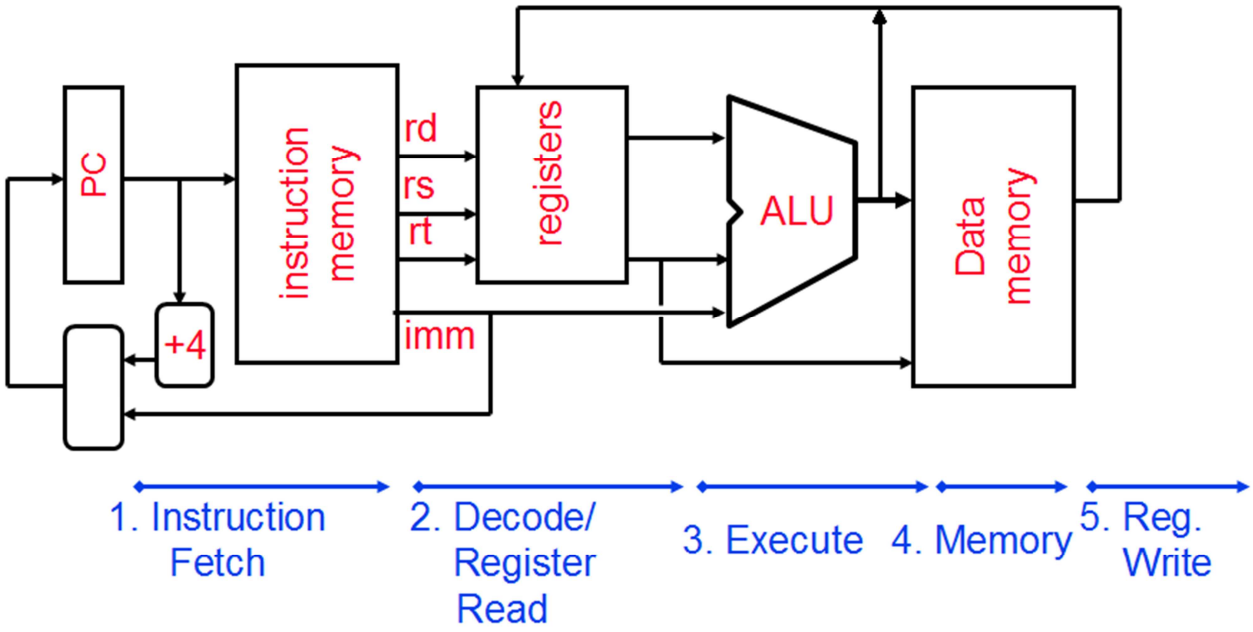
CPU Design

Here is the basic datapath as discussed in lecture, shown in simplified (i.e. incomplete) format.



`rd`, `rs`, and `rt` are 5-bit wires, `imm` is a 16-bit wire. All other wires are 32 bits wide.

Register Transfer Language (RTL)

- Use to describe flow of data: `dest ← src`
- Each line happens in parallel (at the same time): `b ← c, a ← b`
- In MIPS, use `R[x]` for register `x`, and `Mem[y]` for memory at `y`. Similar to array syntax.

Exercises

Assume that the ALU can output an Equals signal, which is on when its two inputs are equal.

1. Label the unlabeled wires in the diagram above to describe what data is on each line. For example, one of the outputs of the registers block could be `R[rs]`.

Signals are labeled in the figure on the next page:

PC block: Input is the next PC, output is the current PC

Instr Mem: Outputs rd, rs, rt, and imm were given. Wire that comes off of imm and back towards the PC block is the branch address.

