CSC258 Winter 2016
Lecture 8

Midterm

- Class average 51%, highest mark 43/50
- Class Average including Lab 1-5 and midterm: 73%
- Make sure your midterm mark is correct on MarkUs

- Midterm solution
  - http://www.cs.toronto.edu/~ylzhang/csc258w16/files/midterm-solution.pdf

- Remarking request form
  - http://www.cs.toronto.edu/~ylzhang/csc258w16/files/csc258-midterm-remarking.pdf

Reflections

What're the reasons why I didn't do as well as I expected?

- Did I do well in the questions that are directly related to labs and quizzes?
- Is there a question that I could have answered better if I had a few minutes more time?
- Is there a question that I could not answer even if I had more time?
- Does the midterm test on a reasonable expectation of how much should be learned from this course?
- Provide feedback, using the feedback form, or just talk to me.
- If you got < 40% in the midterm you should arrange a meeting with me to talk about it how to do better in this course.

- Drop date: March 6

Quiz standings

- Threshold for top 20%: 8 points
- Threshold for top 50%: 5 points

Top Quizzers	
Chris C.	12
Elijah M.	11
Michael Z.	10
Frank Y.	10
Jailani D.	10
Farjad A.	9
Nitharsan K.	9
Alex K.	9
Jay G.	9
Eric C.	9


QUIZ TIME!

Question 1:

A word-addressable RAM unit has 10 address bits going into it. How many bytes is the RAM unit able to store?

A word is 4 bytes
Word-addressable means each word has a unique address.

Question 2:

When reading from RAM, what are the values for CE' and OE' ?

CE' = _____ OE' = _____

Question 1:

A word-addressable RAM unit has 10 address bits going into it. How many bytes is the RAM unit able to store?

A word is 4 bytes
Word-addressable means each word has a unique address.

Solution:

Each 4 bytes has a unique address
There are 2^10 = 1K unique addresses
So total size 4 byte x 1K = 4KB

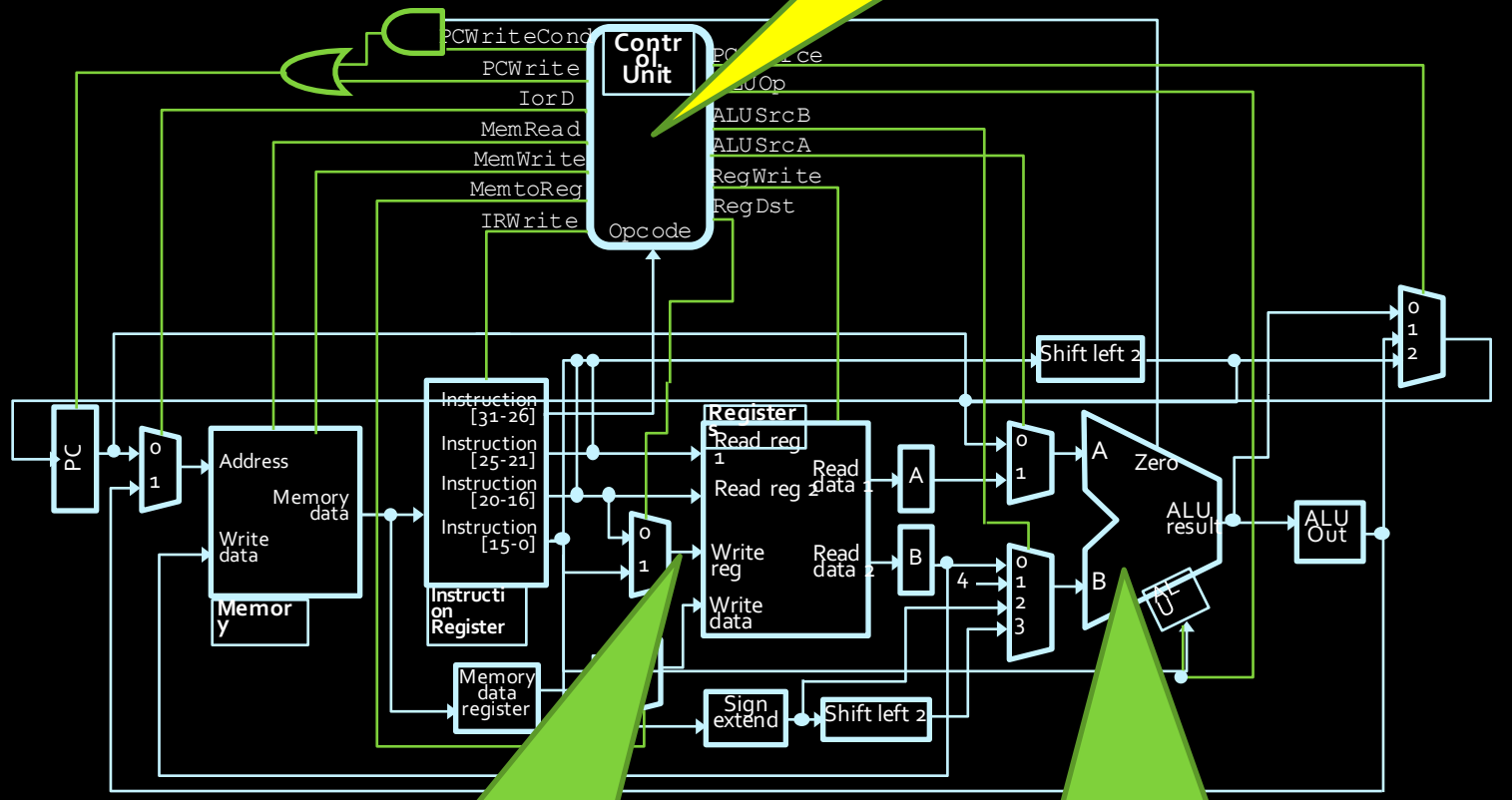
Question 2:

When reading from RAM, what are the values for CE' and OE' ?

CE' = 0

OE' = 0

The Blueprint of a microprocessor



The “Controller Thing”

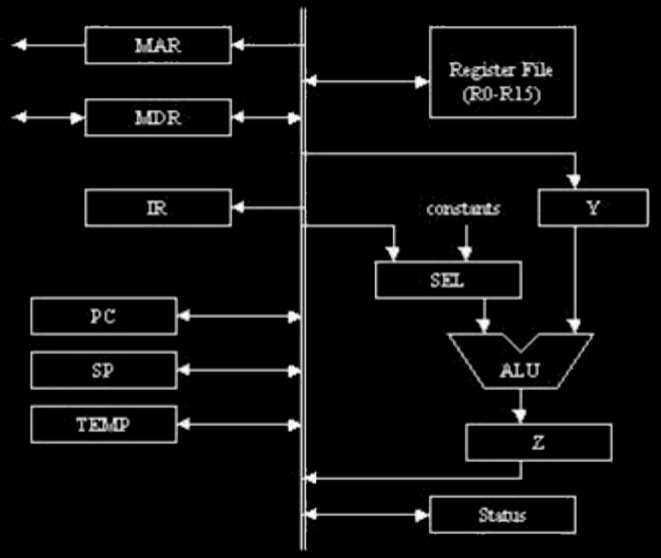
aka: the Control Unit

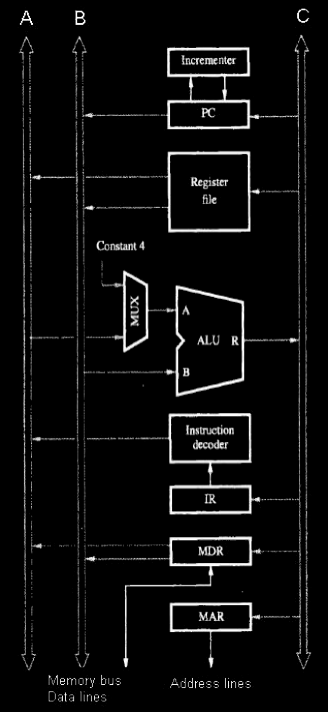
The Control Unit controls how data flows in the datapath.

Different executions have different flows.

Processor Datapath

- The datapath of a processor is a description/ illustration of how the data flows between processor components during the execution of an operation.
- Examples:

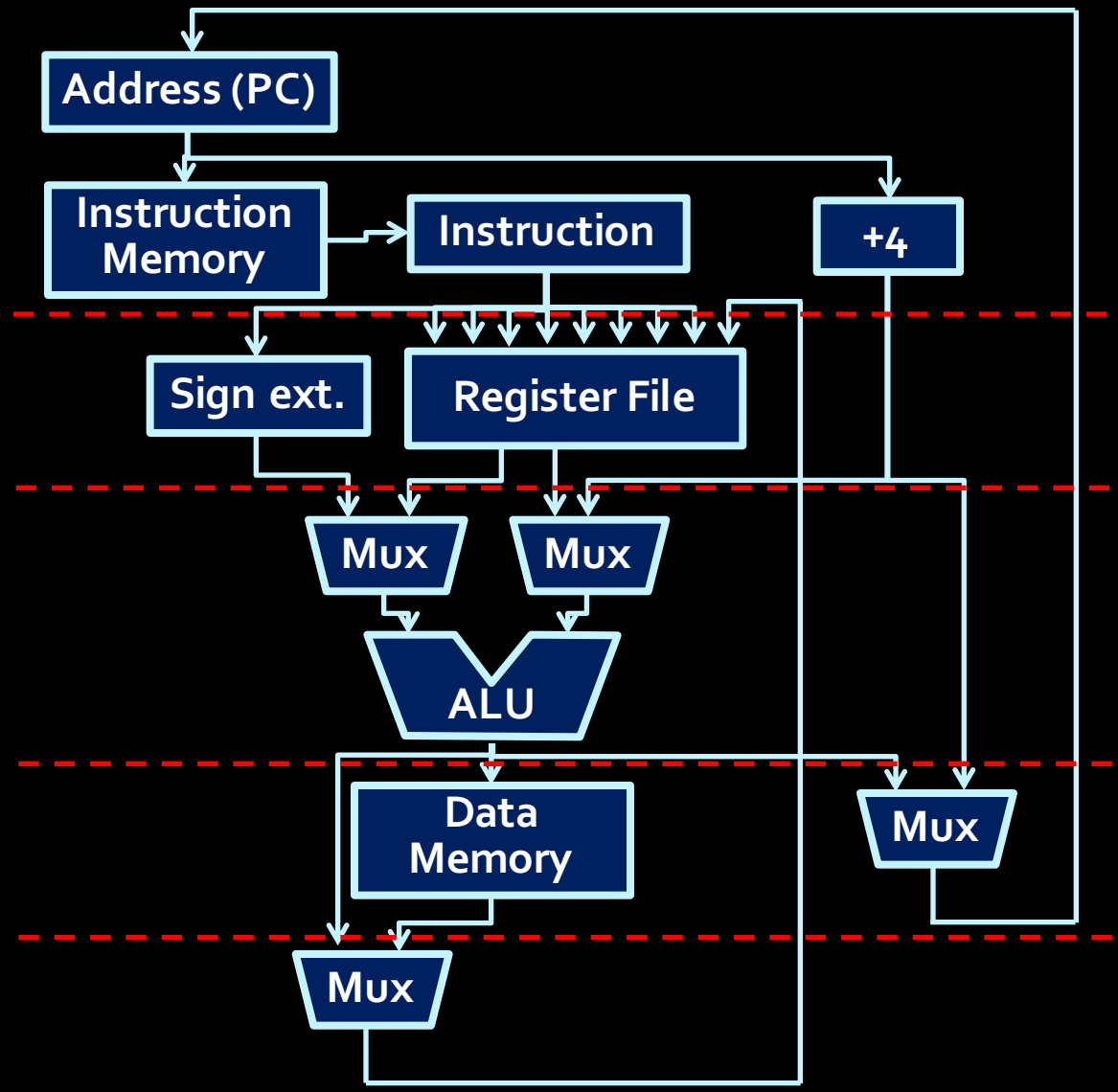




Datapath example

Note: this is just an abstraction ☺

- The simplified datapath for most processor operations has stages as shown in the diagram:
  - Instruction fetch
  - Instruction decode & register fetch
  - Address/data calculation
  - Memory access
  - Write back.



What happens when you run an executable on your computer?

"Quartus.exe", "ls", "minecraft", …

1. The OS loads a bunch of instructions into the memory at certain location.
2. CPU finds that location and executes the instructions stored there one by one.

What does an instruction look like?

```
00000000 00000001 00111000 00100011
```

It's a 32-bit (4-byte) binary string.

How do we remember the location of the current instruction?

- The program counter (PC) stores the location of the current instruction.
  - Each instruction is 4 bytes long, thus we do +4 to increments the current PC location.
  - PC values can also be loaded from the result of an ALU operation (e.g. jumps to a memory address).

So, here is the instruction. Do it!

