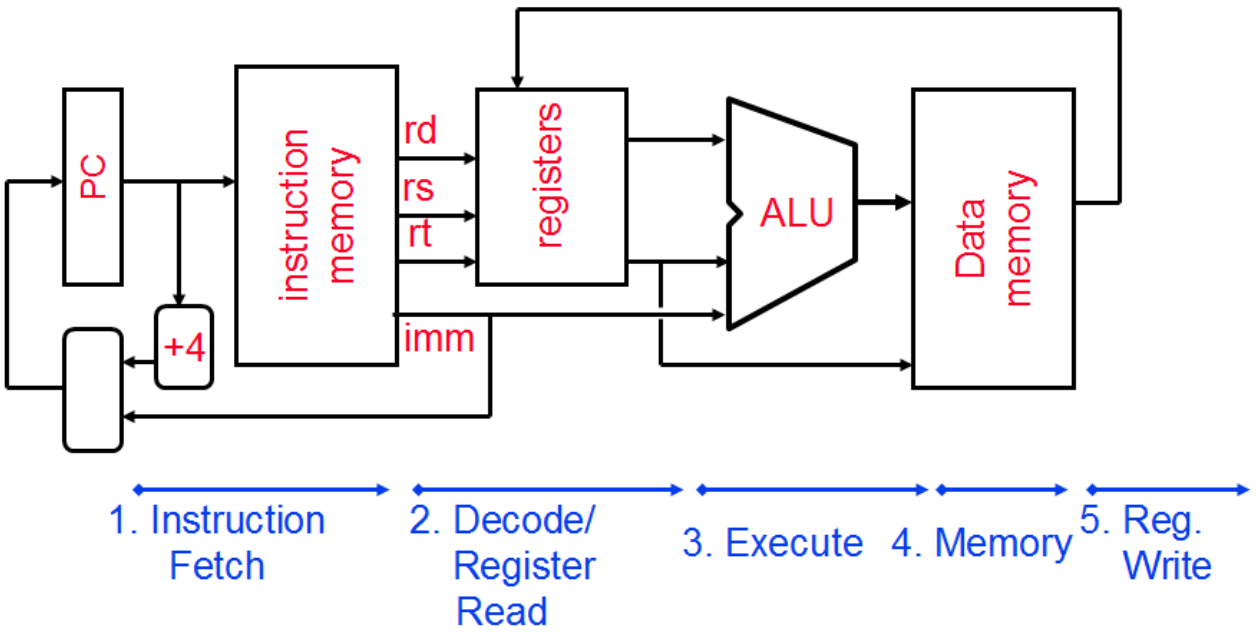
## Quick Review

- Finite state machines (FSMs) are a useful tool for solving many problems in computer science.
- An FSM can be written down in truth table format, with one row in the truth table for each transition in the FSM.
- We can use a truth table to create a Boolean expression for the output based on the inputs, and use this expression draw a circuit level diagram. We can also use Boolean algebra to simplify the expression first before drawing the diagram.

## CPU Design

Here is the basic datapath as discussed in lecture, shown in simplified 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

