Introduction to pipelined processors

Kevin Wang

UNIVERSITY OF MICHIGAN

16 November 2023

Content for this talk borrows heavily from my home university

- EECS 370, Intro to Computer Organization at the University of Michigan
  - The toy assembly language and pipeline simulator in this talk were created for this class
  - Class does **NOT** share most materials. This talks' contents are borrowed from publicly available content and cited accordingly

- I will briefly talk about the course, more details at https://eecs370.github.io/

This will likely be a very difficult talk because:

- I am using a new assembly language to discuss this topic
- I am combining lots of ideas that are complicated on their own to discuss another complicated topic
- It's all in English

- You should have questions and I will regularly prompt you to ask them
  - Feel free to ask questions after the talk or read more into this at a later time
  - Slides available – Kamil has them and I will upload to my GitHub repo

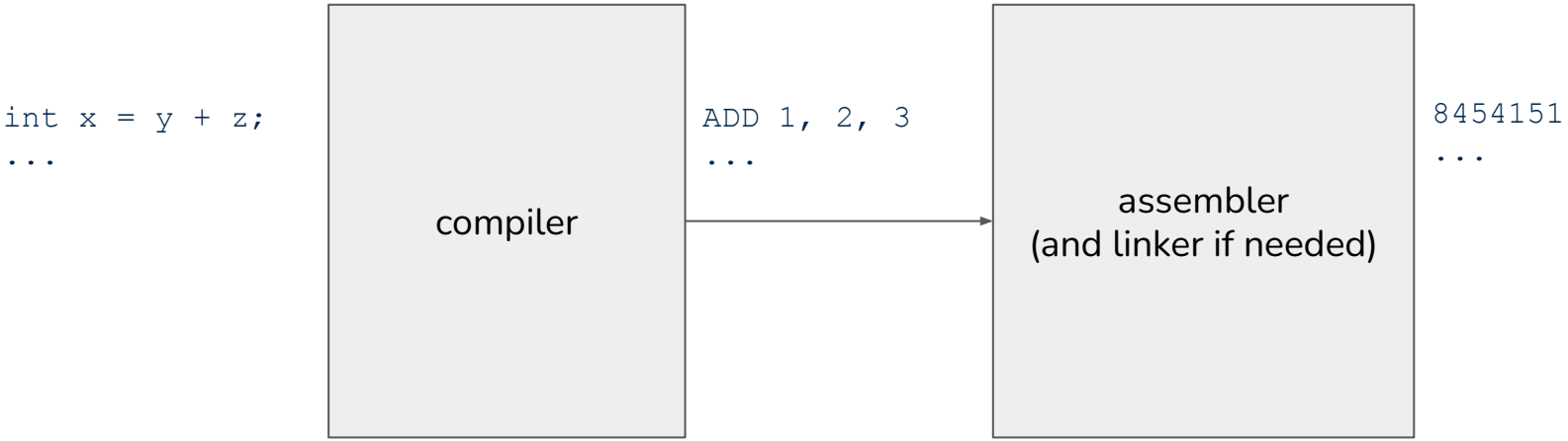
I assume you already know...

- ...how code becomes assembly language instructions
- ...how instructions can use multiple parts of the CPU
- ...how single-cycle and multi-cycle processor CPUs execute instructions

- I will still go over a **very** small bit of what we teach at UofM

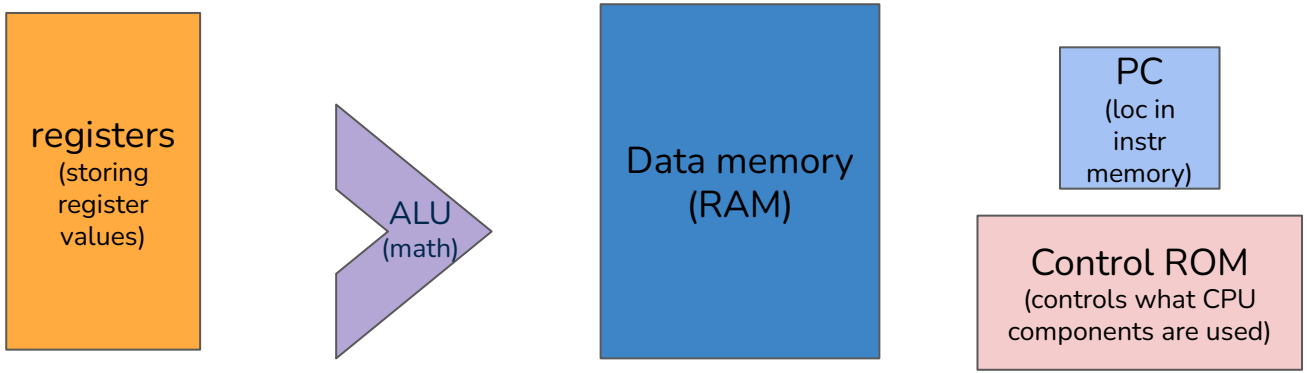
Reviewing what we know: transforming into assembly code

The compiler and assembler transform human-readable code into machine code



Reviewing what we know: basics of a CPU

- The CPU is the part of a computer that “does stuff”
- There are different parts of the CPU that do different things*



*actual CPUs have many more components! This is a subset for simplification.