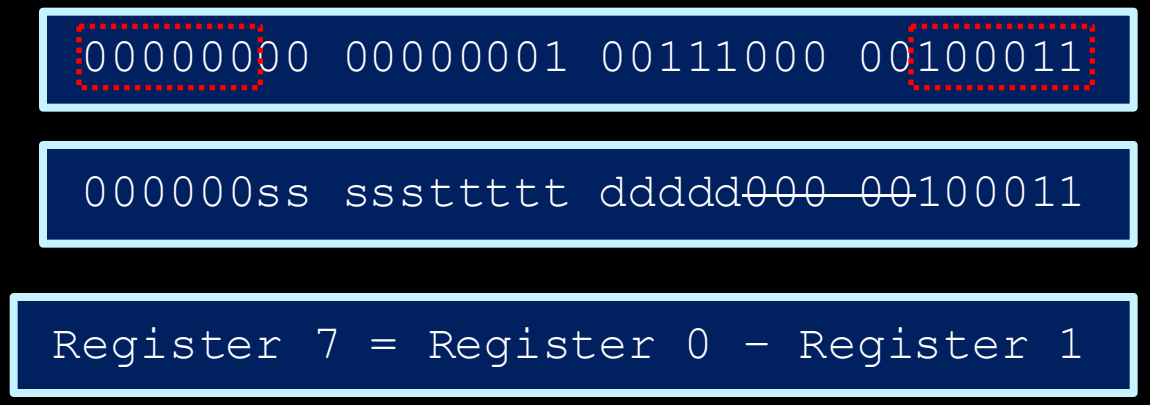
```
00000000 00000001 00111000 00100011
```

What does the instruction mean?
What operations do I do?
Where do I get the inputs and put the output?

We need to **decode** the instruction.

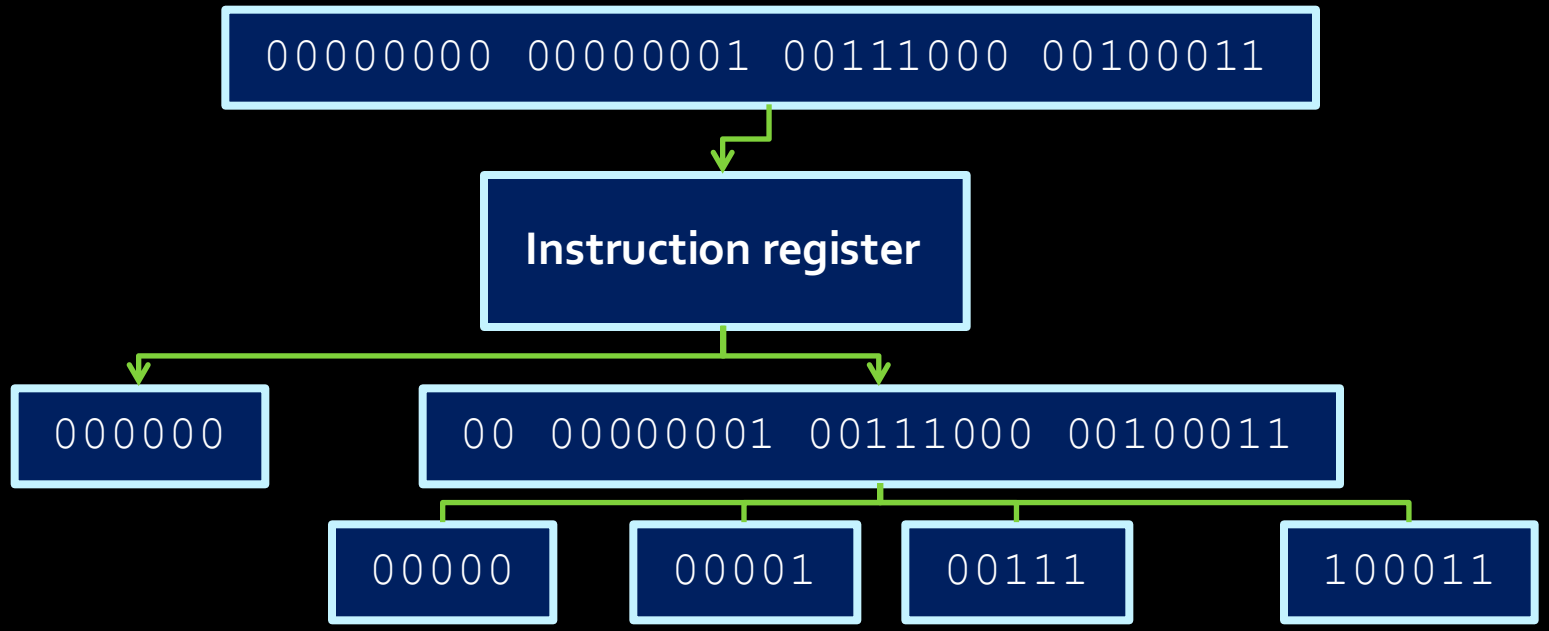
Instruction decoding

- The instructions themselves can be broken down into sections that contain all the information needed to execute the operation.
  - Also known as a control word.
- Example: unsigned subtraction



Instruction registers

- The instruction register takes in the 32-bit instruction fetched from memory, and reads the first 6 bits (known as the opcode) to determine what operation to perform.



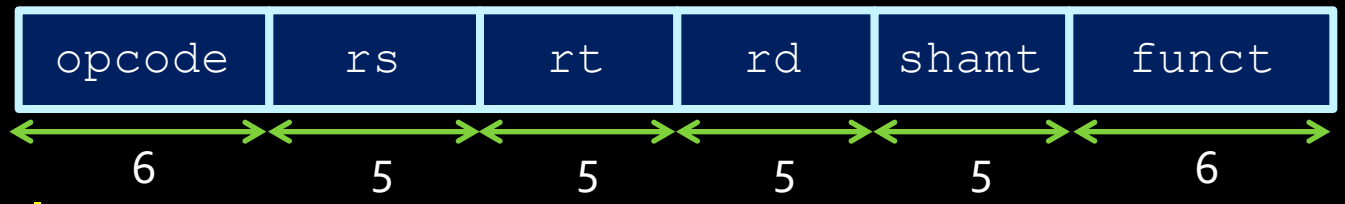
Opcodes

- The first six digits of the instruction (the opcode) will determine the instruction type.
  - Except for "R-type" instructions (marked in yellow)
  - For these, opcode is 000000, and last six digits denote the function.

Instruction	Op/Func	Instruction	Op/Func
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

MIPS instruction types

- R-type:

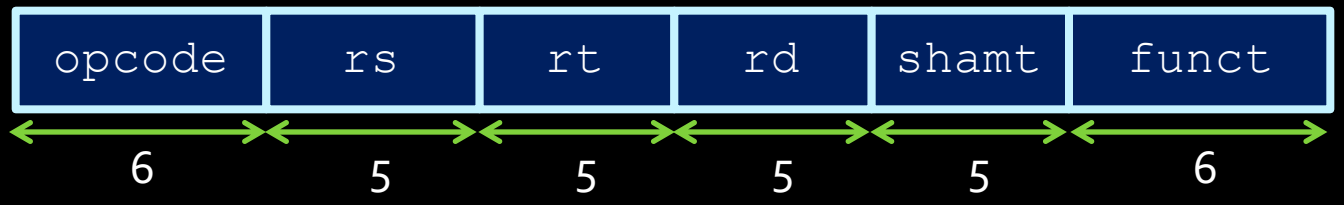


- I-type:

- J-type:

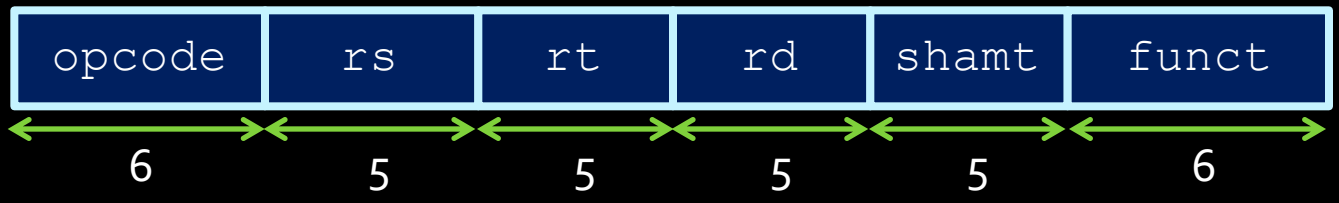
Read the first 6 bits first, then you know how to break it down.

R-type instructions



- Short for "register-type" instructions.
  - Because they operate on the registers, naturally.
- These instructions have fields for specifying up to three registers and a shift amount.
  - Three registers: two source registers (`rs` & `rt`) and one destination register (`rd`).
  - A field is usually coded with all `0` bits when not being used.
- The opcode for all R-type instructions is **`000000`**.
- The function field specifies the type of operation being performed (add, sub, and, etc).

Examples

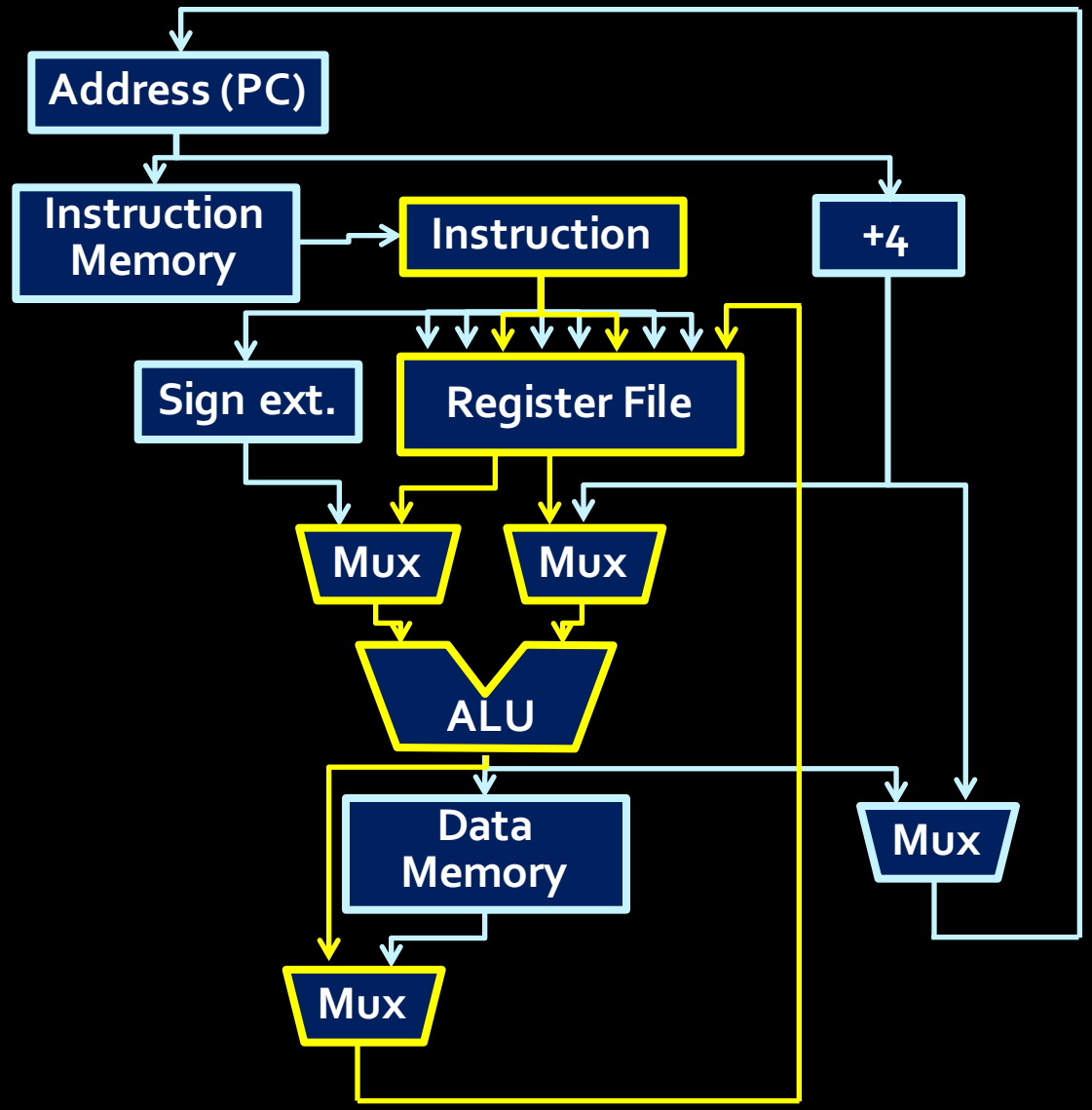



```
Reg_5 = Reg_6 XOR Reg_17
```


```
Left shift what's in Reg_17 by 12 bits
and store result in Reg_5
```

R-type instruction datapath

- For the most part, the `funct` field tells the ALU what operation to perform.
- `rs` and `rt` are sent to the register file, to specify the ALU operands.
  - Register `$0` and `$1` are usually held in reserve.
- `rd` is also sent to the register file, to specify the location of the result.

