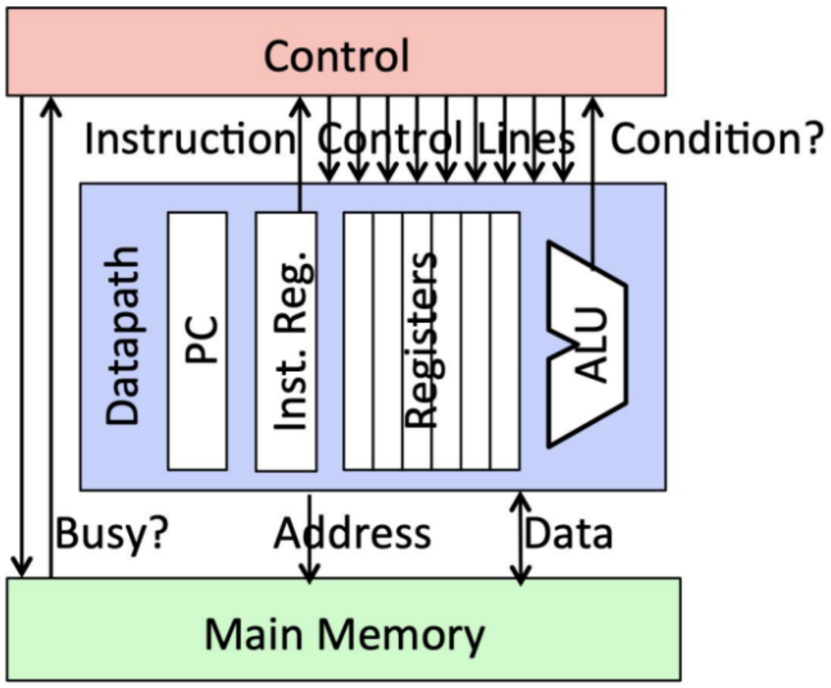
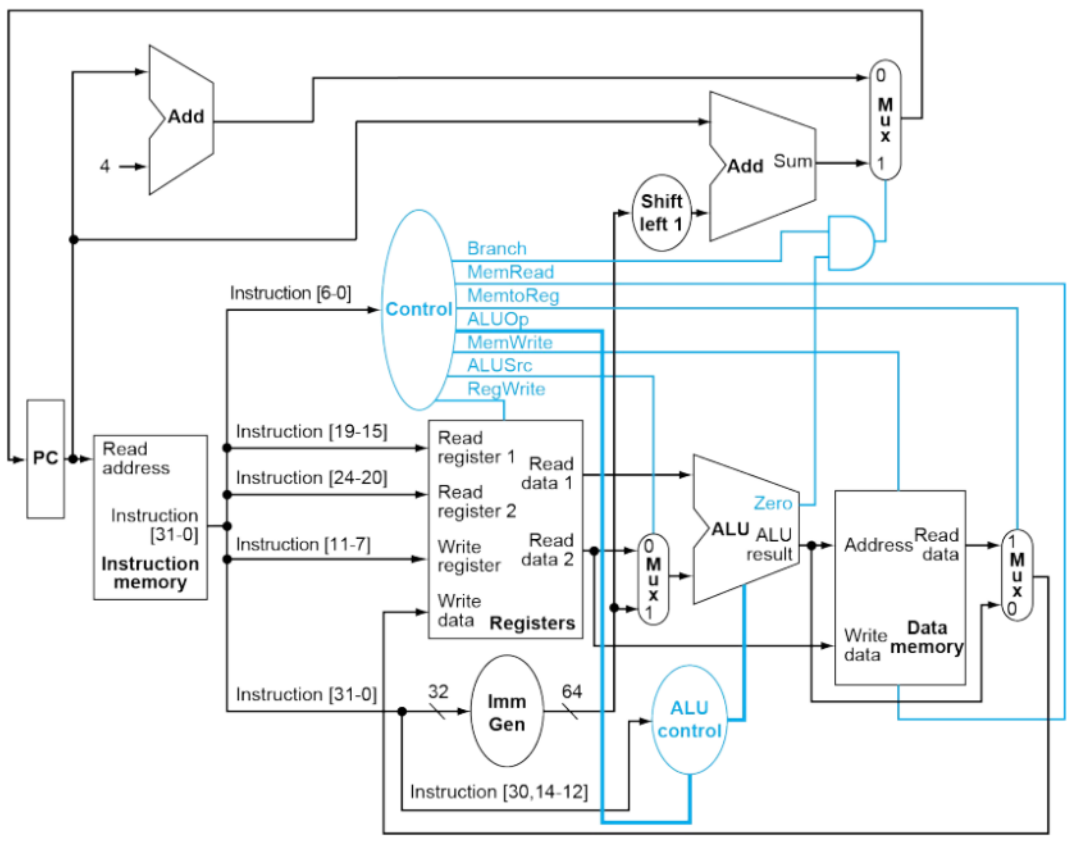
Microarchitecture