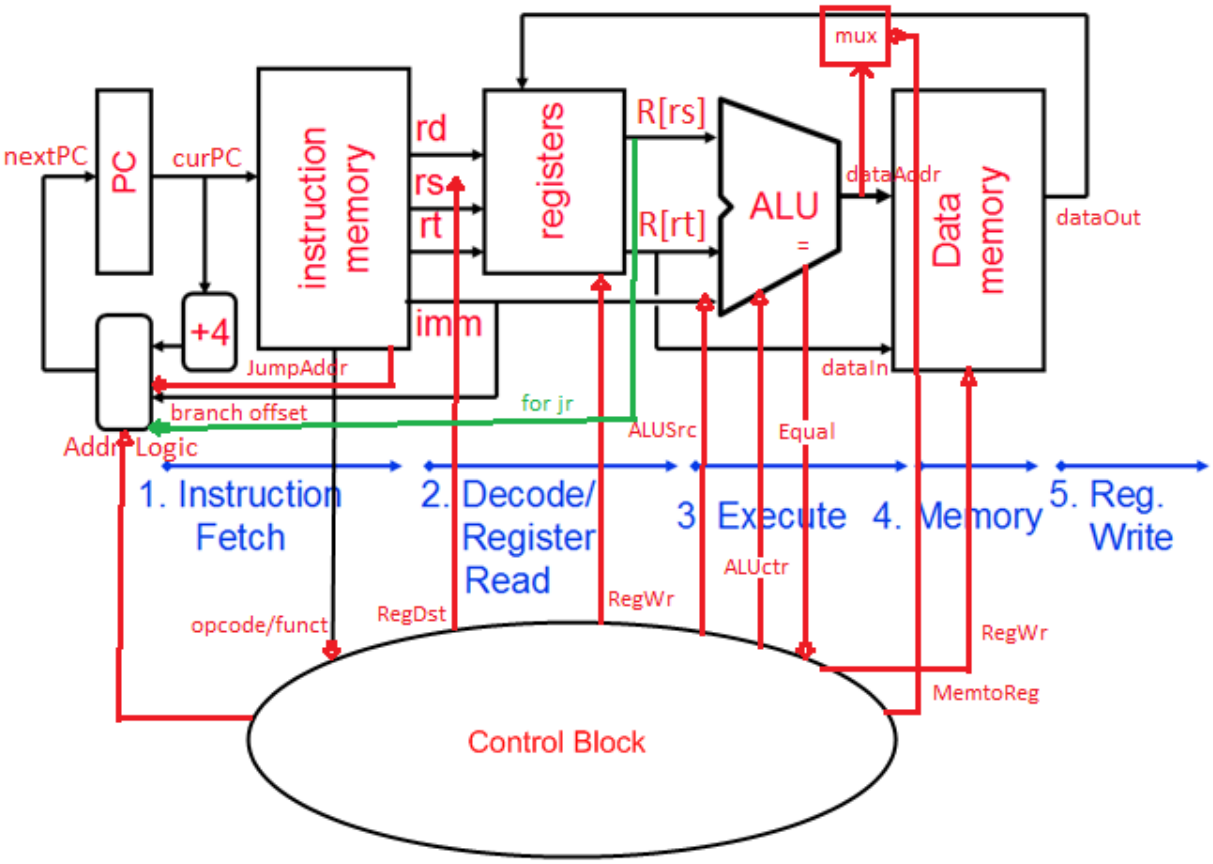
For the following exercises, assume that the ALU can output an equals signal, which is on when its two inputs are equal.

1. Add labels to unlabelled wires in the diagram above, describing what data is on the line. For example, one of the outputs of the registers block could be R[rs].

2. Add control signals to the diagram below so the datapath above could execute the following instructions: add, addi, lw, beq, and j. All control signals come out of the Control Block cloud, which takes as input the current instruction. For example, to execute add, the ALU would need an input that tells it to add its two inputs, among other things.

See lecture slides for more details.

3. Describe what happens during each stage for each of the instructions listed above.




1. Retreive the value InstMem[PC}, ie, the current instruction to execute. With clever implementation, the entirity of j can be implemented in this step (we know the jAddr after instruction fetch).
2. add/addi/lw/beq – read registers.
3. add, addi – Add registers. lw – add offset to base addr. beq- compare registers.
4. lw – read value from memory at location base addr + offset. Everything else – nothing.
5. add/addi/lw – write registers.

4. On the reprinted diagram above, show the modifications you would have to make to support jr. Show any new control signals you may have to create.

5. 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 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 (which is essentially a mux).

**Bonus Question**

You have 100 ants on a rope 100 units long. Each ant either moves left or right at 1 unit/sec. Ants take up no space (assume they are points), but whenever two ants collide they both instantly change directions. You may place the ants on the rope in any initial configuration - what is the maximum amount of time you can keep at least 1 ant on the rope?

We observe that two ants colliding has the same effect as two ants just passing through each other. Thus, any configuration of ants on the rope would allow no ant to stay up for more than 100 seconds.