], B[1000], sum;
sum = 0;
for (int i=0; i<1000; i++){
    sum += A[i] + B[i];
}
```

* assume a direct-mapped cache with block/line size = 8
* suppose we're unlucky: every element A[i] maps to the same cache line as B[i].
* what goes wrong?
  - ↳ accessing B[i] after accessing A[i] will kick 8 integers A[i] → A[i+7] out of cache
  - ↳ will happen every iteration, destroying the cache's performance
* you can rewrite code to prevent the problem
  - ↳ eg: sum up the A[i] separately from B[i]
    (the

Example 2: Accessing 2-Dimensional Arrays (eg. frame buffer)

short buf[240][320];

(0,0) is @ address 1000
(0,1) " " " 1002
(0,2) " " " 1004
(0,319) " " " 1638




```
Void bad_clear(){
  for(int c=0; c<320; c++){
    for(int r=0; r<240; r++){
      buf[r][c] = 0x0;
    }
  }
}
```

- bad because the sequence of addresses is not in sequential order
- lots more misses
- can rewrite to make better
  - ↳ reverse the loop order (next page)

```
void cache_friendly_clear(){
    for (int r=0; r<240; r++){
        for (int c=0; c<320; c++){
            buf[r][c] = 0x0;
        }
    }
}
```

+ does access memory sequentially
  ↳ much better performance

Week 12 : Lecture 3

Performance Analysis of Processors with Caches (running a program)

- processor emmits a request to access memory
- if a memory address hits in the cache, the cash can respond quickly
- let's say "quickly" means that the cache takes c cycles to respond to a hit (processor cycles)
- $c \approx 4$ in a modern processor (of modern processors running at 2→5 GHz)
- if the memory request is a miss, let's say takes m cycles to get the block from main memory and the word requested to the processor.
- $m \approx 300$ cycles on a modern processor
- m is called a "miss penalty"
- lots of misses is bad

Define : hit rate, $h = \dfrac{\#\text{ accesses that hit in cache}}{\text{total }\#\text{ of accesses}}$

$0 \leq h < 1$

- using h, c, m, we can compute the average number of cycles it takes to fulfill a memory request.

- How many cycles would 1000 accesses take?

$$\frac{1000 \times h \times C + 1000 \times (1-h) \times m}{1000}$$

(m: miss rate)

- Average Access Cycles (AAC) $\Rightarrow$ $AAC = h \times C + (1-h)m$

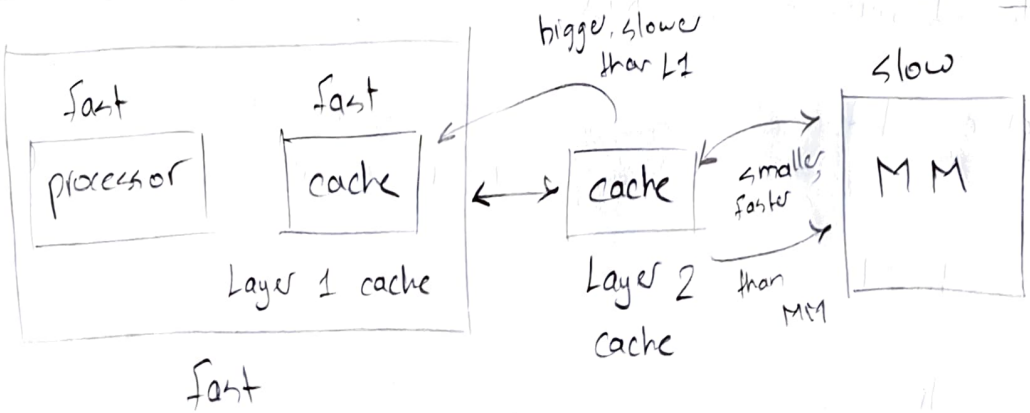
- guess $h = 0.9$ (90% hit rate) $C = 4$, $m = 300$

$AAC = 0.9 \times 4 + 0.1 \times 300 = 33.6$ cycles $\gg 4$

$\Rightarrow$ ∴ slow!

h	AAC
0.85	48
0.90	34
0.95	19
0.98	9.9
0.99	7.0

3% difference in miss penalty (hit ratio?) is 2x performance



- the layer 2 cache will have * cycles to access
  faster than MM, slower than cache
- bigger, better hit rate → speed up things overall
- "effective" miss penalty seen by L1 cache will be lower
- can have L3 as well, L4 not common
- because instruction memory accesses have different "pattern" than data accesses, the L1 cache is often split into
  - L1 instruction cache
  - L2 data cache

  } architect can design different caches for each (diff size, associativity, etc...)