- The hardware implementation of ISA is called *microarchitecture*
- A given ISA (e.g., RISC-V) can be implemented in many different ways (i.e., same ISA, different microarchitecture)
- The goal is to build an *efficient* computer
  - Fast (high performance) and power efficient.

State-Machine View

- How do we make a microarchitecture?
- There are two different parts:
  1. Controller: a unit that directs the operation/activities on the datapath
  2. Data Path: a collection of *functional* units that process the data and create the data-flow





*Images were taken from Hennessy Patterson Book [1].

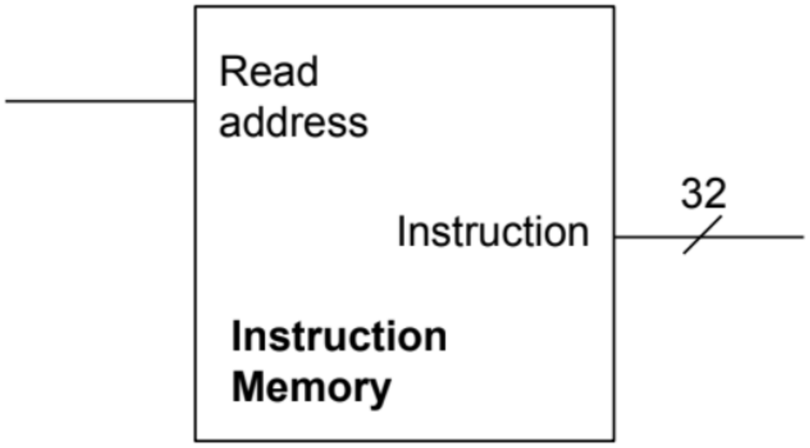
We will go through how this works, step by step. But first, how are instructions executed?

Lifecycle of an instruction

1. Instruction is read (fetch) from the (instruction) memory
2. Operands should be loaded
   - RegFile, (data) Memory, Immediates
   - Need register number / address for each operand
3. Operation should be executed
   - Arithmetic (which type?), data movement, control-flow.
4. Results should be stored
   - RegFile or memory?
5. PC should be updated
   - Usually it is PC += 4, but sometimes it is different due to jumping/branching.

Instruction Fetch

Provide a read address to the instruction, and get a 32 bit instruction.



Registers stores data in a circuit.

- Uses a clock signal to determine when to update the stored value (could be multiple bits).
- Data stored in a register is read/written on the positive edge of the clock cycle.
- Uses D-flip-flop
- Some registers have a write-enable input. If this is present, memory is only written when control input is 1.

Memory is an *array of registers*.

- Specific line can be chosen via a multiplexer.

