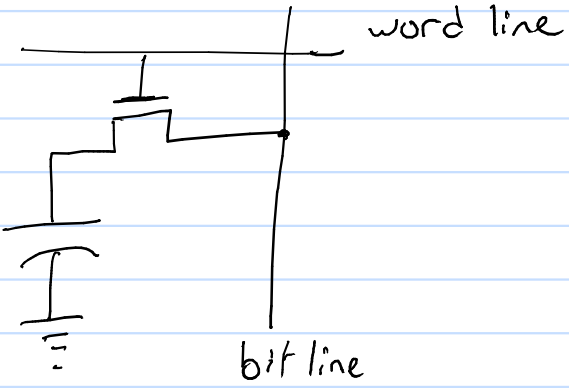
Dynamic RAM Cell



-This DRAM cell stores one bit in a capacitor.

To Read:
1) wordline ⇐ 1
2) bitline ⇐ Hi Z (empty wire)
3) The transistor turns on and capacitor charges or discharges the bitline
4) The bit is rewritten as described below.

To Write:
1) wordline ⇐ 1
2) bitline ⇐ input
3) The transistor turns on and the bitline is held until capacitor is charged
4) wordline ⇐ 0

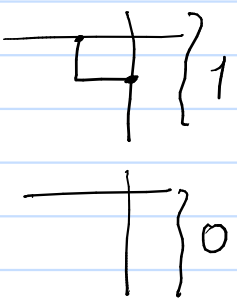
ROM - Read Only Memory

→ Main purpose is reading
- ROMs are read many more times than they're written to

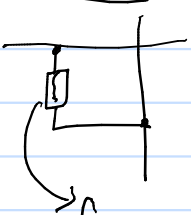
Type	Write?
ROM	During fabrication
PROM	Once by user by burning fuses
EPROM	Multiple times by erasing with UV light
EEPROM	Can be erased and written in circuits
Flash EEPROM	Can be erased and written in circuits. Writes can be done many times
NAND Flash	Faster, denser flash memory

→ SSDs are this type

ROM



PROM



fuse:
active = 1
blown = 0

Building a Single Cycle CPU Step-by-Step

Fetch/Execute Cycle

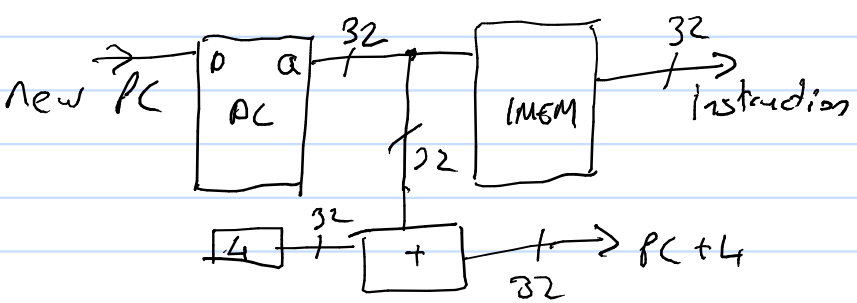
1) Fetch instruction
2) Increment PC
3) Decode instruction
4) Execute

First idea:

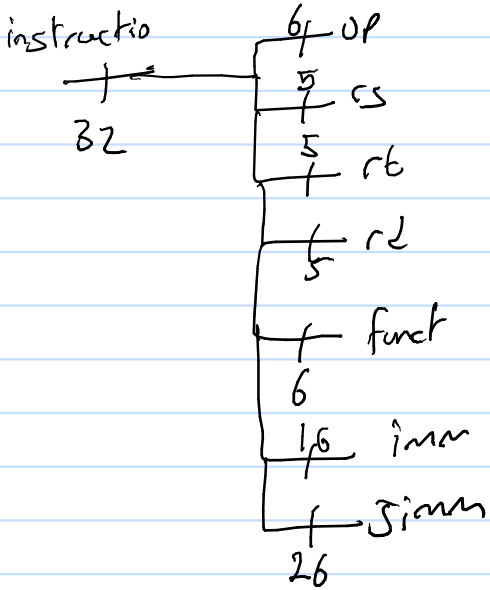
- Having a single memory unit is a structural hazard. We can fix this by adding separate instruction and data memory.