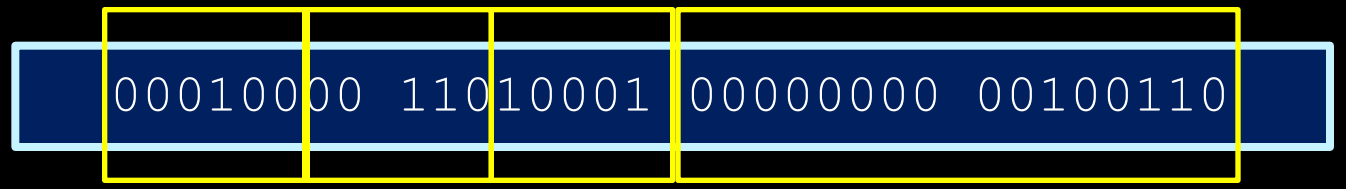


I-type instructions

- These instructions have a 16-bit immediate field.
- This field a constant value, which is used for:
  - an immediate operand,
  - a branch target offset (e.g., in branch if equal op), or
  - a displacement for a memory operand (e.g., in load op).
- For **branch** target offset operations, the immediate field contains the signed difference between the current address stored in the PC and the address of the target instruction.
  - This offset is stored with the two low order bits dropped. The dropped bits are always 0 since instructions are word-aligned.

Word-aligned: Offsets increment by 4, like 0 (0000), 4 (0100), 8 (1000), 12 (1100), note that the two lowest bits are always 00.

Examples



Branch on equal (BEQ) **Reg 6!** **Reg 17!** **Offset = 10011000 = 152**

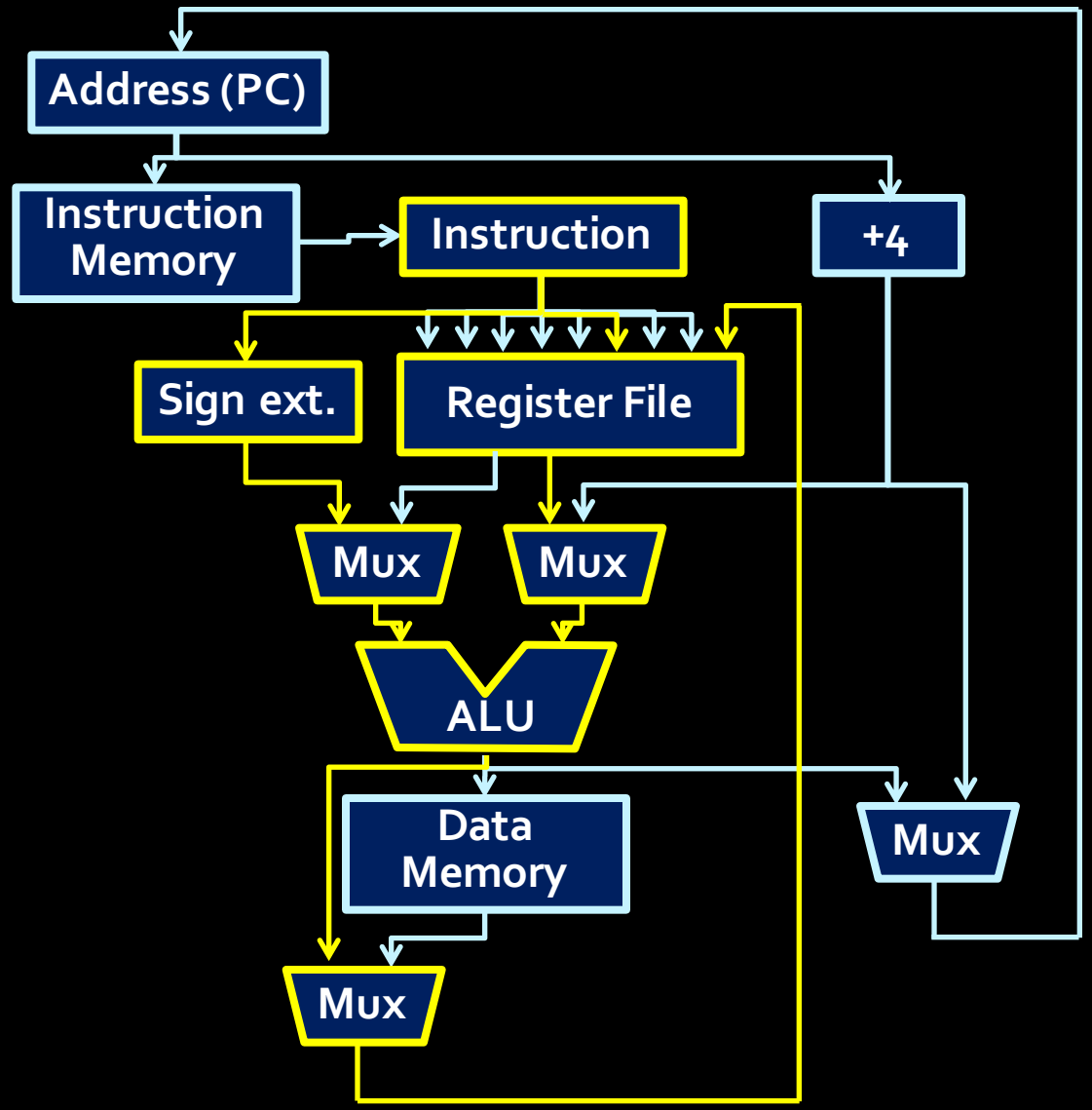
```
If Reg_6 == Reg_17:
    PC += 152
Else:
    PC += 4
```

Examples


```
Load one byte from MEM[Reg_6+38] to Reg_17
```

I-type instruction datapath

- Example #1: Immediate arithmetic operations, with result stored in registers.



Interlude: Sign extension

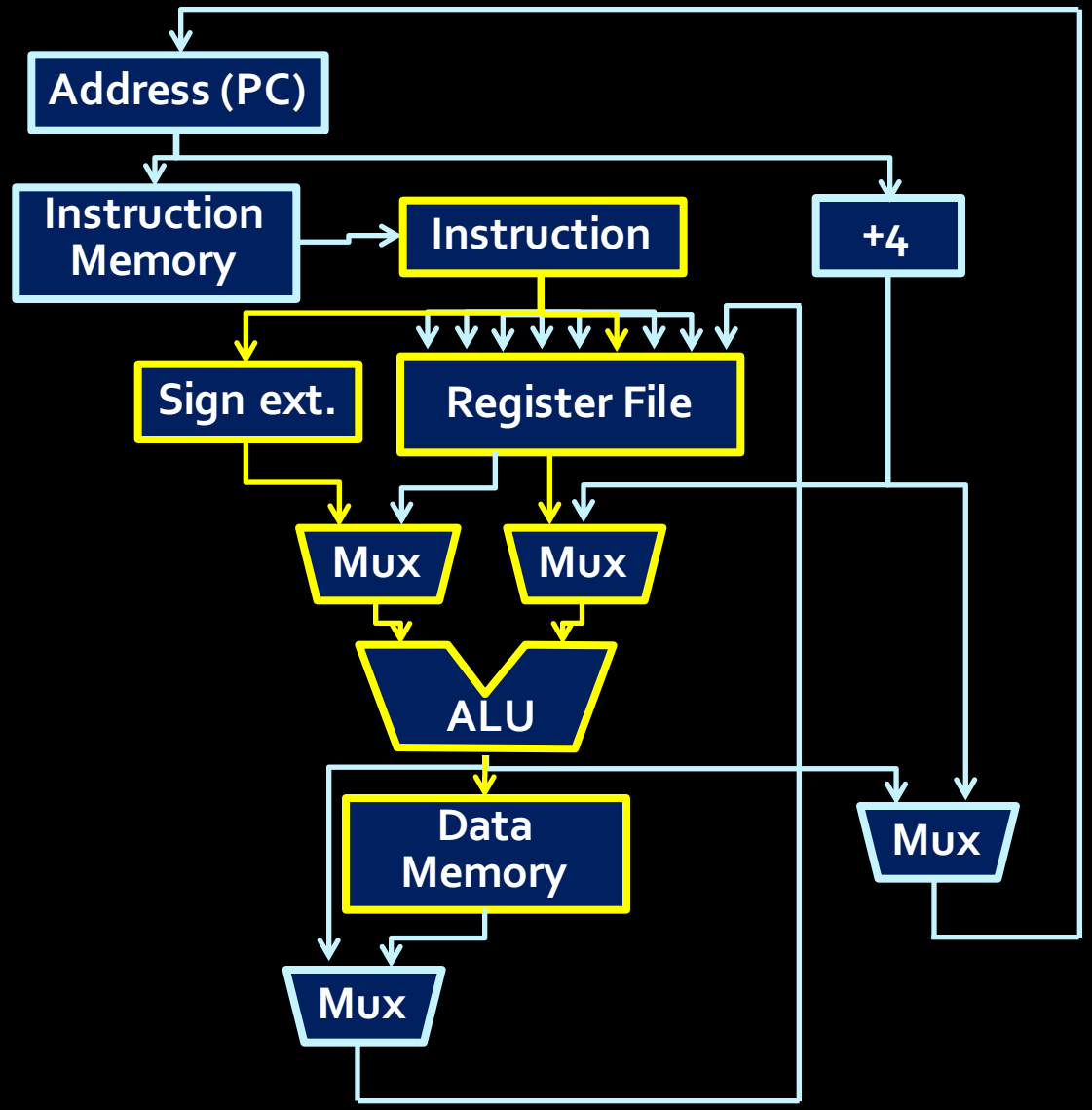
Sign ext.

- The immediately value we get from an I-type instruction is 16-bit long.
- But all operands of ALU are supposed to be 32-bit long.
- So fill the upper 16 bits of the number with the **sign-bit**

- E.g., 1100 1000 1000 1000 becomes
- 1111 1111 1111 1111 1100 1000 1000 1000

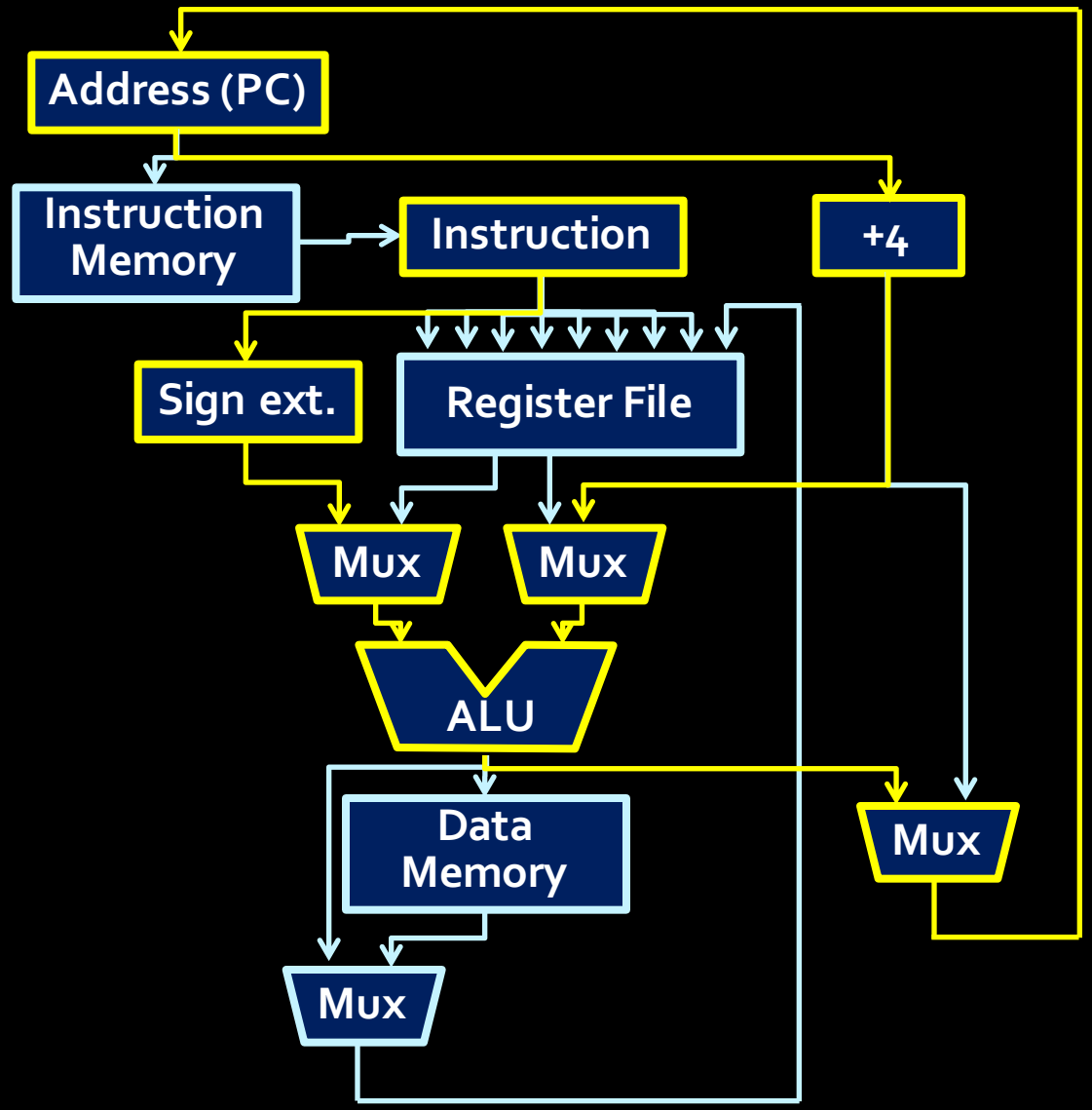
I-type instruction datapath

- Example #2: Immediate arithmetic operations, with result stored in memory.