Memory Technology

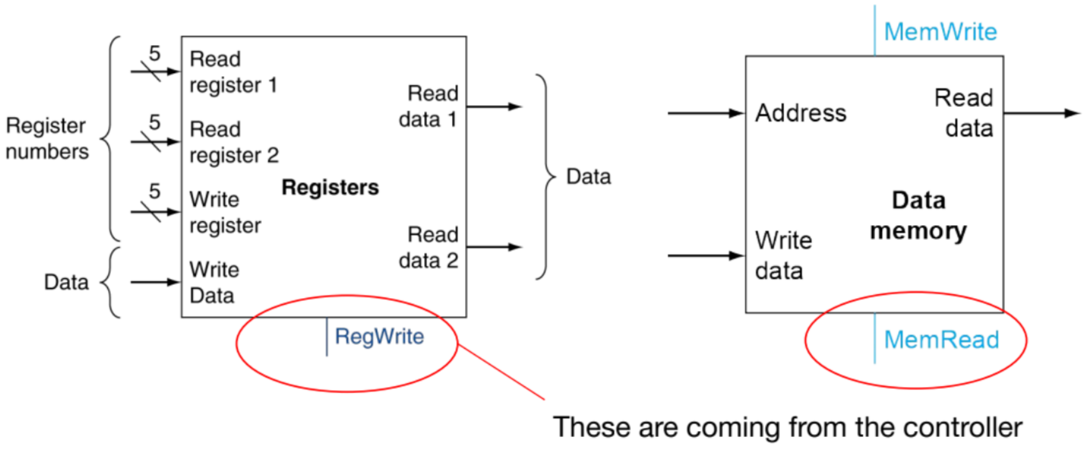
- Are all memory cells registers, SRAM, DRAM, etc.?
- No!
- **Latches and Flip-Flops (aka Registers)**
  - Very fast (1 cycle), parallel access.
  - Very expensive (one bit costs tens of transistors).
- **Static RAM (SRAM)**
  - Relatively fast (10 cycles), only one data word at a time.
  - Expensive (one bit costs 6+ transistors).
- **Dynamic RAM (DRAM)**
  - Slower (100 cycles), one data word at a time, reading *destroys* content (refresh), needs special process for manufacturing
  - Cheap (one bit costs only one transistor plus one capacitor).
- **DISK**
  - flash memory, hard disk
  - Much slower (1000+ cycles), access takes a long time
  - Non-volatile
  - Very cheap

Which type of storage should we use?

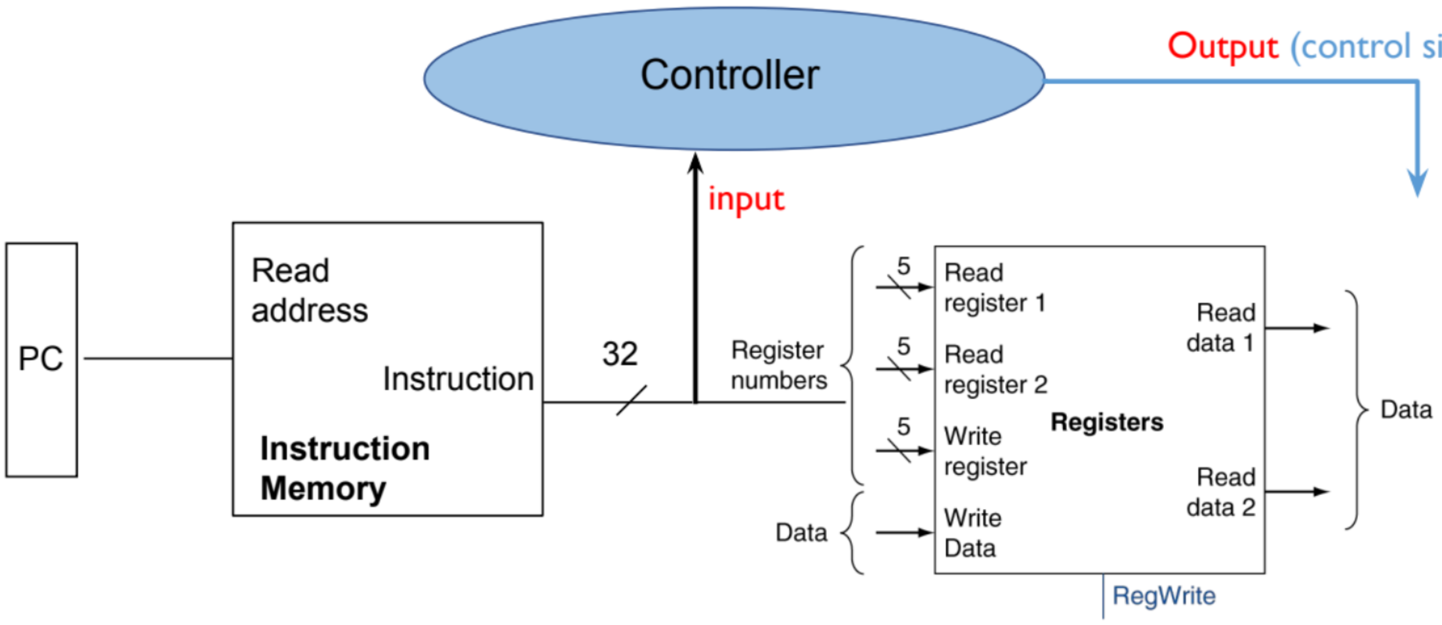
- From Register to DRAM to disk:
  - From *smaller* and *faster* → *bigger* and *slower*
  - From *more expensive* and *power hungry* → *cheaper* and *power efficient*
  - The type we need depends on the purpose we need it for. Use registers for frequently used data and slower and bigger elements for rarely-used permanent data.