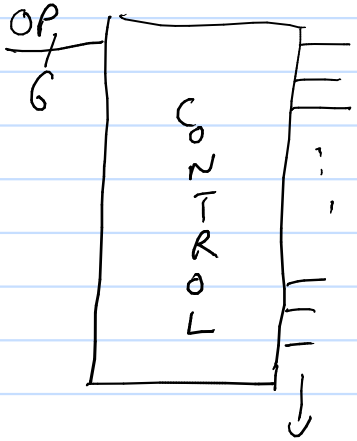
FETCH

1) Access instruction memory
2) Increment PC
3) New PC is determined after instruction is executed (due to jumps and branches)



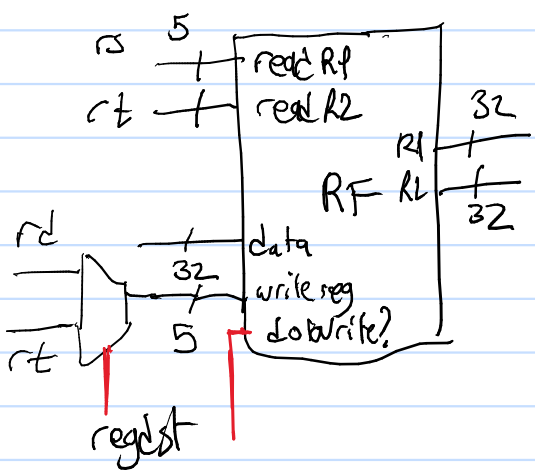
UNBUNDLE & DECODE





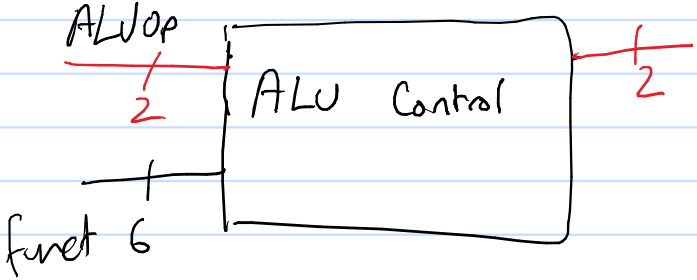
all the control signals required (shown in RED)

REGISTER FILE

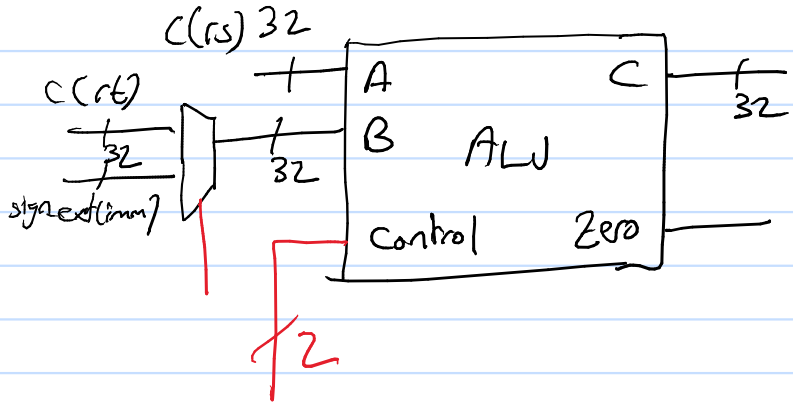


- We read two registers from the register file
- We may use both, only one, or none based on the instruction.
- We want to write to the RF on certain instructions