Reviewing what we know: processor cycles

- **Cycle:** regular unit of time in which a CPU does something
  - Depending on the CPU architecture, will be based on different aspects of the CPU

for **single-cycle** processor, 1 instr = 1 cycle

```
lw   0 4 three
add 2 1 3
beq 3 4 done
```

same latency

for **multi-cycle** processor, slowest component = 1 cycle

```
lw   0 4 three
add 2 1 3
beq 3 4 done
```

different latencies (the memory operation will probably be slowest)

I will use a basic toy assembly language for this lecture

- Created by EECS 370 staff for educational purposes
  - For pipelining, we skip jalr because it presents a lot of complexity that isn't very educational

https://eecs370.github.io/project_1_spec/

Assembly language name for instruction	Instruction Opcode in binary	Action
`add` (R-type instruction)	0b000	Add contents of `regA` with contents of `regB`, store results in `destReg`.
`nor` (R-type instruction)	0b001	Nor contents of `regA` with contents of `regB`, store results in `destReg`. This is a bitwise nor; each bit is treated independently.
`lw` (I-type instruction)	0b010	"Load Word"; Load `regB` from memory. Memory address is formed by adding `offsetField` with the contents of `regA`. Behavior is defined only for memory addresses in the range [0, 65535].
`sw` (I-type instruction)	0b011	"Store Word"; Store `regB` into memory. Memory address is formed by adding `offsetField` with the contents of `regA`. Behavior is defined only for memory addresses in the range [0, 65535].
`beq` (I-type instruction)	0b100	"Branch if equal" If the contents of `regA` and `regB` are the same, then branch to the address `PC+1+offsetField`, where `PC` is the address of this beq instruction.
`jalr` (J-type instruction)	0b101	"Jump and Link Register"; **First** store the value `PC+1` into `regB`, where `PC` is the address where this `jalr` instruction is defined. **Then** branch (set PC) to the address contained in `regA`. **Note** that this implies if `regA` and `regB` refer to the same register, the net effect will be jumping to `PC+1`.
`halt` (O-type instruction)	0b110	Increment the `PC` (as with all instructions), then halt the machine (let the simulator notice that the machine halted).
`noop` (O-type instruction)	0b111	"No Operation (pronounced no op)" Do nothing.

Very basic program in LC2K assembly

- load three numbers from memory into registers
- add r2, r1 and store result in r3
- Check if val in r3 == val in r4 (it does), branch to done
- halt

```
        lw   0 1 one
        lw   0 2 two
        lw   0 4 three
        add 2 1 3
        beq 3 4 done
done    halt
one     .fill 1
two     .fill 2
three .fill 3
```

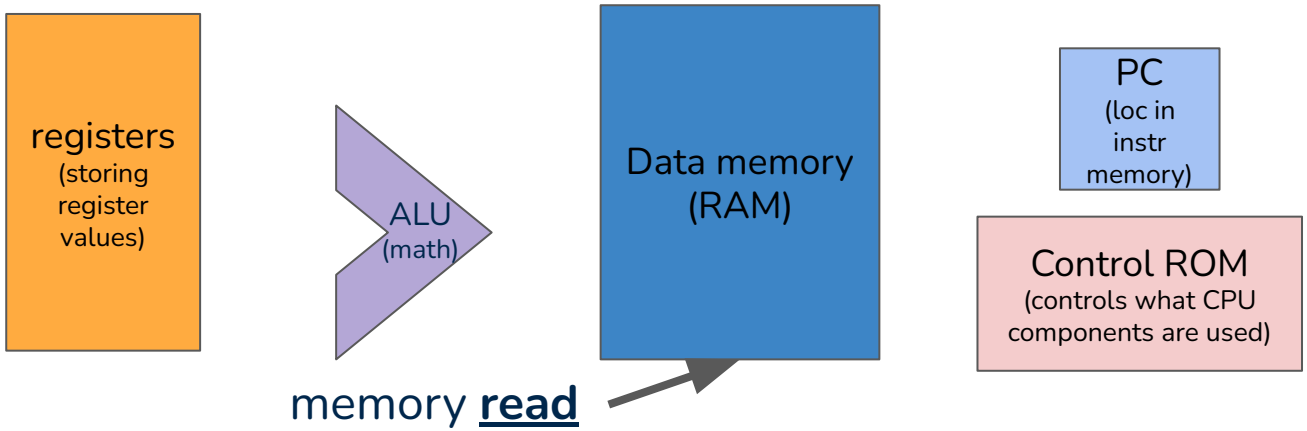
https://eecs370.github.io/project_1_spec/

Note how different instructions use different parts of the CPU

- add: add values in two registers and store result in dest register
- nor: nor values in two registers and store result in dest register
- beq: compare values in two registers and branch if equal

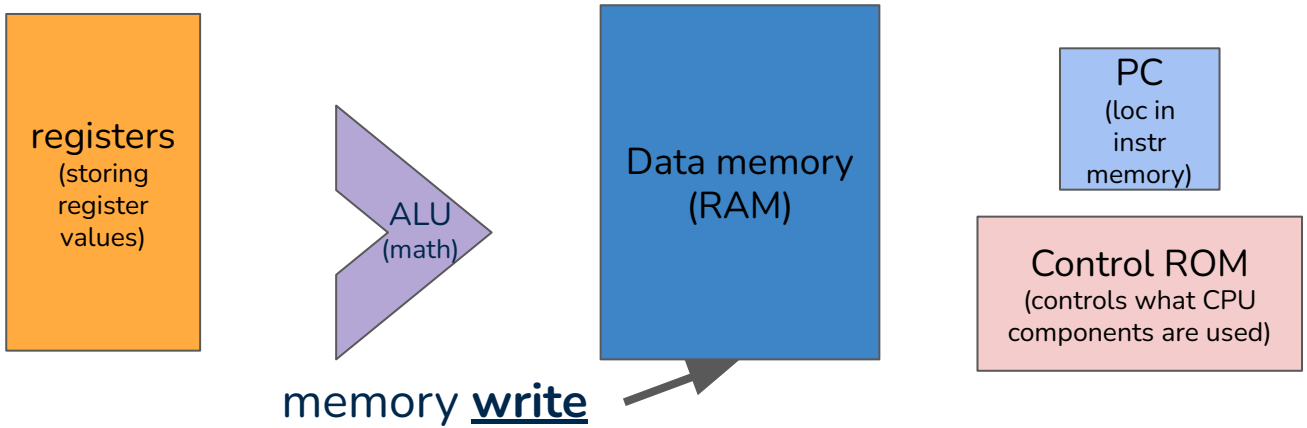
Note how different instructions use different parts of the CPU

- lw: load value from memory into register
  - addr formed by adding a register value with literal offset