Loading Operands

- To load values from registers, we provide an address to read, with the write data bit at 0.
- To write, we provide an address, the data to write, and the write enable bit.
- However, *how could we know which instruction we read?*
  - We need a controller!



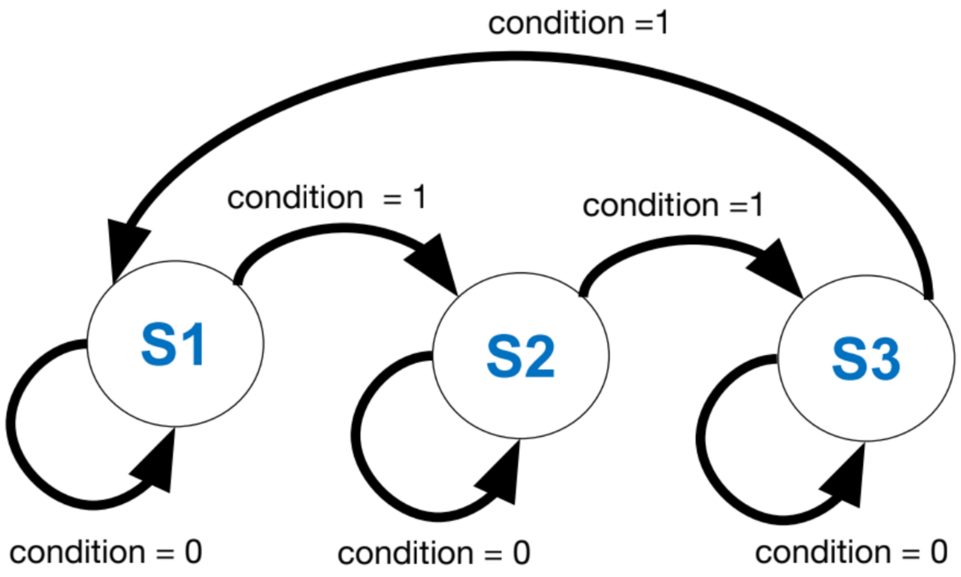
The controller directs the operations for each instruction.



We build controllers using finite state machines (FSMs).

Finite-State Machine (FSM)

- A *mathematical* model of computation
- At each point, the system can be *only at one* of the several, but **finite** states.
- FSM shows all the states and how they transit to each other.
- FSM also includes finite number of inputs and outputs.



Remember these from CS181?

The FSM/Controller for a processor is simply a giant FSM with many states!