I-type instruction datapath

- Example #3: Branch instructions.
  - Output is written to PC, which looks to that location for the next instruction.



J-type instructions

- Only two J-type instructions:
  - jump (`j`)
  - jump and link (`jal`)
- These instructions use the 26-bit coded address field to specify the target of the jump.
  - The first four bits of the destination address are the same as the current bits in the program counter.
  - The bits in positions `27` to `2` in the address are the 26 bits provided in the instruction.
  - The bits at positions `1` and `0` are always `0` since instructions are word-aligned.

Examples

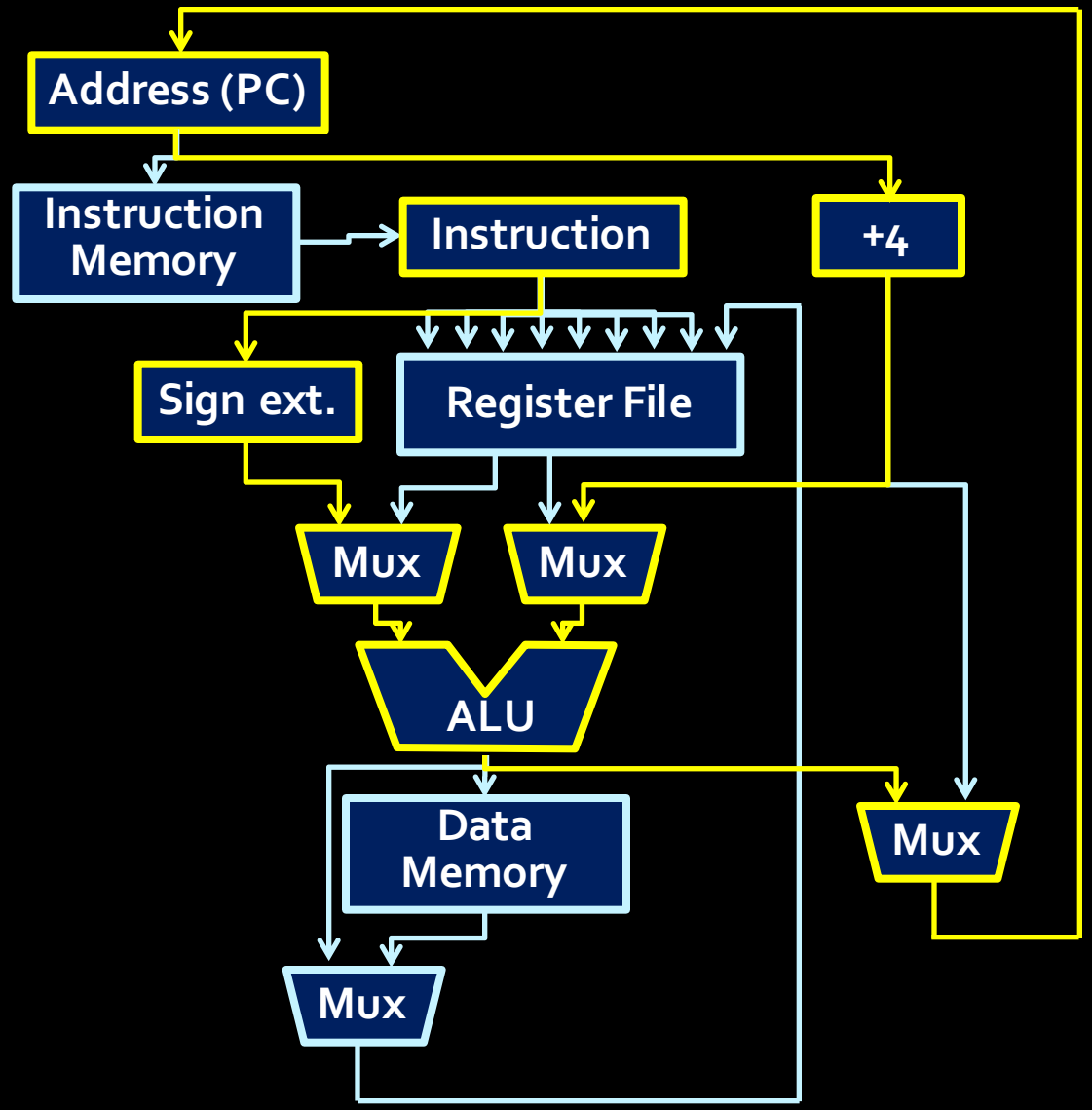
Jump (J)

destination address= {PC[31:28], whats-above, 00}

PC jumps to address
{PC[31:28] (4 bits), what's above (26 bits), 00 (2 bits)}
(32 bits total)

J-type instruction datapath

- Jump and branch use the datapath in similar but different ways:
  - Branch calculates new PC value as old PC value + offset. (relative)
  - Jump loads an immediate value over top of the old PC value. (absolute)



Takeaway

Different instructions flow in different datapaths.

In other words, if we can control the paths of flow, then we can control what instruction to execute.

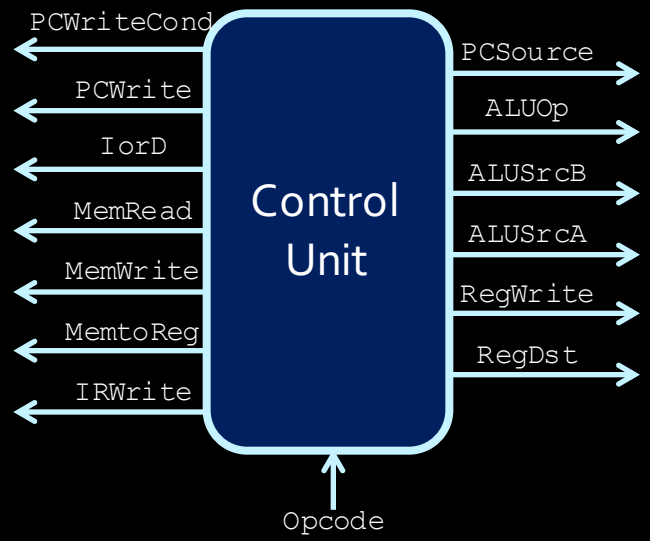
Datapath control

Datapath control

- These instructions are executed by turning various parts of the datapath on and off, to direct the flow of data from the correct source to the correct destination.
- What tells the processor to turn on these various components at the correct times?

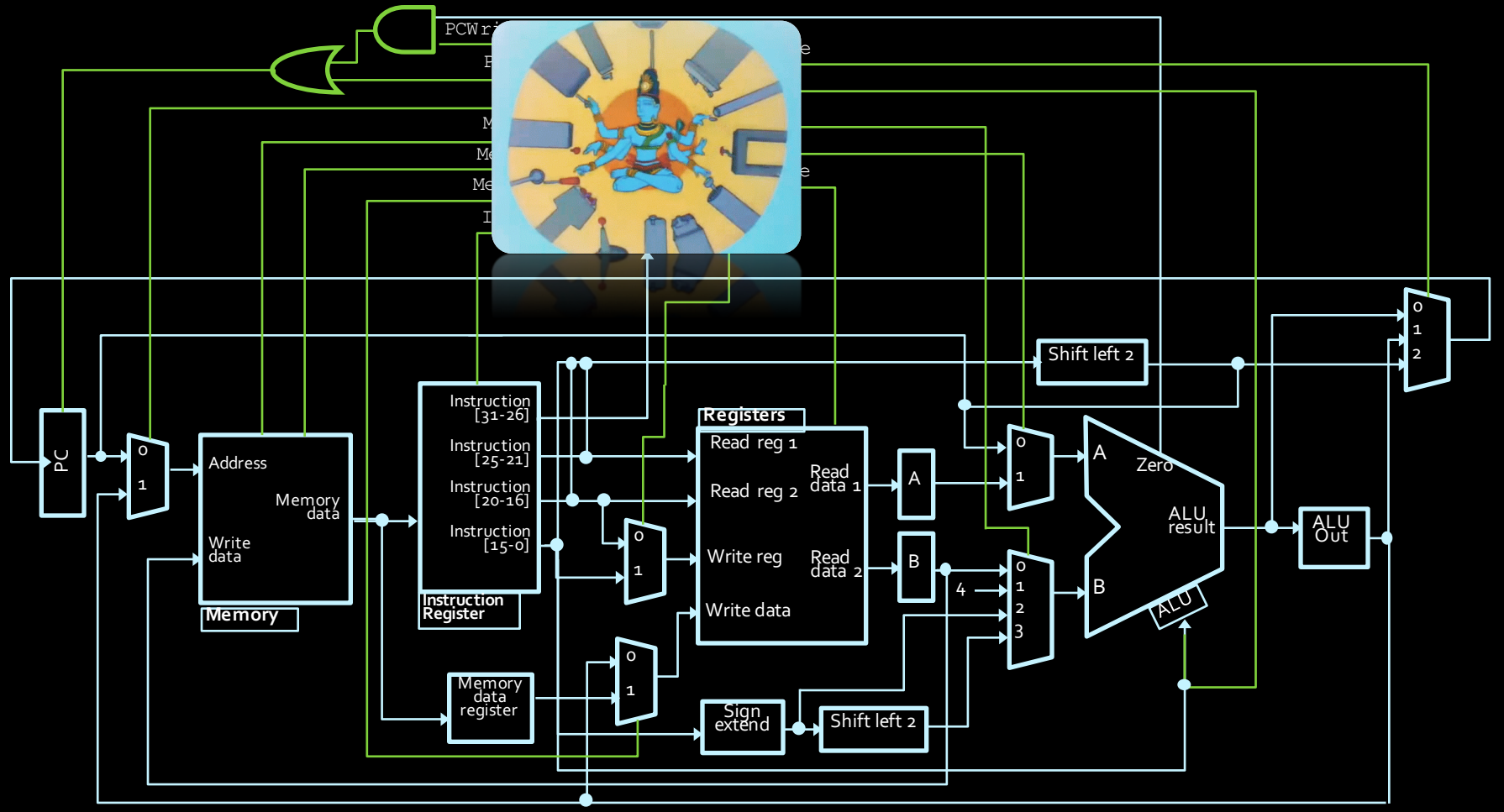
Control unit

- The control unit takes in the opcode from the current instruction, and sends signals to the rest of the processor.



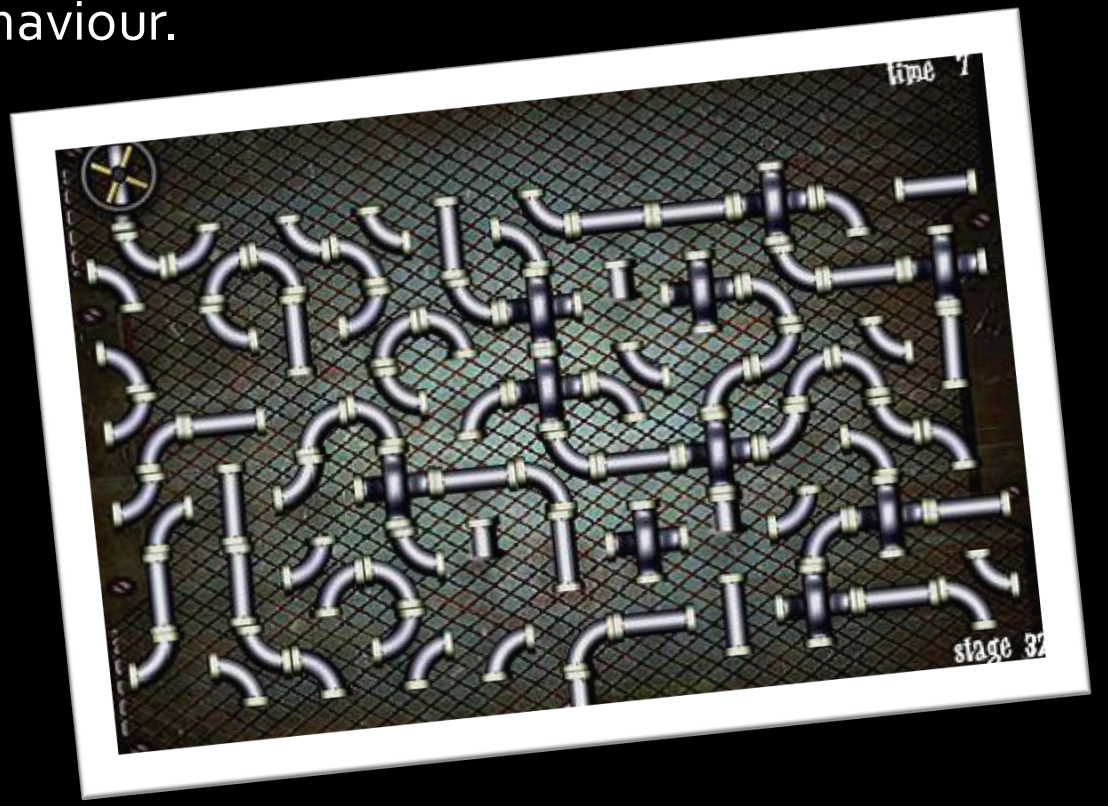
- Within the control unit is a finite state machine that can occupy multiple clock cycles for a single instruction.
  - The control unit send out different signals on each clock cycle, to make the overall operation happen.