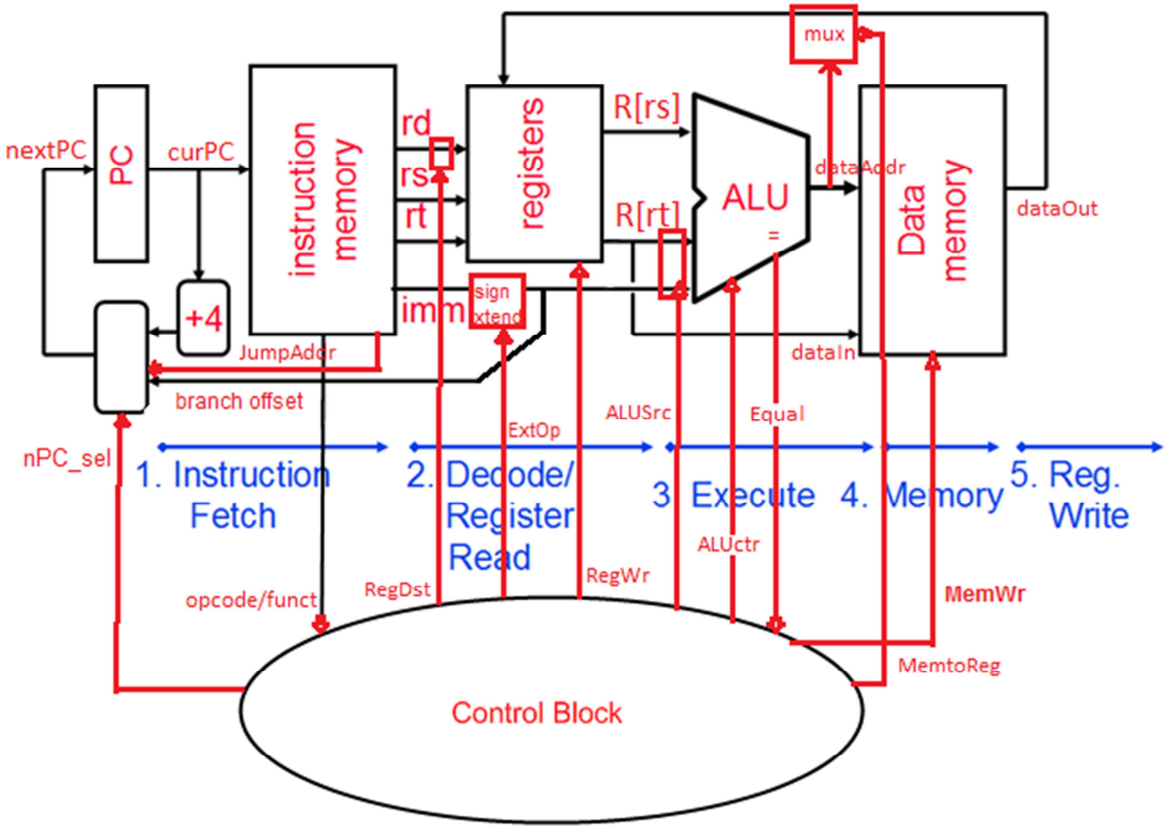
Reg File: Outputs are `R[rs]` on top and `R[rt]` on bottom. Order does matter (why?)!

Data Mem: Inputs from ALU (address) and Reg File (data), and output is just output data.

ALU: Basic signals already covered by other blocks.

2. Add control signals and missing elements (such as multiplexers) to the diagram below so that the datapath can execute the following instructions: `add`, `lui`, `sw`, `bne`, `j`.

See the figure on the next page. More explanation can be found below the figure.



I realize this is a lot to take in at once, so let's break it down by instruction:

j: Need to get the 26-bit target address from the instruction, hence we add a new line **`JumpAddr`** from Instr Mem to the "address logic" module that computes the `nextPC` (rounded rectangle in lower-left). In addition, we need a way for the address logic module to choose between a step, a branch, and a jump, so we add the control signal **`nPC_sel`** to MUX them. A jump instruction only updates the PC, so we actually don't want to write to the Register File or Data Mem. We add control signals **`RegWr`** and **`MemWr`** to allow us to designate when we want to write or not.

`add`: Two registers are read and added, then the result is stored back into another register. The Register File can already output the values of the correct two registers (`R[rs]` and `R[rt]`) and the ALU can do addition, so now we need to make sure we can store the result back into a register. First we add the control signal **`ALUctr`** to select the proper ALU operation. Currently, the register input is `dataOut`, but the result we want is the output of the ALU. So we add a MUX on the register input line that selects between `dataOut` and `ALUOut` using the new control signal **`MemtoReg`**.

`lui`: Shift the `imm` left by 16 bits, then store into register. The shifting can be done by the ALU, but it needs the `imm` instead of `R[rt]`, so we MUX those signals together and use the control signal **`ALUSrc`** to pick between them. The result of the ALU is stored back into a register. Unlike `add`, though, I-format instructions store into `$rt` instead of `$rd`. So we add a MUX for the `rd` input to the Reg File that selects between `rd` and `rt` based on the new control signal **`RegDst`**.

sw: Store `R[rt]` into Data Mem at address `R[rs] + SignExtImm`. `R[rt]` is already set up as `dataIn` to Data Mem, so we just need to sign extend the immediate and let the ALU do the addition. We add **a sign extend block** on the `imm` line (the **`ExtOp`** signal is only necessary if we also have an instruction that requires ZeroExtImm, such as ori, but is not necessary here).

bne: Branch if `R[rs] != R[rt]`. We assume the proper `nextPC` is calculated from the branch offset in the address logic module. The ALU performs the comparison of `R[rs]` and `R[rt]` and we use the signal **`Equal`** as an *input* to the control logic, which will help determine the value of `nPC_sel`.

3. Fill out the values for the control signals from part 2 (write the names of your control signals in the second row):

Instr	Control Signals							
	nPC_sel	**RegDst**	**RegWr**	**ALUSrc**	**ALUctr**	**MemtoReg**	**MemWr**	**ExtOp**
add	PC+4	rd	1	rt	add	0	0	X
lui	PC+4	rt	1	imm	shiftx16	0	0	Zero*
sw	PC+4	X	0	imm	add	X	1	Sign
bne	branch	X	0	rt	X	X	0	Sign**
j	jump	X	0	X	X	X	0	X