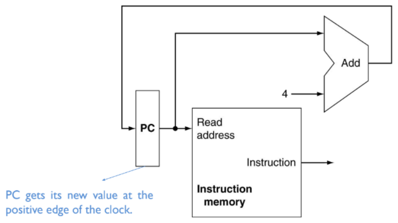
- Lifecycle of each instructon in the controller:
  - *Initial State:* reading the instruction and decoding it.
  - *Next State:* each instruction has its own state.
  - *Future State:* depending on the instruction, there could be several states (e.g., reading registers, loading memory, arithmetic operations, etc.)
- Once all the activities are done, the controller should go back to the *initial state* and *repeat* this process for the next instruction.
  - We call this single-cycle design!

Building a Simple CPU

- RISC-V ISA, 32-bit
- Handful of instructions - only 10
  - add, sub, and, or, lw, sw, beq, addi, andi, ori

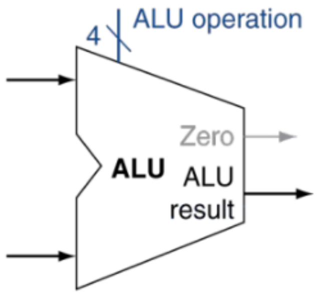
Creating nextPC

Implement the part where you increment the program counter by 4.



- Note that this will be modified later since we have branching, and not every instruction will increment by 4!

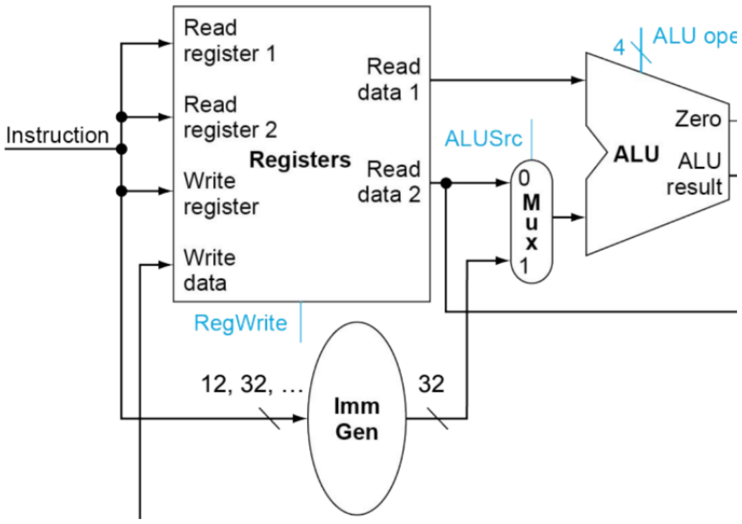
ALU



ALU has two 32-bit inputs, an ALU result and Zero (x0) output, as well as a ALU operation with 4 bits, which selects the ALU operation to execute.

Register to ALU

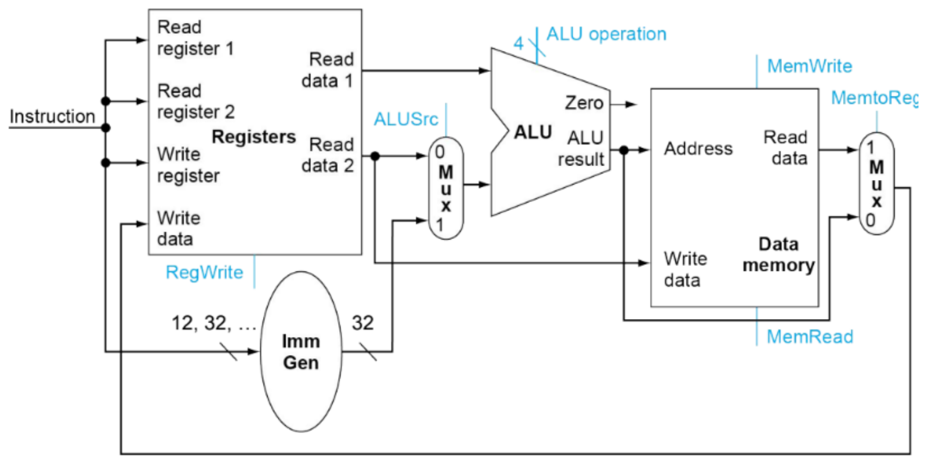
We need to connect the registers to the ALU.