Final Exam Practice

Allan Compiler Questions

1)
```
.global add
add:  add a0, a0, a1
      ret
```

2)
```
.global sum_to_ten
sum_to_ten: la t0, sum
            lw t1, (t0)  * t1 <- sum
            li t2, 1     * int i = 1
my_loop:    add t1, t1, t2  * sum = sum + i
            addi t2, t2, 1
            ble t2, t3, my_loop
            mv a0, t1
            ret
```

at bottom
```
sum: .word 0
```
```
li t3, 10
```

3)
```
.global check_number
check_number: bgz a0, ret_pos
ret_pos:  la t0, pos
          mv a0, (t0)
          ret

          blz a0, ret_neg:
          la t0, zer
          mv a0, (t0)
          ret
ret_neg:  la t0, neg
          mv a0, (t0)
          ret
```

at bottom
```
.data
pos: .byte 70
neg: .byte 6e
zer: .byte 7a
```
0x

4)
```
.global main
main: li a0, 3
      li a1, 4
      call multiply
      ret
multiply: li t0, 0
myloop: add t0, t0, a0
        addi a1, a1, -1
        bnez a1, myloop
        mv a0, t0
        ret
```

* only works if int a and b are (+)ve integers

Simple Processor

10 Instructions

- add Ra, Rb
- sub Ra, Rb
- nand Ra, Rb
- ori imm5
- load Ra, (Rb)
- store Ra, (Rb)
- shift R/L Ra imm2
- bnz imm4
- bpz imm4
- bz imm4

Cycle-by-Cycle Control and Datapath

add / sub / nand

Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
AddrSel = 1	RASel = 0	ALU-A = 1	RegIn = 0	
MemRead = 1	ABLD = 1	ALU-B = 000	RASel = 0	
IRload = 1		ALUop = 000	RFWrite = 1	
...		ALUoutLD = 1		
ALU-A = 0		FlagWrite = 1		
ALU-B = 001				
ALUop = 000				

PCWrite = 1

the above is for add

for sub and nand, exactly the same except

ALUop changes value for sub or nand in cycle 3 //

ori imm5

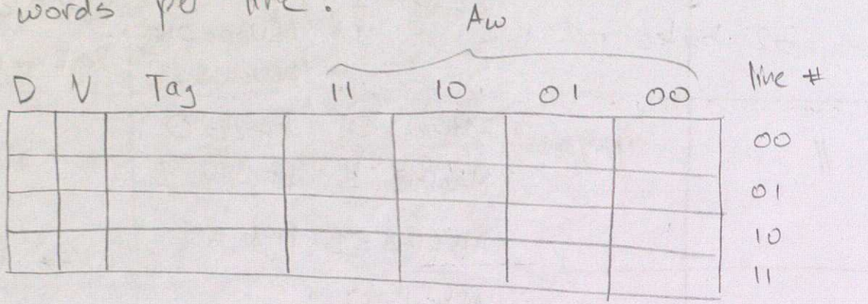
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
	//	//	RASel = 1 ABLD = 1	ALU-A = 1 ALU-B = 011 ALUop = 010 ALUoutLD = 1 **	RASel = 1 RFWrite = 1 RegIn = 0
load	//	//	AddrSel = 0 MemRead = 1 MDRload = 1	RASel = 0 RFWrite = 1 RegIn = 1	
store	//	//	AddrSel = 0 MemWrite = 1		
shift	//	//	ALU-A = 1 ALU-B = 100 ALUop = 100 * ALUoutLD = 1	RASel = 0 RFWrite = 1 RegIn = 0	
bnz/ bpz/ bz	//	//	ALU-A = 0 ALU-B = 010 ALU-op = 000 PCWrite = $\overline{Z}$ ← bnz = $\overline{N}$ ← bpz = Z ← bz		

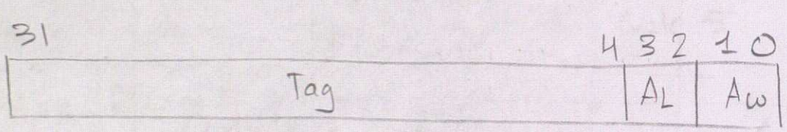
* depends on L or R bit