- The control unit sends signals (green lines) to various processor components to enact all possible operations.



Signals → instructions

- A certain combination of signals will make data flow from some source to some destination.
  - Just need to figure out what signals produce what behaviour.



Control unit signals

- `PCWrite`: Write the ALU output to the PC.
- `PCWriteCond`: Write the ALU output to the PC, only if the Zero condition has been met.
- `IorD`: For memory access; short for "Instruction or Data". Signals whether the memory address is being provided by the PC (for instructions) or an ALU operation (for data).
- `MemRead`: The processor is reading from memory.
- `MemWrite`: The processor is writing to memory.
- `MemToReg`: The register file is receiving data from memory, not from the ALU output.
- `IRWrite`: The instruction register is being filled with a new instruction from memory.

More control unit signals

- `PCSource`: Signals whether the value of the PC resulting from an jump, or an ALU operation.
- `ALUOp` (3 wires): Signals the execution of an ALU operation.
- `ALUSrcA`: Input A into the ALU is coming from the PC (value=`0`) or the register file (value=`1`).
- `ALUSrcB` (2 wires): Input B into the ALU is coming from the register file (value=`0`), a constant value of 4 (value=`1`), the instruction register (value=`2`), or the shifted instruction register (value=`3`).
- `RegWrite`: The processor is writing to the register file.
- `RegDst`: Which part of the instruction is providing the destination address for a register write (`rt` versus `rd`).

Example instruction

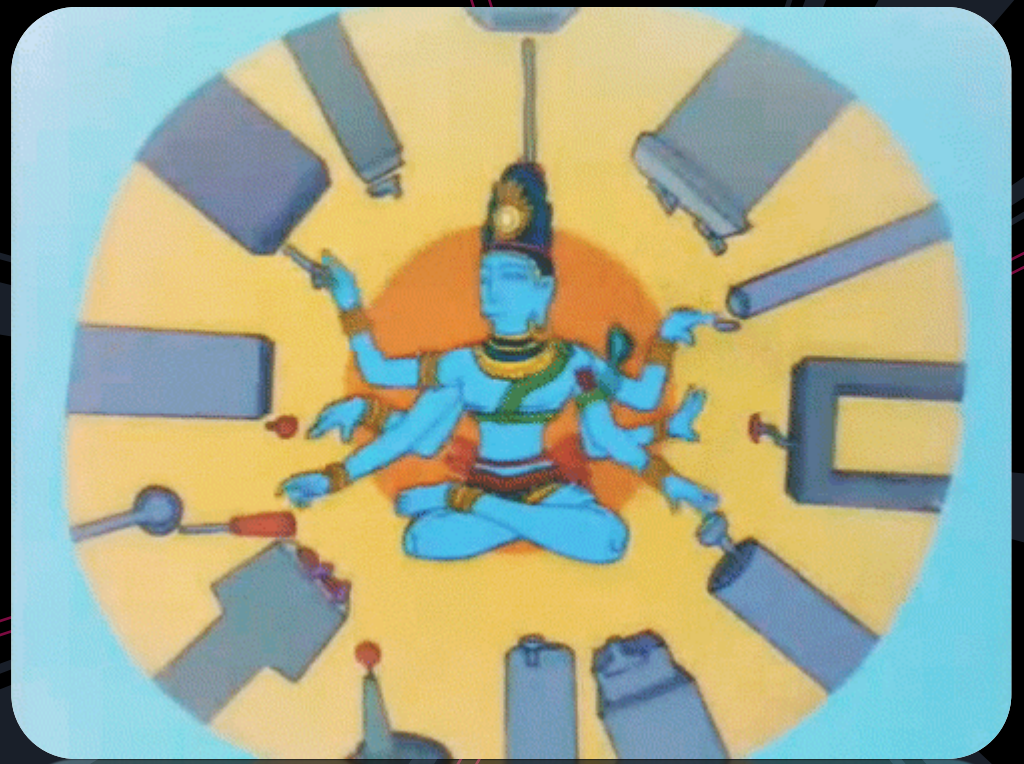
- **addi $t7, $t0, 42**
  - PCWrite = 0
  - PCWriteCond = 0
  - IorD = X
  - MemWrite = 0
  - MemRead = 0
  - MemToReg = 0
  - IRWrite = 0
  - PCSource = X
  - ALUOp = 001 (add)
  - ALUSrcA = 1
  - ALUSrcB = 10
  - RegWrite = 1
  - RegDst = 0

Setting these signals will result in adding register $t0 by 42 and storing result in register $t7

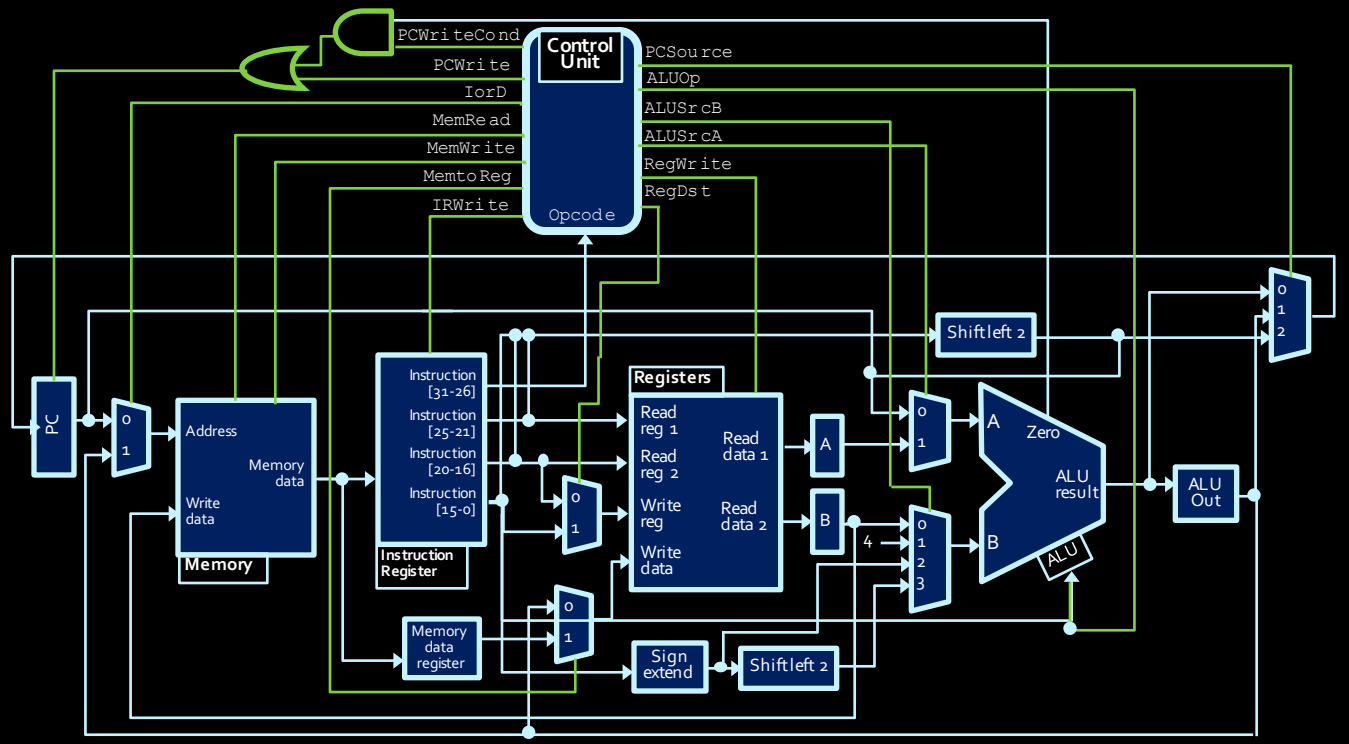
- `addi $t7, $t0, 42`

This is a line of
assembly language

Controlling the Datapath



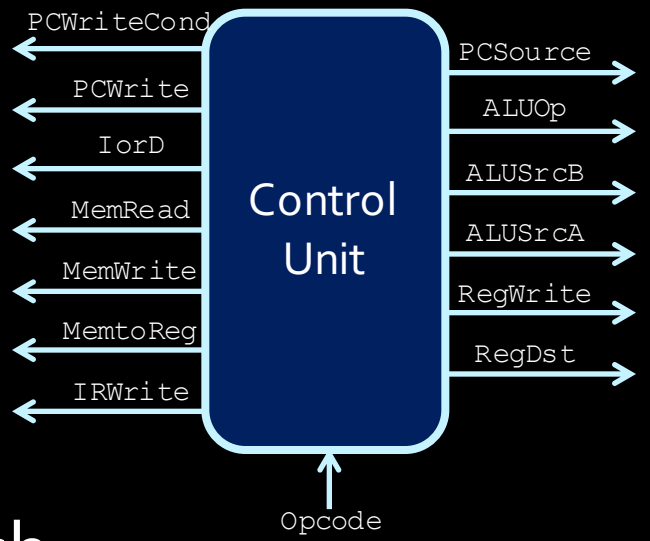
MIPS Datapath



- So, how do we do the following?
  - Increment the PC to the next instruction position.
  - Store $t1 + 12 into the PC.
  - Assuming that register $t3 is storing a valid memory address, fetch the data from that location in memory and store it in $t5.

Controlling the signals

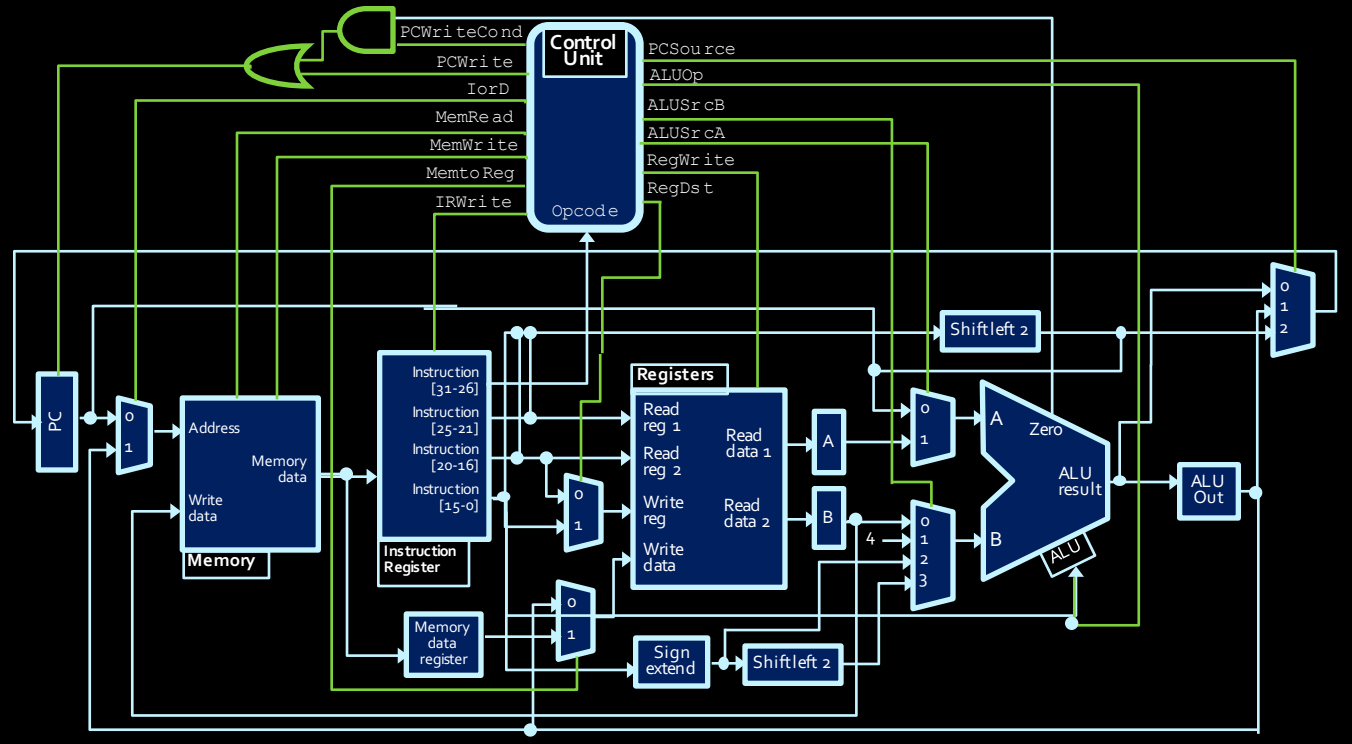
- Need to understand the role of each signal, and what value they need to have in order to perform the given operation.
- So, what's the best approach to make this happen?