- The Immediate Generator takes in the bits of the instruction and shuffles it with the immediates to get the correct instruction format. (funct7, rs2, rs1, funct3, rd, opcode, etc.) Also does zero-padding if necessary.
- The registers provides the values in the registers that are needed.

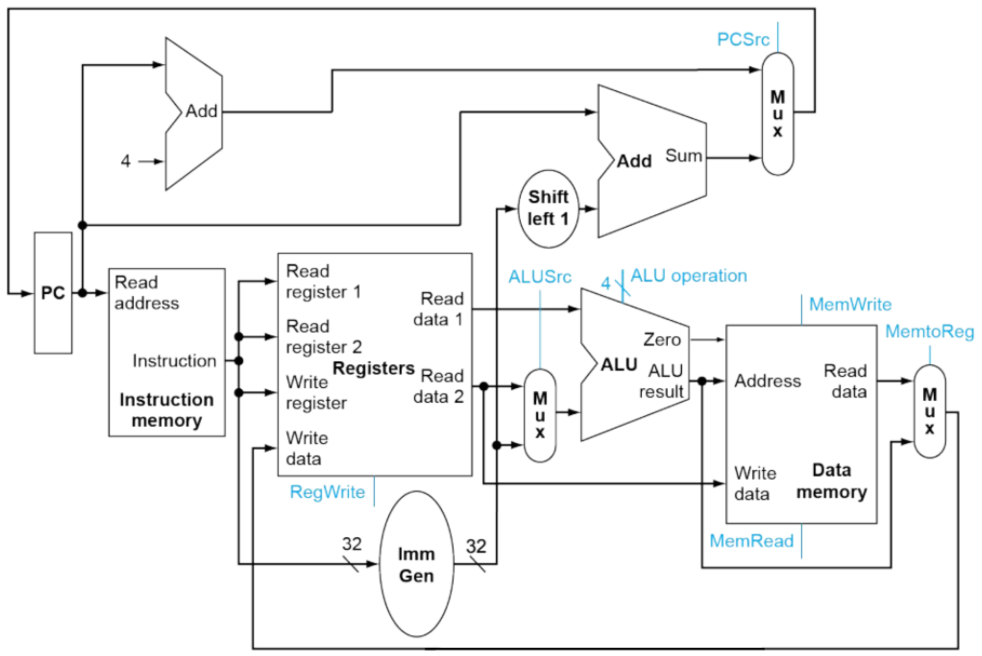
- For R-type instructions without the need of immediates, the immediate generator generates garbage. However, it does not matter since the MUX will not select it for the ALU.

Putting it all together



The MemtoReg multiplexer decides whether data should be written.

- This is the completed datapath used to execute an instruction.
- Now we need to wire it to the controller to pick which instruction to go to.



- Notice that we added an additional MUX to pick whether to go to the next instruction (PC += 4), or branch somewhere else.
- The immediate generator also generates the offset in which to jump.
- We now have the datapath and the instruction selector, but still need a module to fill in the rest (control signals - blue wires).