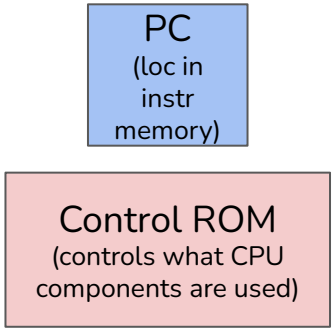
Note how different instructions use different parts of the CPU

- sw: load value from memory into register
  - addr formed by adding a register value with literal offset
- beq: compare values in two registers and branch if equal



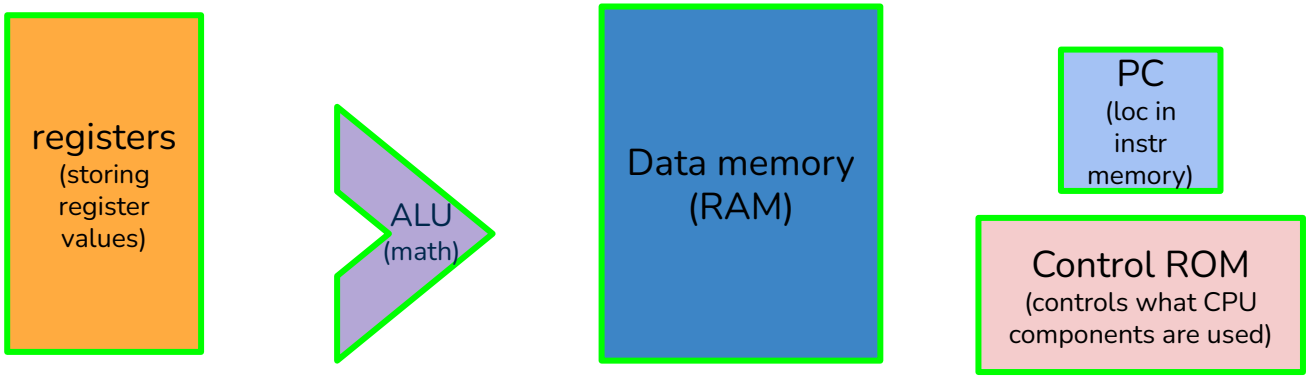
Note how different instructions use different parts of the CPU

- halt: stop program
- noop: do nothing



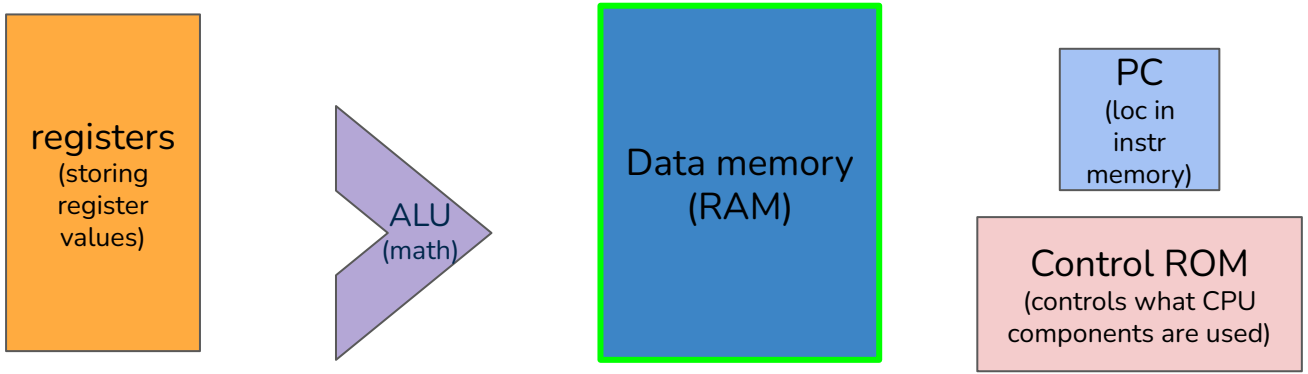
Multi-cycle processors optimize based on component latencies

- One cycle is the slowest CPU operation, not the slowest instruction
  - For most benchmarks, this makes multi-cycle processors faster than single-cycle
  - For LC2K, memory read is the slowest operation
- Pipelining also bases cycle time on individual operations



Single-cycle cycle period is calculated by adding latencies of all* components

Multi-cycle cycle period just looks at the slowest component

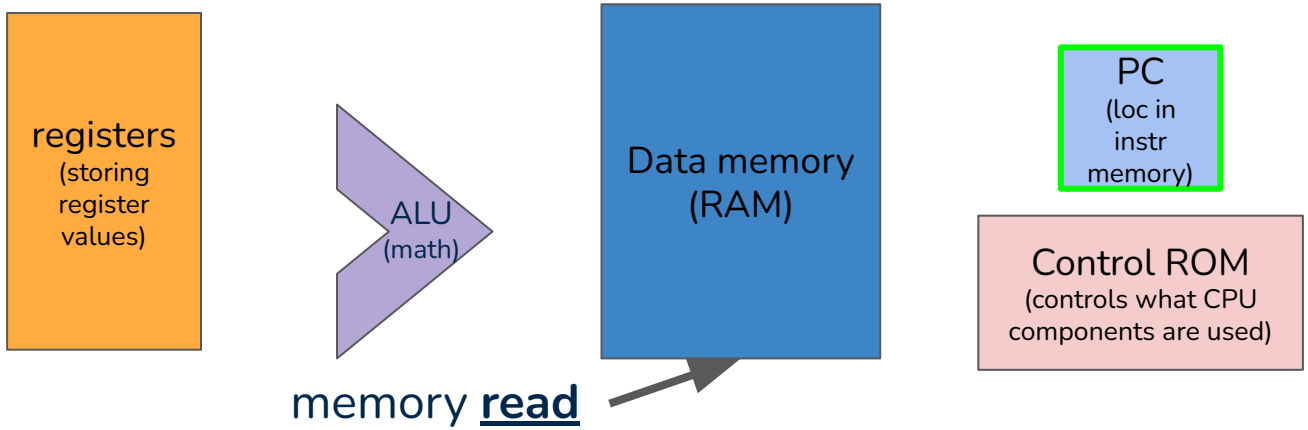


The main problem with a multi-cycle processor

- Multi-cycle processors break instructions down into individual operations...
  - ...but all those instructions are only being run one at a time