** FlagWrite = 1

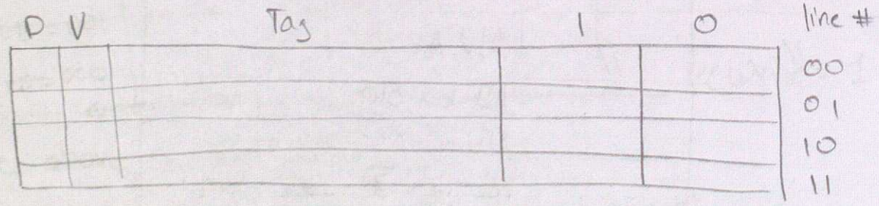
Direct Mapped Cache

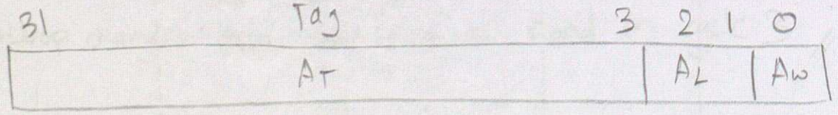
Consider a computer system that is word-addressable and has a direct-mapped cache with 4 lines and 4 words per line:





Now consider a 32-bit word-addressible system with a 4-line cache that has 2 words per line





M-way set-Associative Cache

Consider a practical, 4-way set associative cache implemented for the HIOS V computer system, a 32-bit byte addressable system. The 32 kbyte cache has 32 bytes per line. To design the cache:

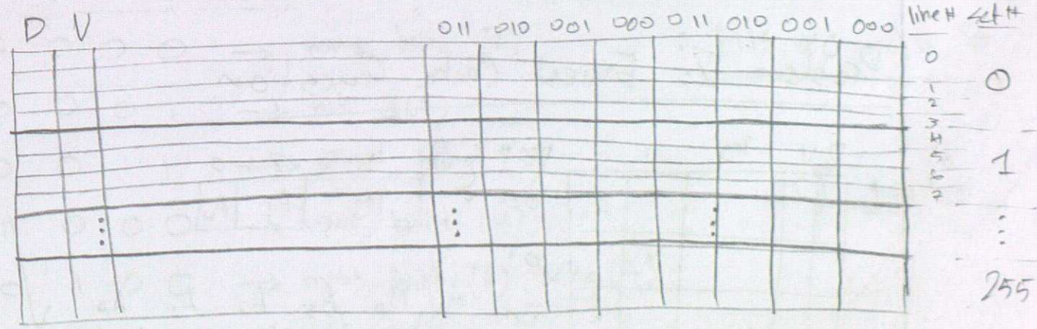
* words per line: $\frac{32 \text{ bytes} \times \frac{8 \text{ bits}}{\text{byte}}}{32 \text{ bits/word}} = \boxed{8 \text{ words/line}}$ ②

* lines in cache: $\frac{32 \text{ kbytes}}{32 \text{ bytes/line}} = 1\text{k lines} = 1024 \text{ lines}$

* sets in cache: $\frac{1024 \text{ lines}}{4 \text{ lines/set}} = \boxed{256 \text{ sets in cache}}$ ③

* already known that ∃ $\frac{32 \text{ bits/word}}{8 \text{ bits/byte}} = \boxed{4 \text{ bytes/word}}$ ①

① : $\log_2(4) = 2$ ② $\log_2(8) = 3$ ③ $\log_2(256) = 8$



Prof. Rose Final Lecture Questions

Problem 1: Processor

a) MAC RA Imm2 ⋇ $RA = RA + R1 \times 2^{\wedge Imm2}$

① RA = RA + (R1 << Imm2)
② PC = PC + 1

7 6	5 4	3 0
RA	Imm2	1110

Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
AddrSel = 1	RASel = 0	RASel = 1	ALU_A = 1	ABLD = 1	RASel = 0
MemRead = 1	ABLD = 1	ABLD = 1	ALU_B = 100	ALU_A = 1	RFWrite = 1
IRLoad = 1			ALUop = SH.LF.	ALU_B = 101#	Regin = 0
ALU_A = 0			ALUoutLD = 1	ALUop = ADD	
ALU_B = 001			RASel = 0	ALUoutLD = 1	
ALUop = ADD					
PCwrite = 1					

b) we need wire going from ALUout to ALU-B mux and call those select bits 101 # (new #)

Problem 2: Easiest Cache Question

a)

D	V	Tag	1	0	line #
				4	00
			3		01
				5	10
					11

4 3	2 1	0
Tag	A_L	A_W

∴ A_4 A_3 | A_2 A_1 | A_0 ✓
Tag | line | word
Tag | A_L | A_W

b) 10 00 0 → miss bring (1) into cache, 1
10 00 1 → hit ↑(1) 2
10 01 1 → miss " (3) " " " 3
11 00 0 → miss → evict and bring (4) 4
00 10 0 → miss bring (5) " " 5
00 10 1 → hit ↑(5) 11 01 0 / 10 01 0 6
∴ hits are 2 and 6 ✓

c) 10 01 0 → this will hit since ref # 3 brought that block ∴ 4 → 10010 ✓

d)

P	V	Tag	1	0	line #	set #
				5 / 1	0	0