Basic approach to datapath

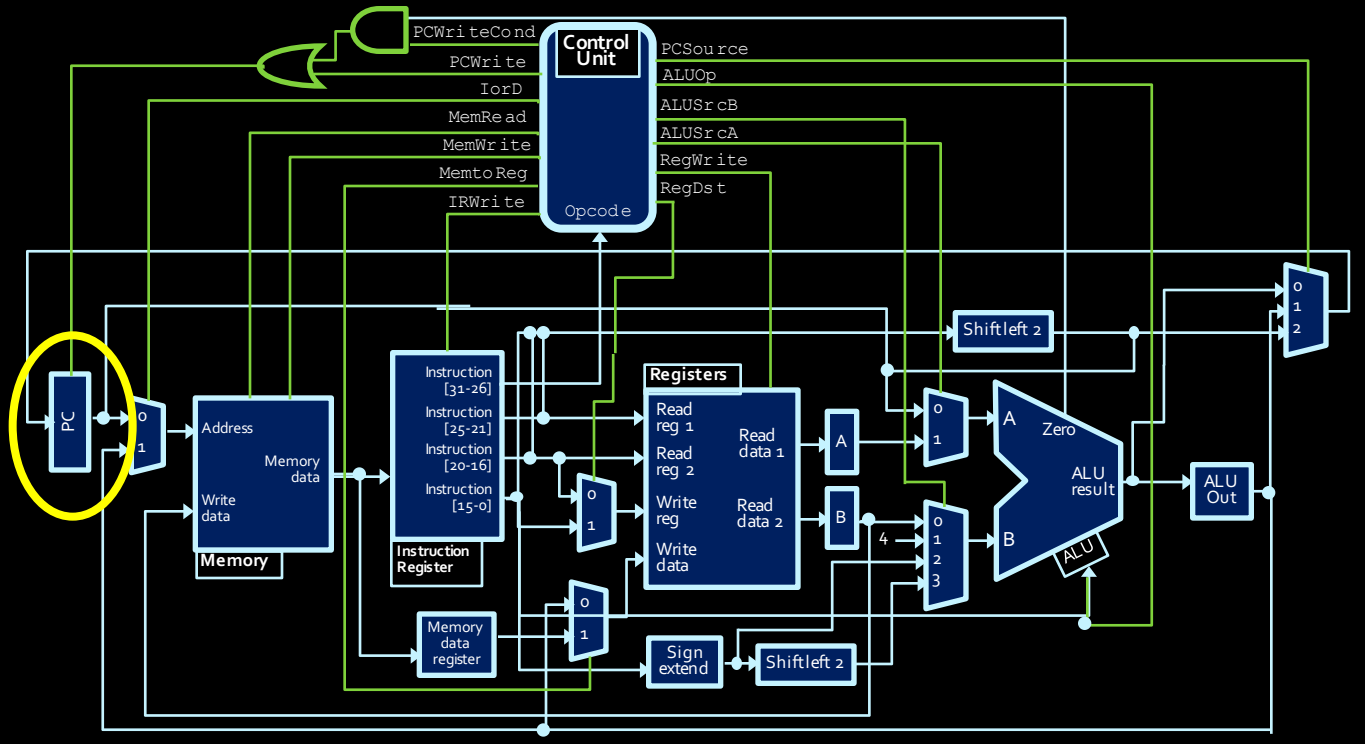
1. Figure out the data source(s) and destination.
2. Determine the path of the data.
3. Deduce the signal values that cause this path:
   a) Start with `Read` & `Write` signals (at most one can be high at a time).
   b) Then, mux signals along the data path.
   c) Non-essential signals get an `X` value.

Example #1: Incrementing PC



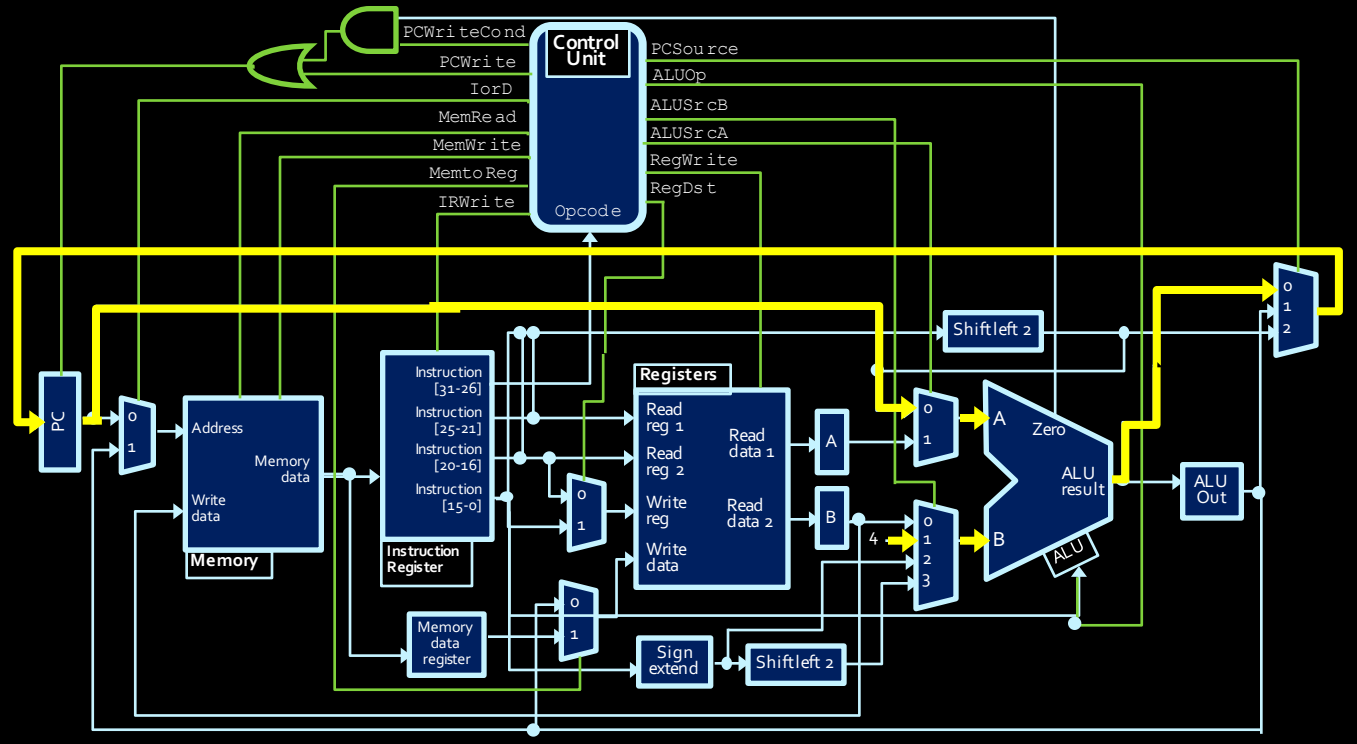
- Given the datapath above, what signals would the control unit turn on and off to increment the program counter by 4?

Example #1: Incrementing PC



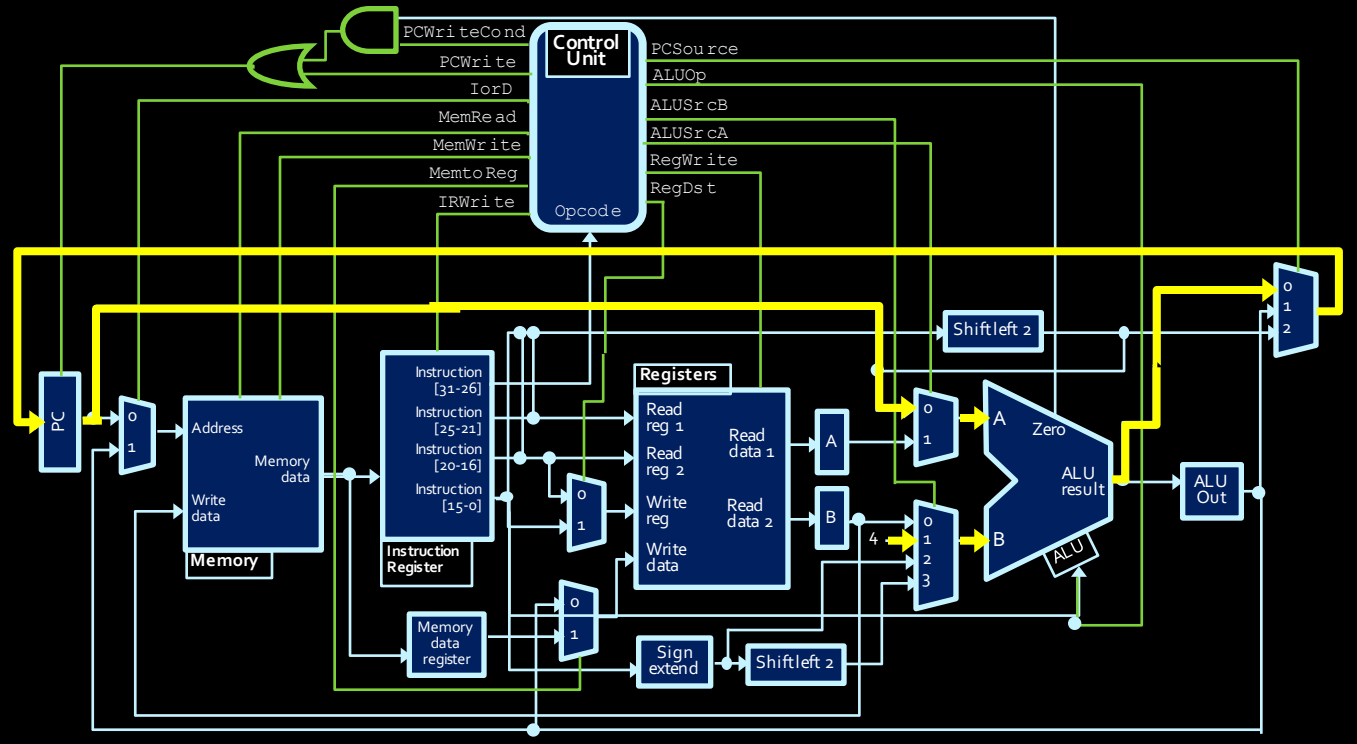
- Step #1: Determine data source and destination.
  - Program counter provides source,
  - Program counter is also destination.

Example #1: Incrementing PC



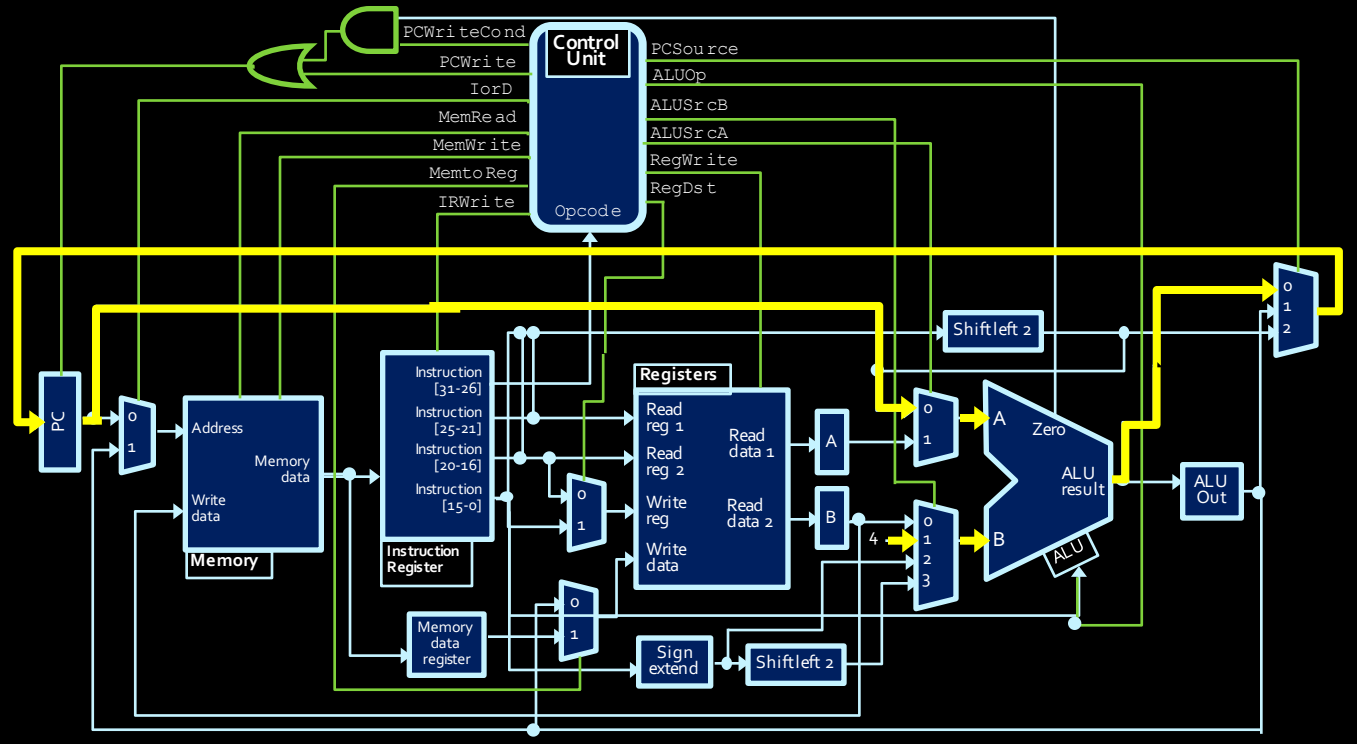
- Step #2: Determine path for data
  - Operand A for ALU: Program counter
  - Operand B for ALU: Literal value 4
  - Destination path: Through mux, back to PC