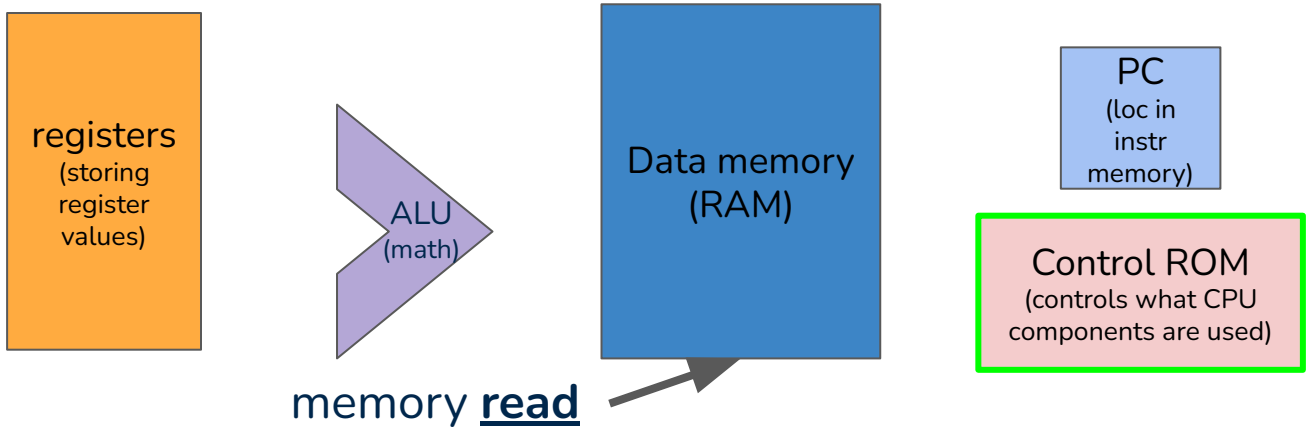
- Not every CPU component is used on every cycle of the program

Example: executing a lw



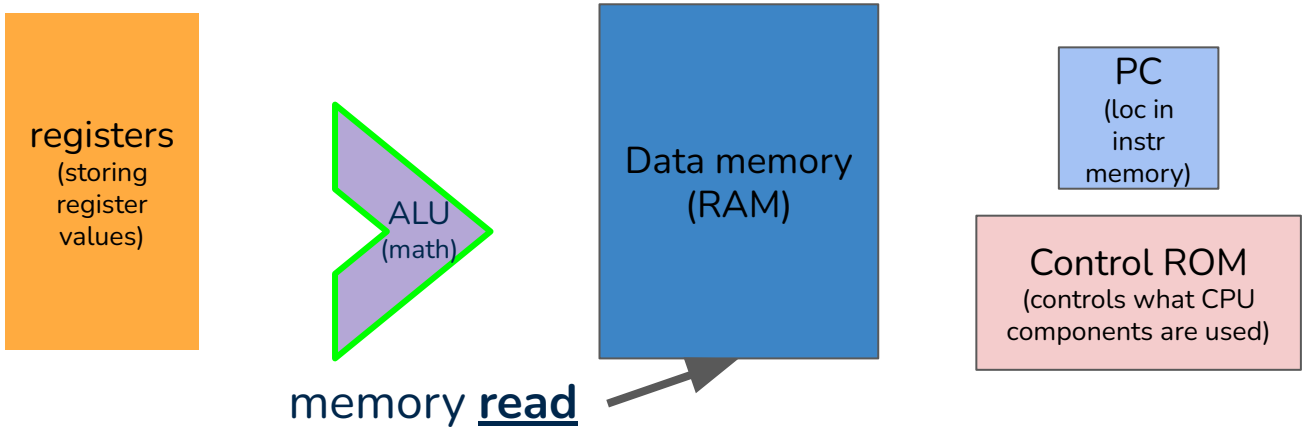
1. update PC and fetch the instruction

Example: executing a lw



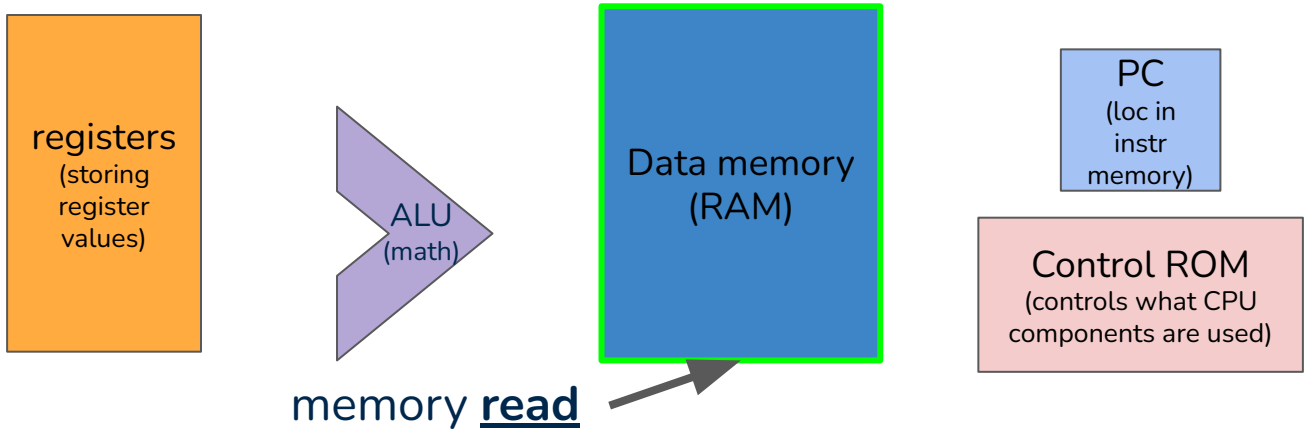
2. decode the instr so you know it's a lw