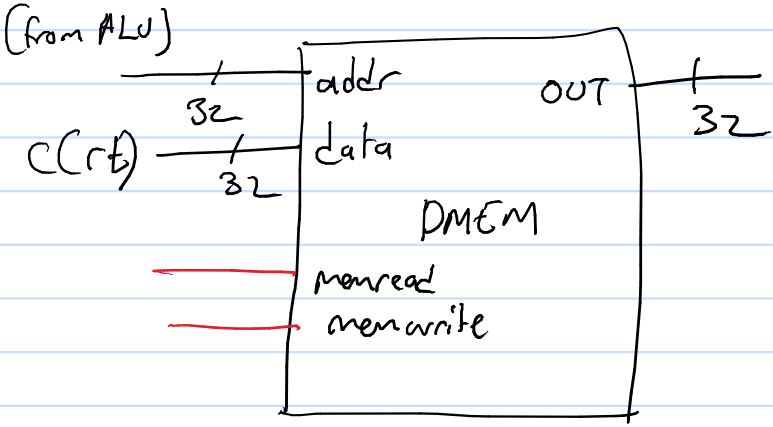
ALU and ALU Control

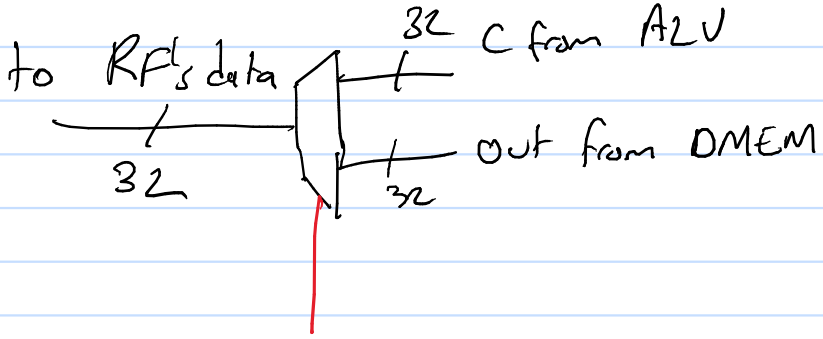


Our ALU does add, sub, or, and, slt

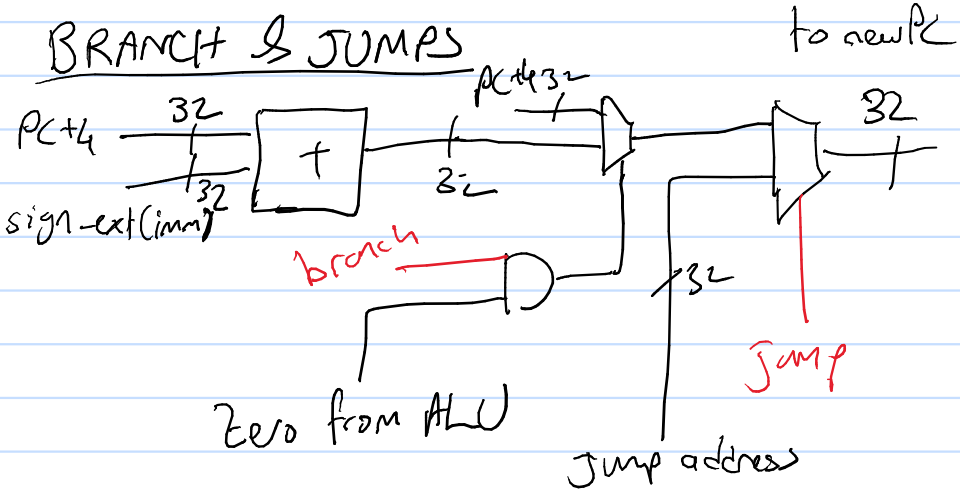


DATA MEMORY





BRANCH & JUMPS



Determining Control Signals

Please practice on the OneCycle sheets found on Canvas for all instructions. Here's what to keep in mind:

- Anything that changes the processor state (Registers, PC, DMEM) can't be a don't care
- Otherwise, decide signals based on what sources and results the instruction needs according to the MIPS sheet.
- For instance, a lw clearly needs an output from DMEM, an add needs an output from the ALU and a beq shouldn't change Registers or DMEM at all.
- Note that if you set a structure like registers and DMEM to NOT write anything with control signals, it doesn't matter what their data inputs are.

Calculating Timings

- Please practice on OneCycle sheets.

- For add, sub, and, or, slt instructions:
  - Start from PC, get the instruction from IMEM then calculate all the way to the write data into the register file.
  You also need to calculate the path from PC → PC. The longest of these is your cycle time.

- For beq, calculate until PC → PC is determined. Note that for PC to be determined, we need the Zero output from the ALU.

- For j, calculate until PC → PC

- For lw, calculate the path from PC → PC and from RF → RF going through ALU and DMEM.

- For sw, calculate the path from PC → PC and from RF → DMEM going through ALU.