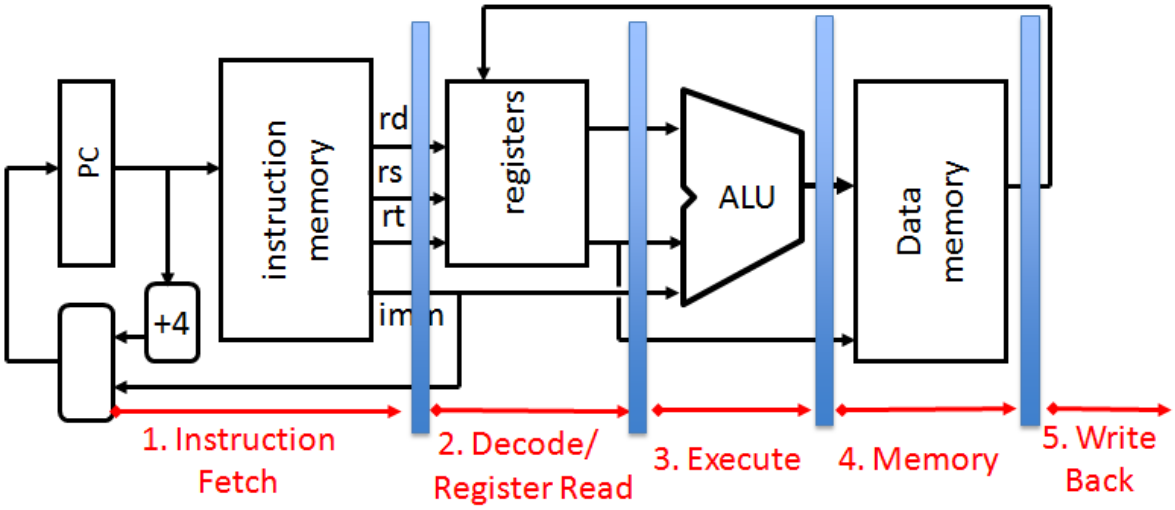
* Doesn't really matter since `ALUctr` is shiftx16, so could set to `Sign` and eliminate `ExtOp`
** Branch also multiplies immediate by 4 at some point

4. Suppose you wanted to add a new instruction, `beqr`, which will be used like this: `beqr $x, $y, $z` will branch to the address in `$z` if `$x` and `$y` are equal, otherwise continue to the next instruction. Show any changes that would need to be made to the datapath above to make this instruction work.

The diagram is already crowded, so I will just explain the changes. Since we're reading three registers at once here, we need to add a third read port to the register file, and correspondingly, a third read address port. We can reuse `beq`'s datapath to compare two registers, `rs` and `rt`, so `rd` would contain the address to jump to. We would then connect `R[rd]` to the address logic module.

The Pipelined Datapath

Here is the same datapath that you've seen before but pipelined. Note that this rough diagram is not entirely correct: signals rd, rs, rt, and imm are not produced until the Decode stage. A more detailed diagram is in the textbook and in lecture slides.



What about the CPU Control: What would happen if we didn't change it and still wired each control signal directly to its destination? How do we fix this problem?

The control signals for one instruction will be used for a different instruction! Control signals need to be pipelined to the stage they are needed in. For example, RegWrite would be pipelined from the Decode stage all the way to the Write-Back stage. This will make it so that each pipeline stage will always have the control signal values that correspond to the instruction currently in that pipeline stage.

Pipelining Hazards:

Structural – Hazards due to competition for the same resource (register file read vs. write back, instruction fetch vs. data read).

How are these hazards dealt with in hardware?

Early Register Write, late Register Read (register timing)
Separate Instruction and Data Caches

Control – Hazards due to non-sequential instructions (jumps and branches).

How are these hazards dealt with in hardware?

Branch/Jump Delay slots (MIPS) or branch prediction

Data – Hazards due to data dependencies (instruction requires result from earlier instruction).

How are these hazards dealt with in hardware?

Forwarding (does not avoid Load-Use Data Hazards)
Out of Order Execution (hardware re-orders the sequence instructions are executed in)

Pipelining Exercises

1. Suppose you've designed a MIPS processor implementation in which the stages take the following lengths of time: IF=20ns, ID=10ns, EX=20ns, MEM=35ns, WB=10ns. What is the minimum clock period for which your processor functions properly? Where should the bulk of your R&D budget go for the next generation of processors?

The memory stage is the bottleneck, limiting the clock period to 35ns. Research should be put into speeding up the memory stage.

2. When is there never going to be 5 instructions executing at the same time in your pipelined datapath? How does this affect your performance factor increase over a non-pipelined datapath?

At the beginning, when the pipeline needs to be filled, and also at the end, when the pipeline needs to be drained. This limits your performance factor increase to always be less than 5, even in ideal situations with non-hazardous code.

3. Your friend tells you that his processor design is 10x better than yours, since it has 50 pipeline stages to your 5. Is he right? Why or why not? (This is intentionally vague)

No, for many, many reasons:
- Much higher power consumption
- More hardware required to implement, more expensive to manufacture.
- Increased complexity in implementation, more hazards
- Overhead from implementing the pipeline stages would result in <10x speedup, even at best
- Unlikely to evenly split the logic into 50 stages, also resulting in <10x speedup.
- Other technologies (for example, caches) might also limit the performance of any one stage.
- Increased penalty for missed branch predictions / longer to fill the pipeline