Example: executing a lw



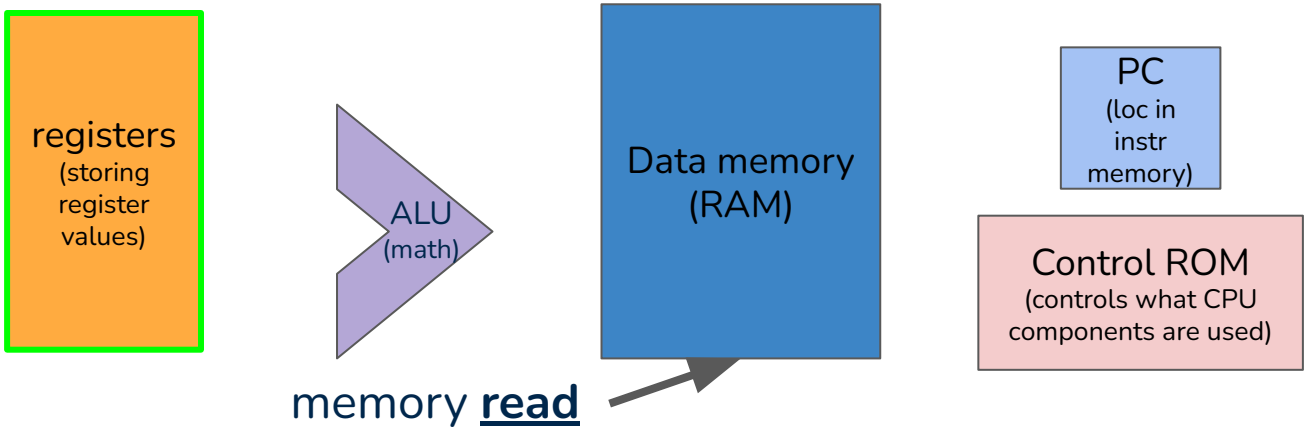
3. calculate memaddr

Example: executing a lw



4. load value from memory

Example: executing a lw



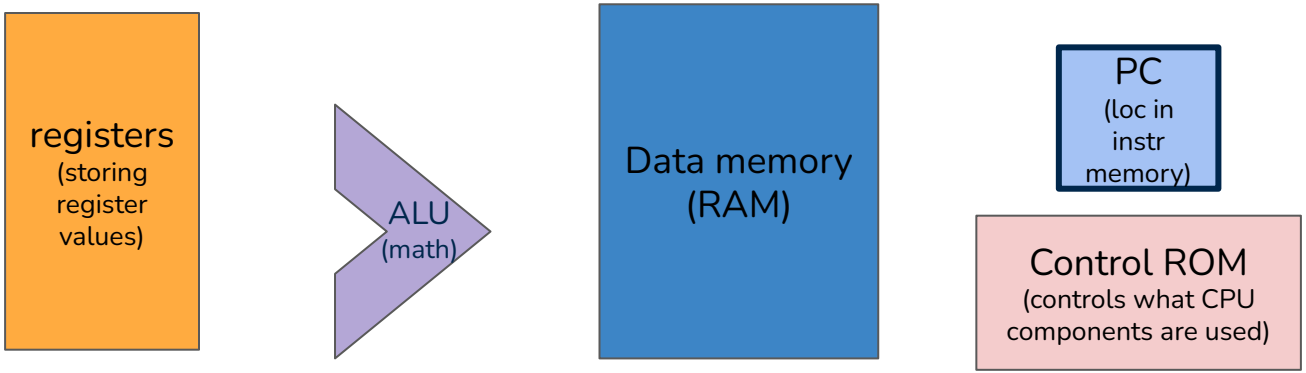
5. write value to register

Pipelined processors remove this inefficiency (mostly)

- Pipelined processors **overlap** instructions
- Every part of the CPU is used on every* cycle!

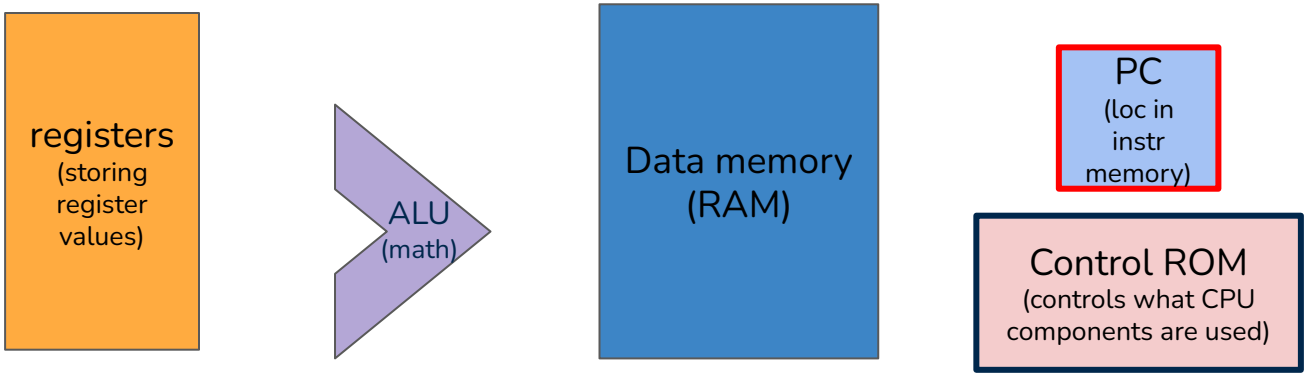
*except the “warmup” and “cooldown” cycles