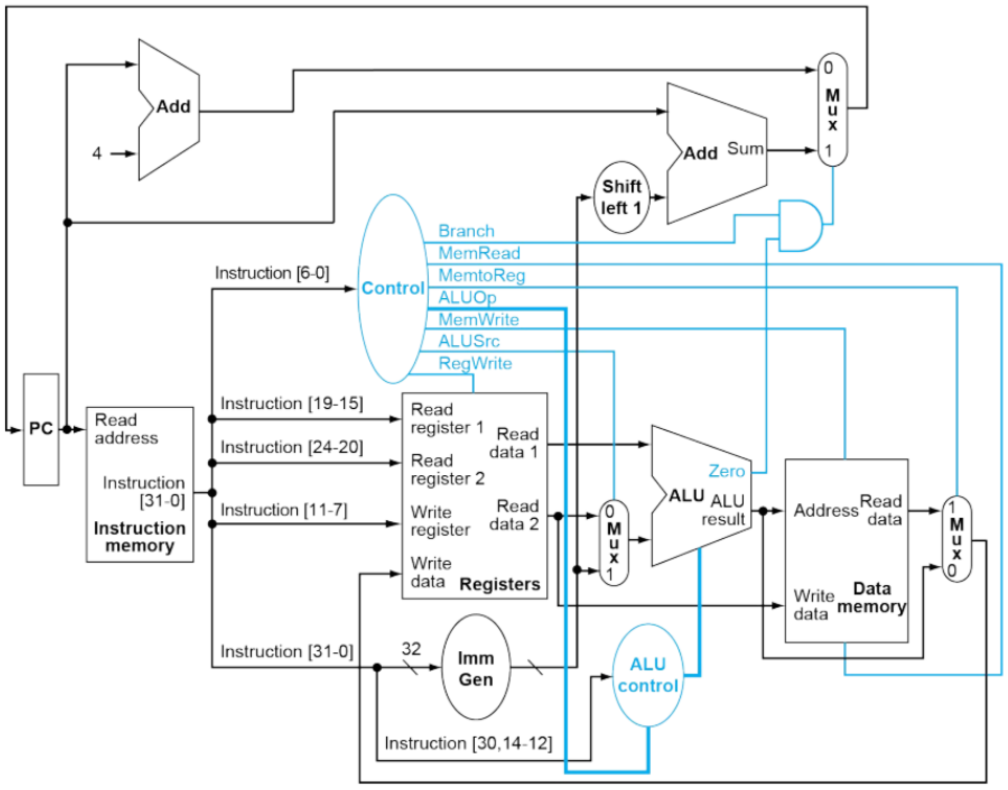
Control Signals

There are seven control signals:

- RegWrite, ALUSrc, ALUOp, MemRead, MemWrite, MemtoReg, PCSrc
- This is where the finite state machine is used.

Control Signal FSM

- First, initialize the FSM state to zero.
- Based on the instruction (opcode), and current state, some control signals should be changed.
  - First, we need to find out which instruction. (Think: big opcode switch statement. Include funct3 and funct7 if necessary)
  - Once we know the instruction, we issue proper control signals (based on the list).



*Images were taken from Hennessy Patterson Book [1].