Example #1: Incrementing PC



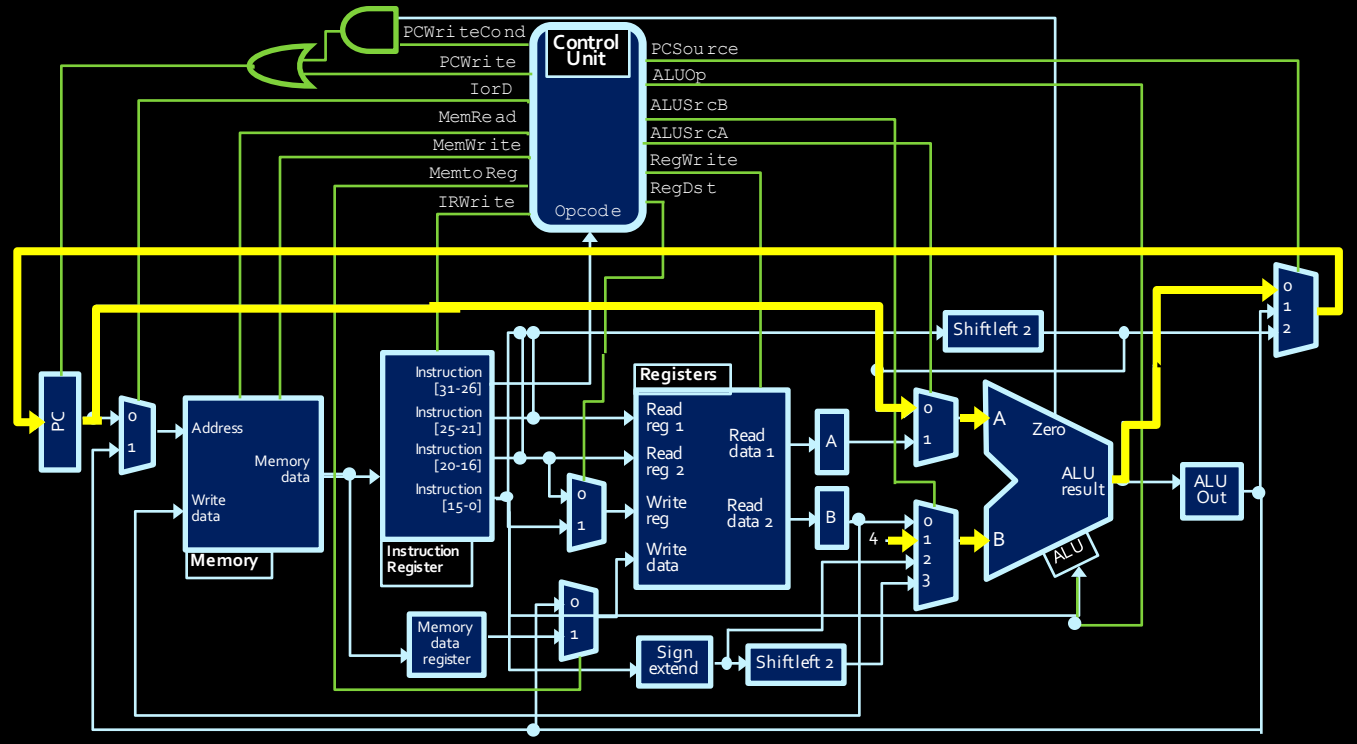
- Setting signals for this datapath:
  1. Read & Write signals:
     - `PCWrite` is high, all others are low.

Example #1: Incrementing PC



- Setting signals for this datapath:
  2. Mux signals:
     - `PCSource` is `0`, `ALUSrcA` is `0`, `ALUSrcB` is `1`
     - all others are "don't cares".

Example #1: Incrementing PC



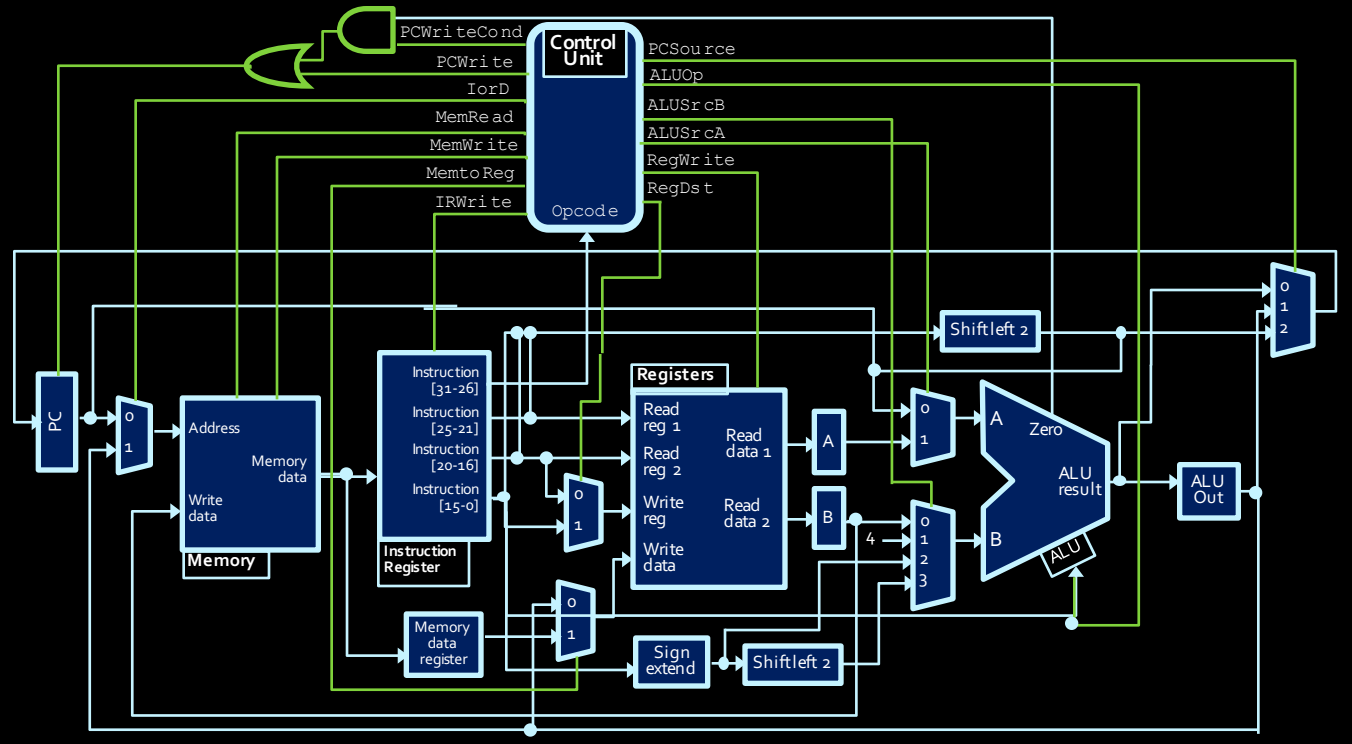
- Other signals for this datapath:
  - `ALUOp` is `001` (Cin =0 , S1 = 0, So = 1: A+B)
  - `PCWriteCond` is `X` when `PCWrite` is `1`
    - Otherwise it is `1` except when branching.

Example #1 (final signals)

- PCWrite = 1
- PCWriteCond = X
- IorD = X
- MemRead = 0
- MemWrite = 0
- MemToReg = X
- IRWrite = 0
- PCSource = 0
- ALUOp = 001
- ALUSrcA = 0
- ALUSrcB = 01
- RegWrite = 0
- RegDst = X

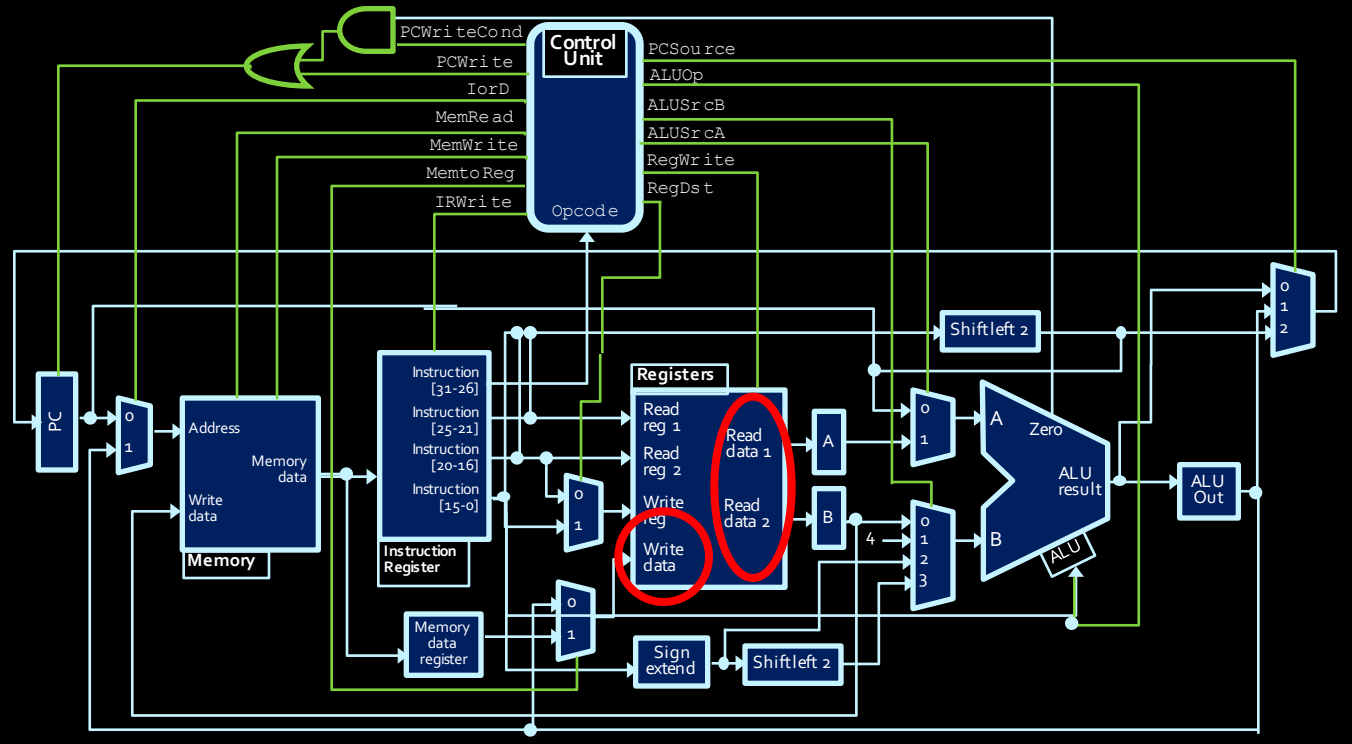
Another example

add $r7, $r1, $r2



- Given the datapath above, what signals would the control unit turn on and off in order to add `$r1` to `$r2` and store the result in `$r7`?

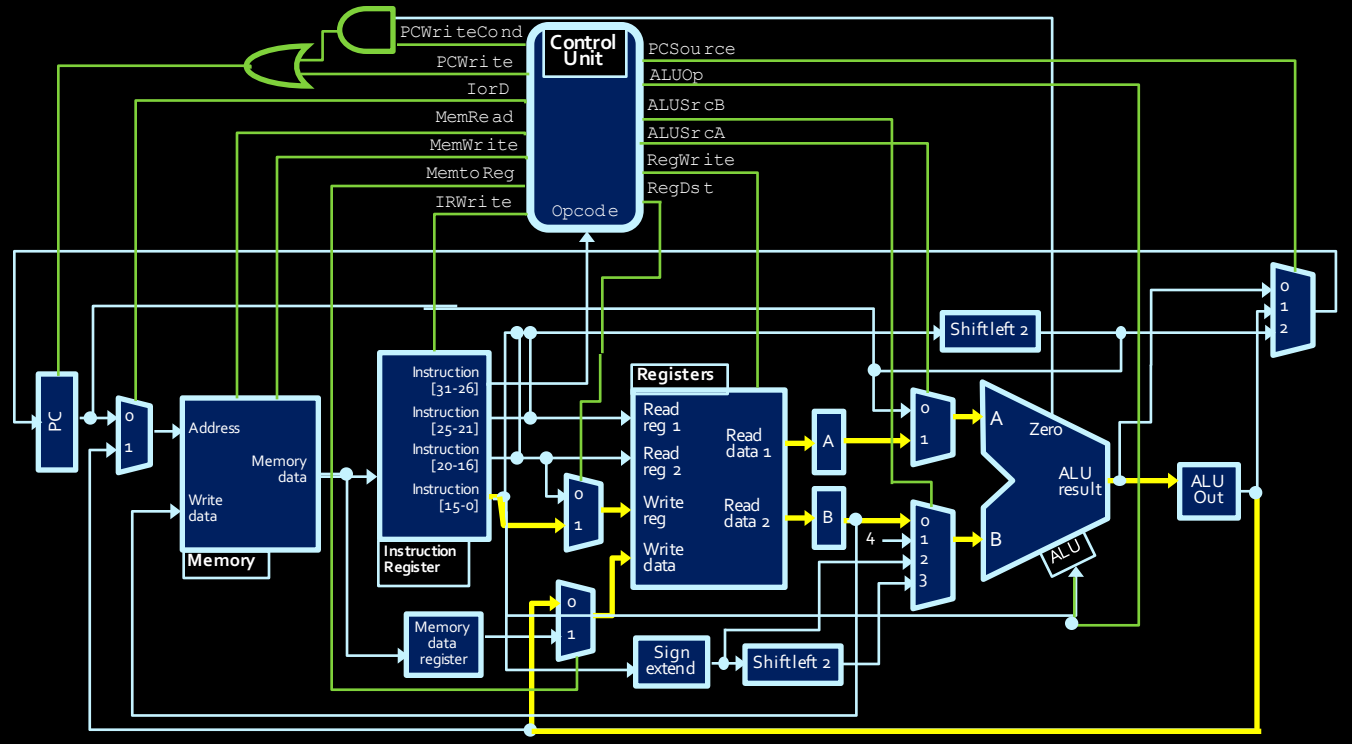
add $r7, $r1, $r2



- Step #1: Data source and destination
  - Data starts in register block.
  - Data goes to register block.