Running a large benchmark with a pipelined processor



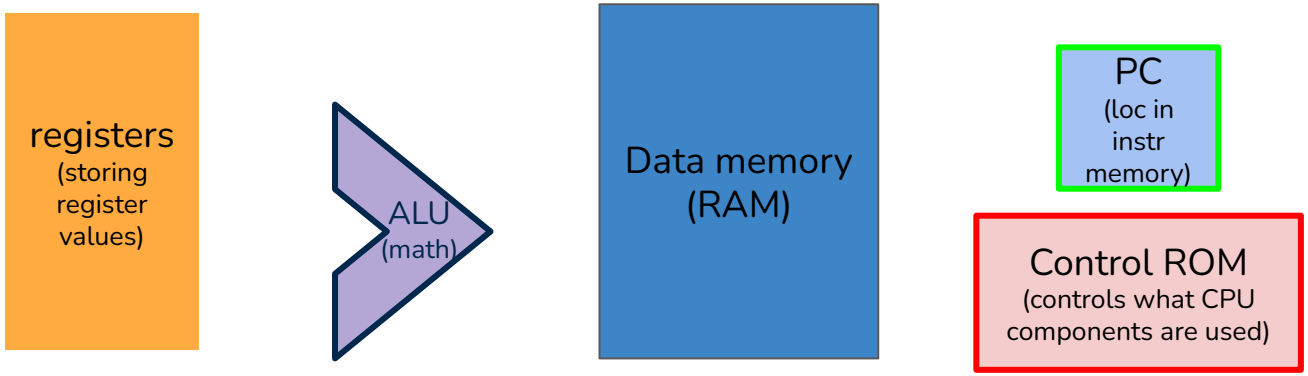
1. update PC and fetch the instruction

Running a large benchmark with a pipelined processor



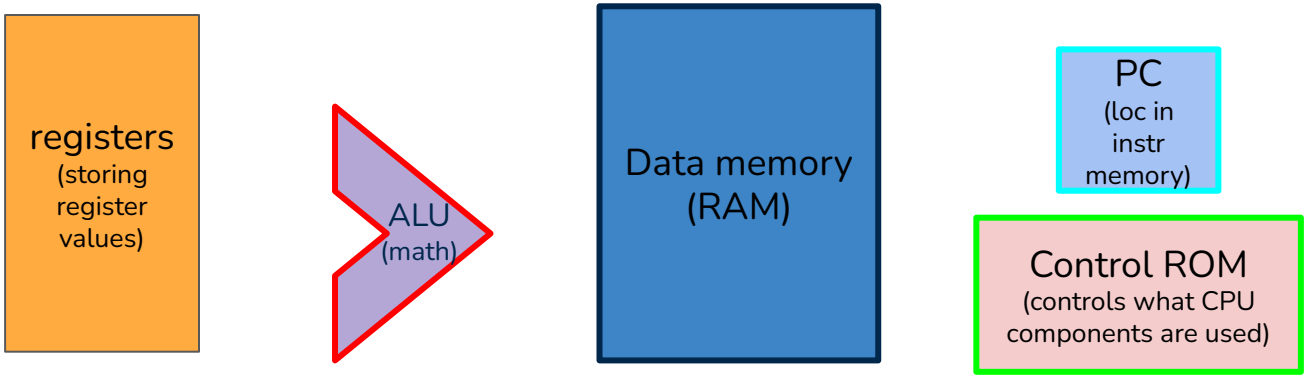
1. update PC and fetch the instruction
2. decode the instr so you know it's whatever instr it is

Running a large benchmark with a pipelined processor



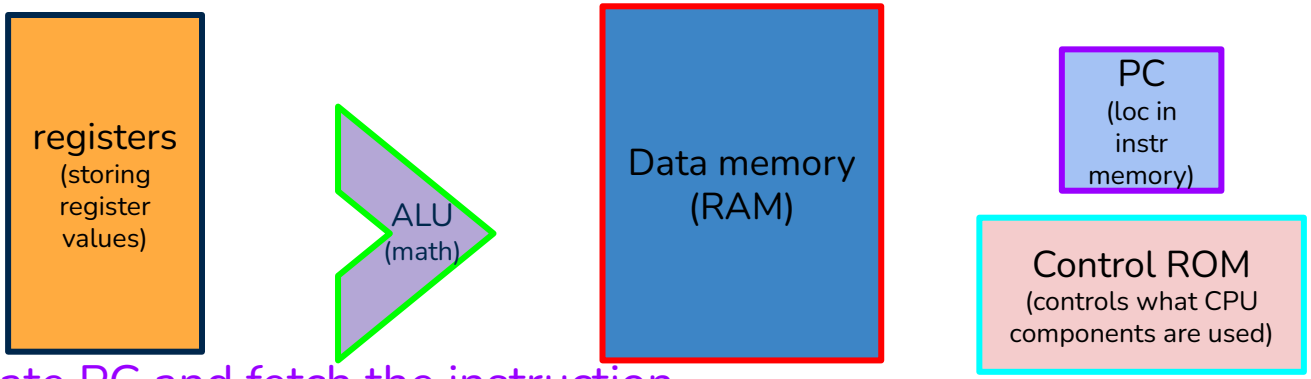
1. update PC and fetch the instruction
2. decode the instr so you know it's whatever instr it is
3. do whatever you need to do with ALU

Running a large benchmark with a pipelined processor



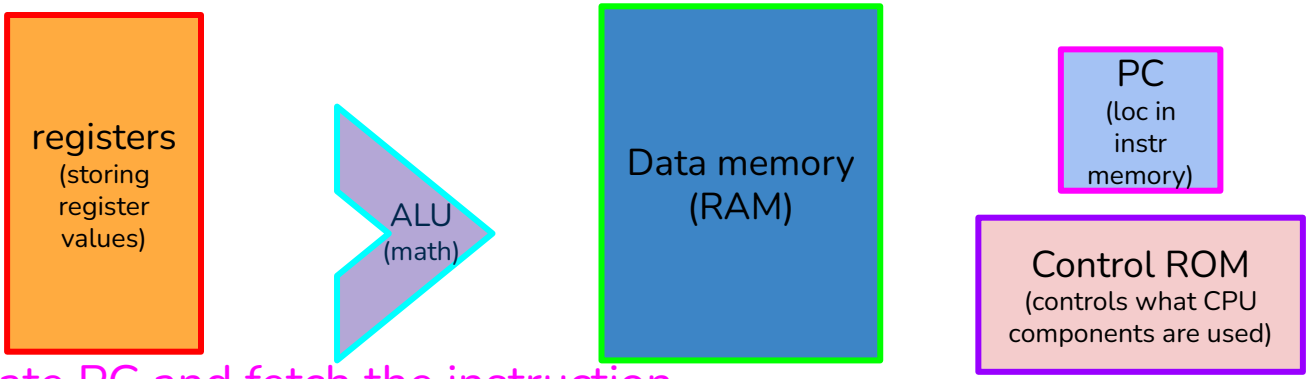
1. update PC and fetch the instruction
2. decode the instr so you know it's whatever instr it is
3. do whatever you need to do with ALU
4. read or write value to / from memory