				4	1	
			3		2	1
				7	3	

A_4 A_3 | A_2 | A_1 | A_0
Tag | set | word ✓

e)

1	10	0	0	0	→ miss, bring (1)
2	10	0	0	1	→ hit ↑(1)
3	10	0	1	1	→ miss, bring (3)
4	11	0	0	0	→ miss, bring (4)
5	00	1	0	0	→ miss, evict (1), bring (5)
6	00	1	0	1	→ hit ↑(5)
7	11	0	1	0	→ miss, bring (7)
8	10	0	1	0	→ hit ↑(3)

∴ hits are 2, 6, 8 ✓

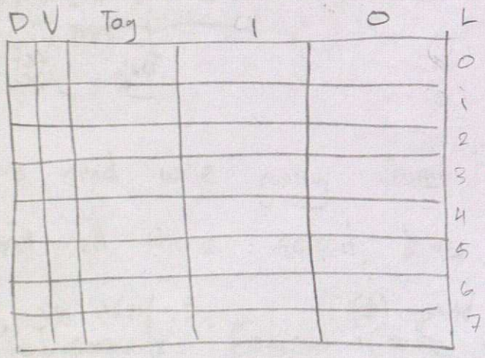
f) 11110 ✓
set 1, word 0 ... empty, so miss

3. Hardest Cache Question

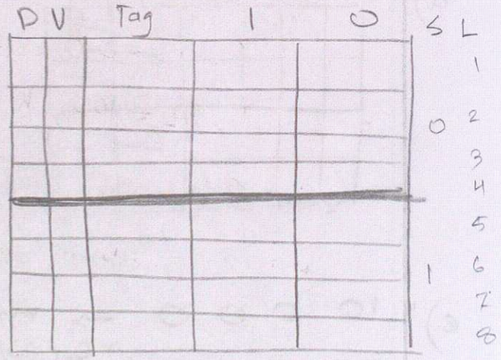
- address bus 16 bits
- word size 8 bits (1 byte)
- cache size 16 bytes (16 words)
- words per line is 2 bytes (2 words)
- write-back cache

- 4-way set associative (2 sets total)
- eviction via LRU model

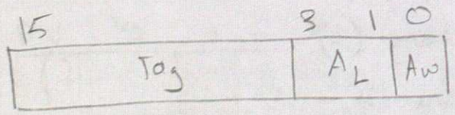
Direct-Mapped



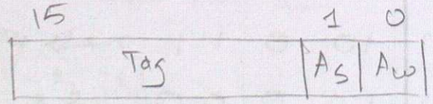
4-way Set-Associative



a)



b)



Assuming word-addressibility

c) i)

1: $A_W = 0$, $A_L = 000$, miss, bring (1)
2: hit ↑(1)
3: $A_W = 1$, $A_L = 011$, miss, bring (3)
4: $A_W = 0$, $A_L = 111$, miss, bring (4)
5: $A_W = 0$, $A_L = 101$, miss, bring (5), edit so D = 1
6: $A_W = 1$, $A_L = 111$, miss, bring (6), evict (4)
7: $A_W = 0$, $A_L = 001$, miss, bring (7), edit so D = 1
8: $A_W = 1$, $A_L = 111$, miss, bring (8), evict (6)
9: $A_W = 0$, $A_L = 110$, miss, bring (9)
10: $A_W = 0$, $A_L = 100$, miss, bring (10), edit so D = 1
11: $A_W = 0$, $A_L = 010$, miss, bring (11)
12: $A_W = 1$, $A_L = 010$, hit ↑ (11)
13: $A_W = 1$, $A_L = 100$, hit ↑ (10)
14: $A_W = 0$, $A_L = 011$, miss, bring (14), evict (3)
15: $A_W = 0$, $A_L = 100$, miss, bring (15), evict (13)

ii)

* hits = 3
* misses = 12

hit rate = $\frac{\text{\# hits}}{\text{total \#}} = \frac{3}{15} = \underline{0.2}$

d) i)

	A0	A5	
1:	0	0	miss , bring (1)
2:	1	0	hit ↑ (1)
3:	1	1	miss, bring (3)
4:	0	1	miss, bring (4)
5:	0	1	miss, bring (5), edit so D=1
6:	1	1	miss, bring (6)
7:	0	1	miss, evict LRU(1) = (3), bring (7), edit so D=1
8:	1	1	hit ↑ (4)
9:	0	0	miss, bring (9)
10:	0	0	miss, bring (10), edit so D=1
11:	0	0	miss, bring (11)
12:	1	0	hit ↑ (11)
13:	1	0	hit ↑ (10)
14:	0	1	miss, evict LRU(1) = (5), bring (14)
15:	0	0	miss, evict LRU(1) = (1), bring (12)

ii) * hits = 4
* misses = 11

hit ratio = $\frac{\text{\# hits}}{\text{total \#}} = \frac{4}{15} = \underline{0.267}$