PERFORMANCE

- Performance of a processor is determined by how much time a certain workload takes on different processors.

$$\text{time} = \#\text{insts} \times \underset{\left(\frac{s}{\text{cycle}}\right)}{\text{cycle time}} \times \underset{\left(\frac{\text{cycle}}{\text{insts}}\right)}{\text{CPI}}$$

Frequency of a circuit is 1/cycle time

$$1\,Hz = \frac{1}{\text{second}}$$

Ex:

500ns cycle time, frequency = ?

$$\frac{1}{500ns} = \frac{1}{5 \times 10^{2} \times 10^{-9} s} = 0.2 \times 10^{+7}\,Hz$$

$$= 2 \times 10^{6}\,Hz$$

$$= 2\,GHz$$

Improving Performance

- We can't change our benchmark program.
- We can reduce our cycle time
- We can reduce our CPI

Improving Cycle Time

- There are usually tradeoffs to doing an improvement.
  - Either circuit size increases, or CPI increases based on how we're speeding up.
- We can make improvements by optimizing our circuits.
- Since our cycle time is based on the instruction with the longest propagation delay, we must improve that to improve our cycle time at all.

Pipelining

Idea: Make use of idle circuit parts to work on the next instruction.

- Intuitively reduces cycle time if each pipeline stage does a smaller piece of work.
- However, fetching and executing instructions ahead of time can cause several hazards.

For these hazards, we either need preventative action or corrective action.

To prevent:
- We can stall :) If we stall at every possible problem point, we won't have any issues

To correct:
- If an incorrect instruction is fetched, squash
- If incorrect data is being used, forward appropriate data from a later stage

Forwarding:

- Do not violate causality while forwarding: You can't forward data back in time!

- You need a stage's circuits to do work with the forwarded data, so the only sensible thing is to forward data from the beginning of a cycle to the end of a cycle.

Squashing

- To squash an instruction, set the stage to contain an instruction that doesn't affect the state

Hazards, Detection and Correction

3 types of main hazards:

1) Control Hazard

- Happens when we go too far with fetching instructions after changing control flow.

Ex: 3 stage pipeline,

IF EX MEM

↳ branches resolved here

```
beq  $3,$4, label   ($3 == $4)
add  $4,$5,$2  ──┐
                 ↓
label: sub $4,$5,$3   this will be incorrectly fetched
```

Correction:
- SQUash

Prevention:
- Stall until branch is resolved

2) Data Hazard

- Happens when an instruction in the pipeline requires the result of a prior instruction but that result has not been committed to state yet

Ex: 5-stage pipeline,


```
add $4, $3, $2
sub $5, $4, $2
```

sub needs $4 at ID, but it hasn't been written back yet. Note that we don't have to wait at ID for forwarding. Knowing forwarding is available lets us proceed because we know the correct value is going to show up.

Correction: Forwarding & Stalls
Prevention: Stall

- Note that it may not always be possible to forward without stalling.

Ex:

```
lw $4, 0($5)
add $7, $4, $3
```

causes a stall in any pipeline where MEM completes after EX

3) Structural hazard

- Happens when a single resource is required by multiple sources.

Happened with memory earlier, we fixed it by splitting it into IMEM and DMEM. So, ...

Correction: Stall
Prevention: Add more of the same resource

Performance in pipelines:

Any stall or squash introduces a bubble in the pipeline. This affects the CPI.

Example.

In a pipelined processor where.
a) taken branches cause 2 bubbles
b) data dependencies after lw cause 1 bubble
c) jumps cause 1 bubble

taken branches => 10% of dynamic code
jumps => 5% of dynamic code
data dep. after lw => 10% of dynamic code

$$CPI = 3 \times 0.1 + 2 \times 0.05 + 2 \times 0.1 + 1 \times 0.75$$

- 3×0.1: taken branches; 1 cycle for ex, 2 bubbles, 10% of the time
- 2×0.05: jumps, 1 cycle for ex, 1 bubble, 5% of the time
- 2×0.1: data deps, 1 cycle ex, 1 bubble, 10% of the time
- 1×0.75: normal

Startup cost of pipelines

- If pipeline has n stages, the first instruction is finished at cycle n-1. This startup penalty can be ignored most of the time because it's only incurred once and is negligable after running many instructions.

Typical MIPS Pipeline

1) Instruction Fetch - IF
2) Instruction Decode - ID → reg. read, jumps
3) Execute - EX → branches
4) Memory - MEM → mem. instr
5) Writeback - WB

Important

- Go over practice problems on Canvas.