Running a large benchmark with a pipelined processor



1. update PC and fetch the instruction
2. decode the instr so you know it's whatever instr it is
3. do whatever you need to do with ALU
4. read or write value to / from memory
5. write any values that need to be written to a register

Running a large benchmark with a pipelined processor



1. update PC and fetch the instruction
2. decode the instr so you know it's whatever instr it is
3. do whatever you need to do with ALU
4. read or write value to / from memory
5. write any values that need to be written to a register

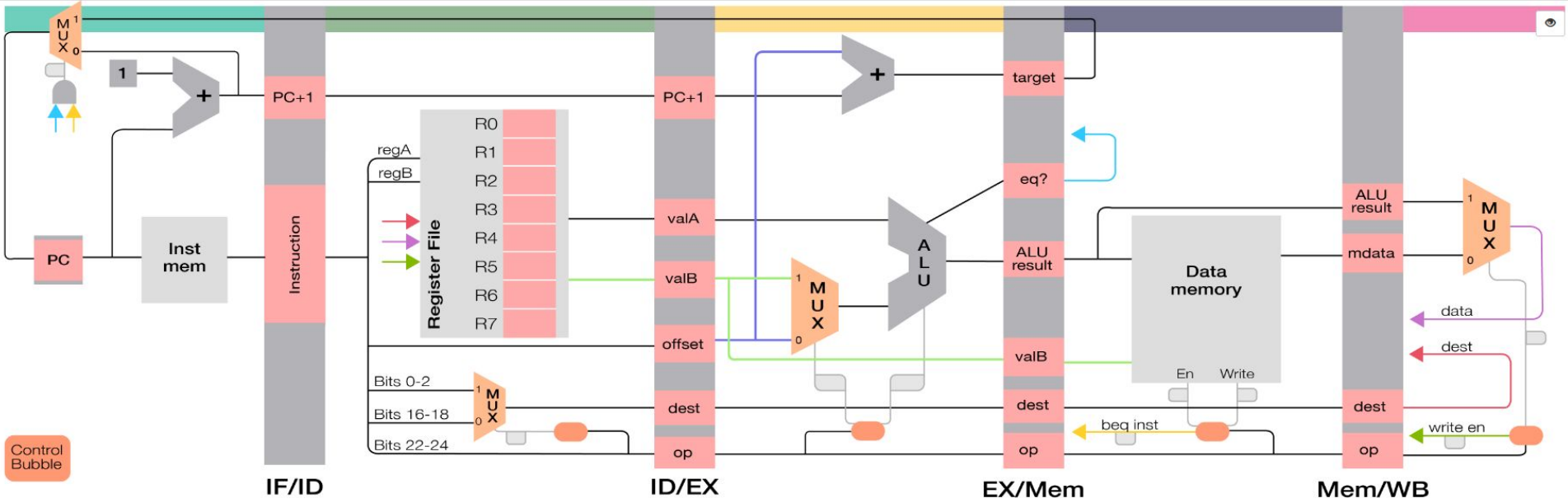
...this keeps going until we run out of instructions

Architecture of a pipeline

- Operations are broken down into stages
  - CPU components are used by only one stage
- Instructions pass from one stage to the next until completed

- There are additional components needed to make sure everything works – we will focus only on the important parts

Architecture from EECS 370 pipeline simulator



https://eecs370.github.io/simulators/pipeline/

Pipelines are almost always faster than multi-cycle processors

- This is because the instructions are overlaid and every component is used all the time
- We can compare cycles per instruction (CPI): num cycles / num instructions
  - For multi-cycle processors, most cycles do not complete an instruction
  - For pipelined processors, one instruction completes every cycle*
- We can be more formal, but intuitively pipelining will reduce the num cycles and therefore decrease CPI
  - And for both, cycle time should be the same (latency of slowest component)

*except the “warmup” and “cooldown” cycles

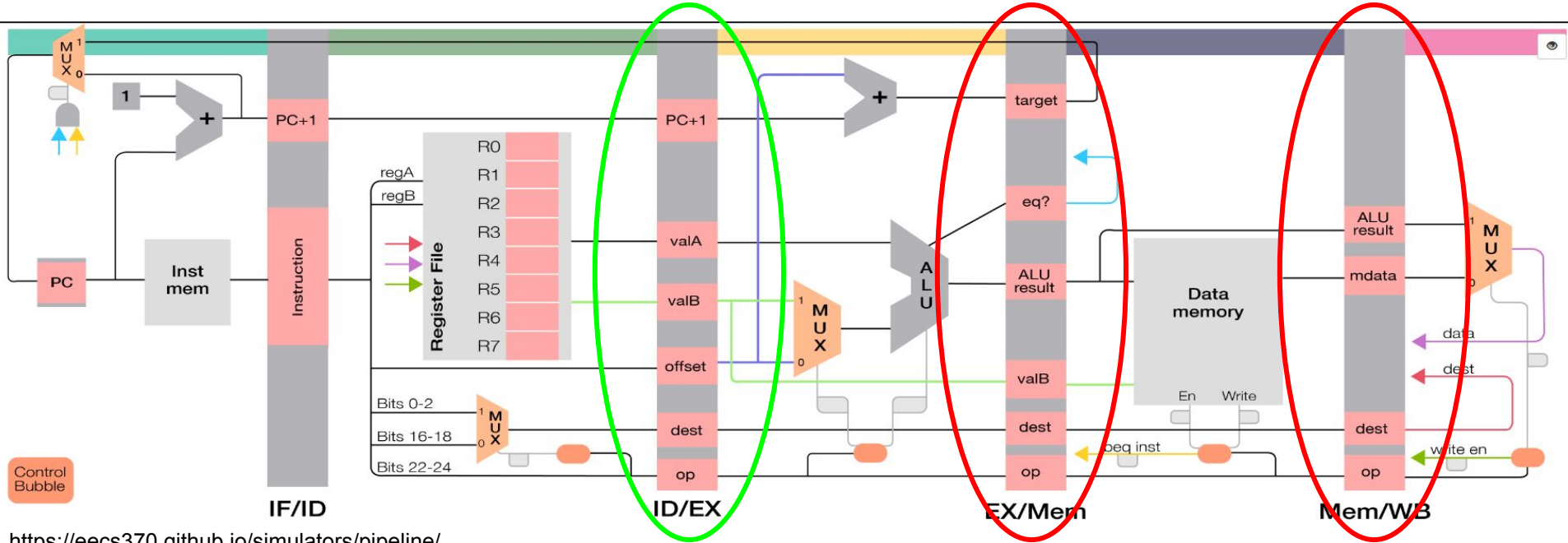
Does this all come for free?