Question #1 (cont'd)

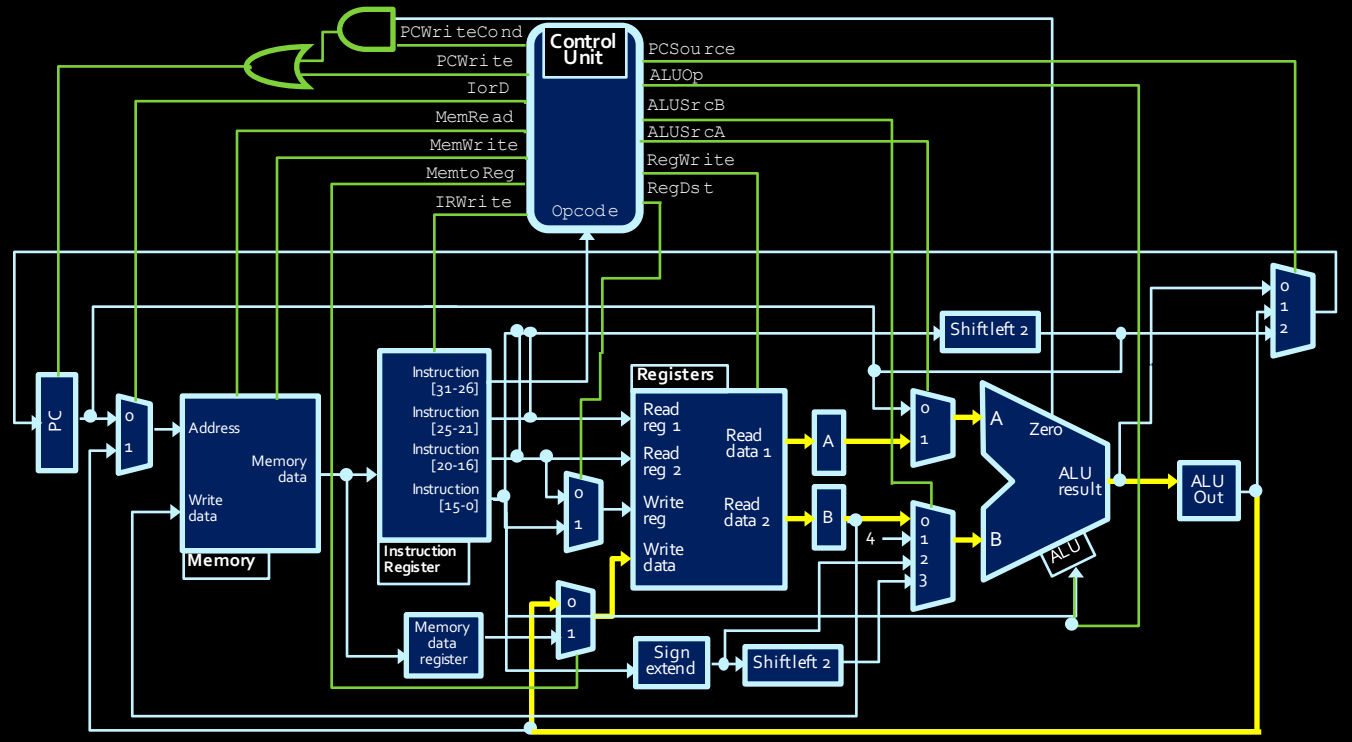
add $r7, $r1, $r2



- Step #2: Determine the path of the data
  - Data needs to go through the ALU before heading back into the register file.

Question #1 (cont'd)

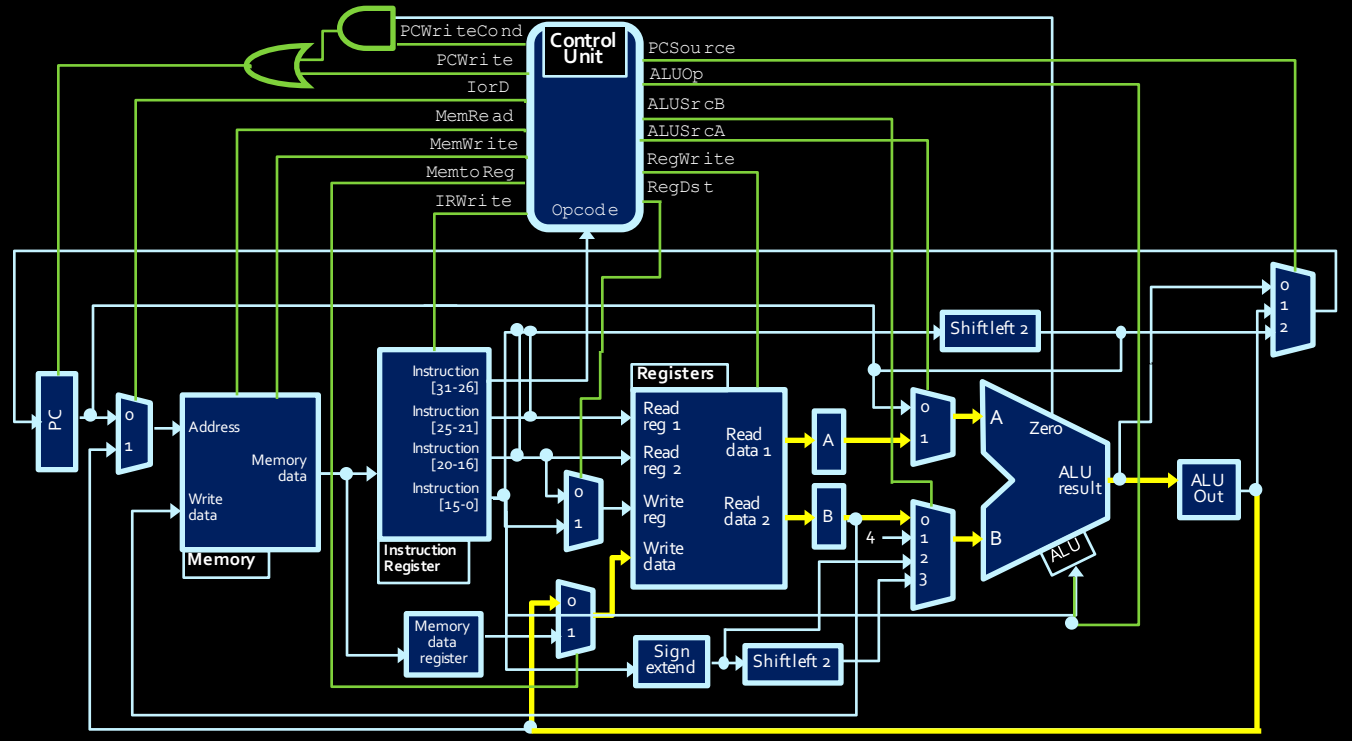
add $r7, $r1, $r2



- Step #3a: `Read` & `Write` signals
  - Only `RegWrite` needs to be high.
  - `PCWrite`, `PCWriteCond`, `MemRead`, `MemWrite`, `IRWrite` would be low.

Question #1 (cont'd)

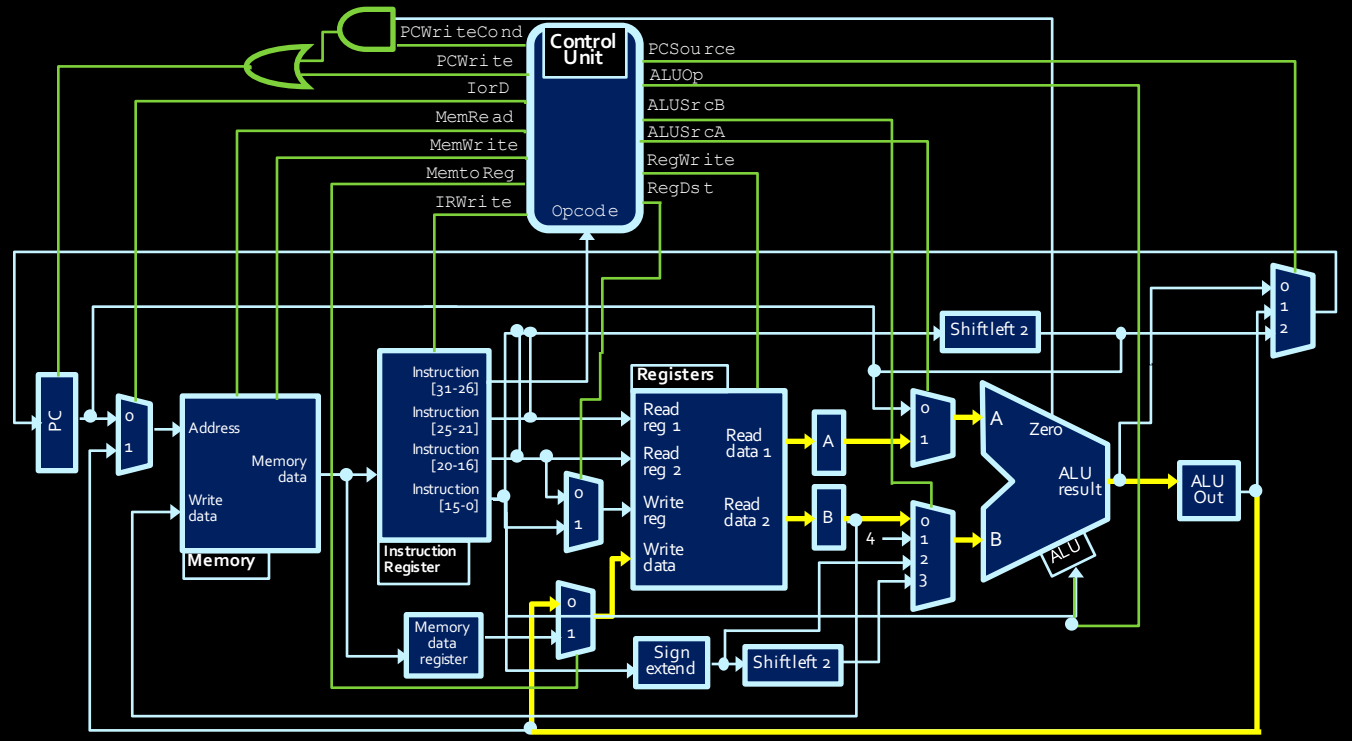
add $r7, $r1, $r2



- Step #3b: Relevant mux signals
  - Muxes before ALU: `ALUSrcA` → `1`, `ALUSrcB` → `00`.
  - `ALUOp` → `001` (Add)
  - Mux before registers: `MemToReg` → `0`

Question #1 (cont'd)

add $r7, $r1, $r2



- Step #3c: Irrelevant mux signals
  - No writing to PC: `PCSource` → `X`.
  - No reading from memory: `IorD` → `X`.

Question #1 (cont'd)

- PCWrite = 0
- PCWriteCond = 0
- IorD = X
- MemRead = 0
- MemWrite = 0
- MemToReg = 0
- IRWrite = 0
- PCSource = X
- ALUOp = 001
- ALUSrcA = 1
- ALUSrcB = 00
- RegWrite = 1
- RegDst = 1

Note: RegDst rule
high for 3-register operations
low for 2-register operations
X if not using register file

The Tale of "Hello world"

1. You, the programmer, write a piece of code called hello.c/java/whatever
2. You compile the code, which translate the code into machine instructions and save the in an executable file (e.g., hello, hello.exe)
3. You run the executable, OS load the executable (the instructions) into memory, set PC.
4. CPU loads the instruction pointed by PC into instruction register.
5. Control Unit checks the opcode (first 6 bits), decode the rest of the instruction and send signals to setup the datapath (billions of MOSFETs switching ON/OFF).
6. Data flow through the datapath, electrons move around…
7. And BOOM!, "hello world" shows up on your screen

The Tale of "Hello world"