Is it this easy?

Are there no problems whatsoever?

https://i.ytimg.com/vi/sYekLbgY080/maxresdefault.jpg

Data hazards: instructions using the wrong values in execute stage



https://eecs370.github.io/simulators/pipeline/

Data hazards can be resolved by detect-and-stall or detect-and-forward

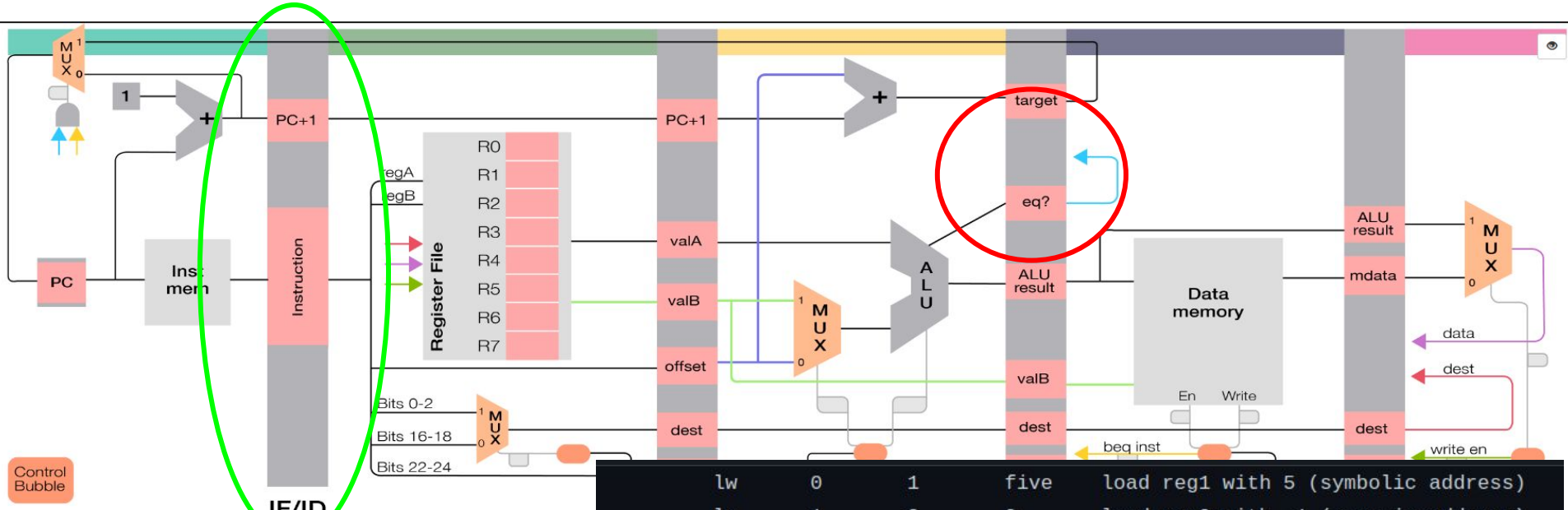
- If we have time, we'll discuss these on the board

Control hazards: executing the wrong instructions

- Pipeline assumes that all instructions execute in order
- But after a branch, you might need different instructions

```
        lw      0       1       five    load reg1 with 5 (symbolic address)
        lw      1       2       3       load reg2 with -1 (numeric address)
start   add     1       2       1       decrement reg1
        beq     0       1       2       goto end of program when reg1==0
        beq     0       0       start   go back to the beginning of the loop
        noop
done    halt                            end of program
```

Result of execute stage may mean you need to run different instructions



https://eecs370.github.io/simulators/pipeline/

```
        lw      0       1       five    load reg1 with 5 (symbolic address)
        lw      1       2       3       load reg2 with -1 (numeric address)
start   add     1       2       1       decrement reg1
        beq     0       1       2       goto end of program when reg1==0
        beq     0       0       start   go back to the beginning of the loop
        noop
done    halt                            end of program
```

UNIVERSITY OF MICHIGAN

Control hazards can be resolved by stalling or speculate-and-squash

- If time, we will discuss on the board

Countdown program in LC2K assembly

- load three numbers from memory into registers
- in loop:
  - add -1 to value in r2
  - branch to “done” if value in r2 == 0
- halt

```
        lw 0 1 neg1
        lw 0 2 ten
        lw 0 3 one
loop    add 2 1 2
        beq 2 0 done
        beq 0 0 loop
done    halt
neg1    .fill -1
ten     .fill 10
one     .fill 1
```

Pipeline simulator

https://vhosts.eecs.umich.edu/370simulators/pipeline/simulator.html

(designed by EECS 370 course staff)

You can try inputting this example into the simulator

```
        lw      0       1       five    load reg1 with 5 (symbolic address)
        lw      1       2       3       load reg2 with -1 (numeric address)
start   add     1       2       1       decrement reg1
        beq     0       1       2       goto end of program when reg1==0
        beq     0       0       start   go back to the beginning of the loop
        noop
done    halt                            end of program
five    .fill   5
neg1    .fill   -1
stAddr  .fill   start                   will contain the address of start (2)
```

Thank you!

Please ask questions about anything you are curious about.

kwang@al.edu.pl