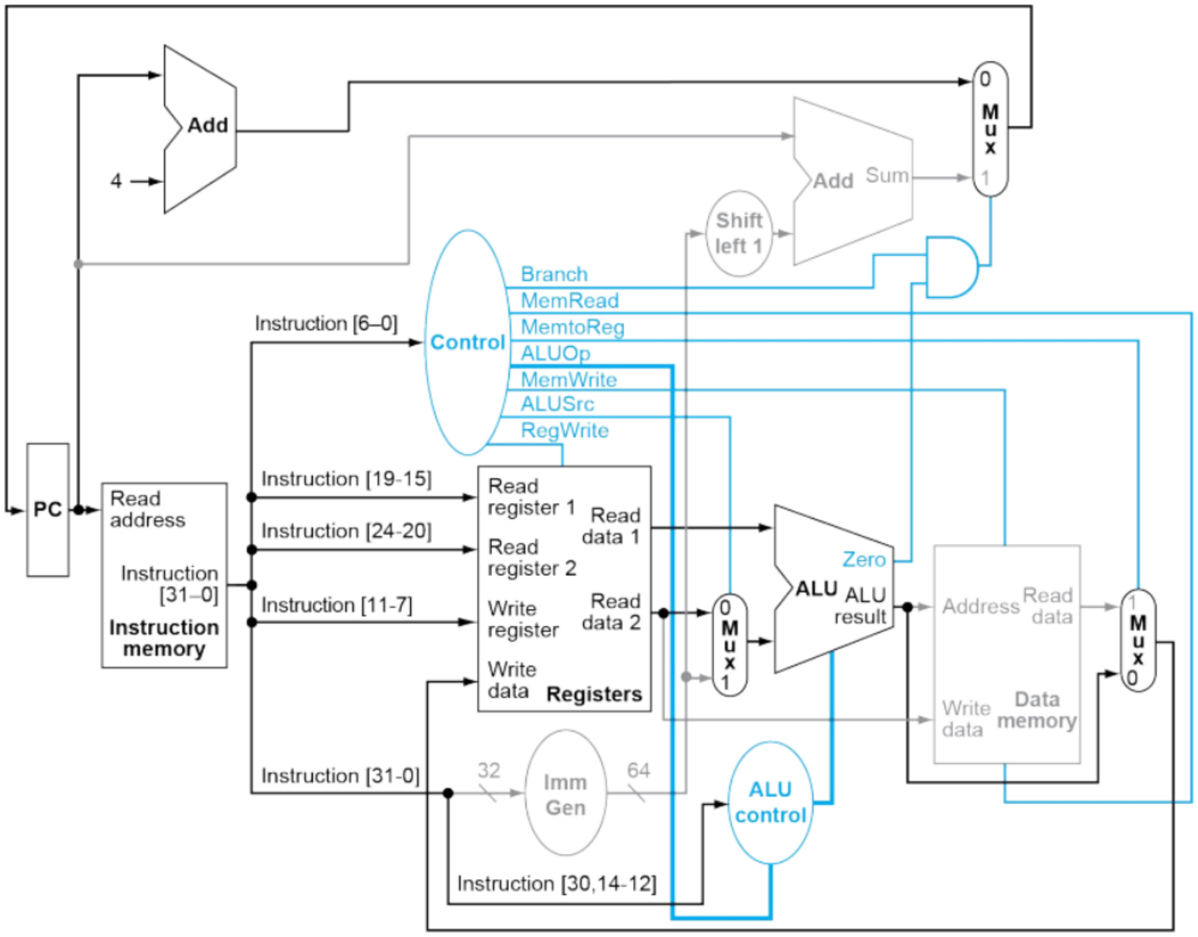
Single-Cycle Design

Instruction	Opcode	RegWrite	AluSrc	Branch	MemRe	MemWr	MemtoReg	ALUOp
R-type	0110011	1	0	0	0	0	0	-
I-type	0010011	1	1	0	0	0	0	-
lw	0000011	1	1	0	1	0	1	-
sw	0100011	0	1	0	0	1	0	-
beq	1100011	0	0	1	0	0	0	-

ALUOp for each instruction:

Instruction	ALU Function	ALU Op
lw	add	0010
sw	add	0010
beq	subtract	0110
R-type / I-type	add	0010
	subtract	0110
	AND	0000
	OR	0001

Dataflow with ALU Controller



Efficiency

- The single-cycle processor is very simple and easy to design, but not the most efficient!
- This is because different instructions have different paths and hence different delays.
- **Longest delay** determines clock period.
- The critical path is the *load instruction*, which takes the longest: Instruction memory → register file → ALU → data memory → register file
- The only advantage of single-cycle designs is that they are simpler.

Can we do better? How do we assess this?

Basic Metrics

- Latency
  - How long does it take to finish a task (measured in sec, ms, μs, ...)
  - Lower is better
- Throughput
  - How many tasks can be done for a given time (measured in instruction per cycle, bit/second, ...)
  - Higher is better
- Power
  - Measured in J/sec
  - Lower is better
- Energy
  - Product of power and latency (a function of both)

Throughput vs. Latency

Is throughput $= \frac{1}{\text{latency}}$?

- If it takes t seconds to do 1 task, then latency $= t$, does throughput $= 1/t$?
- If it takes T seconds to do N tasks, then throughput $= N/T$, does latency $= T/N$?
- If there is concurrency (e.g., running in parallel, then throughput $\neq$ 1 / latency).

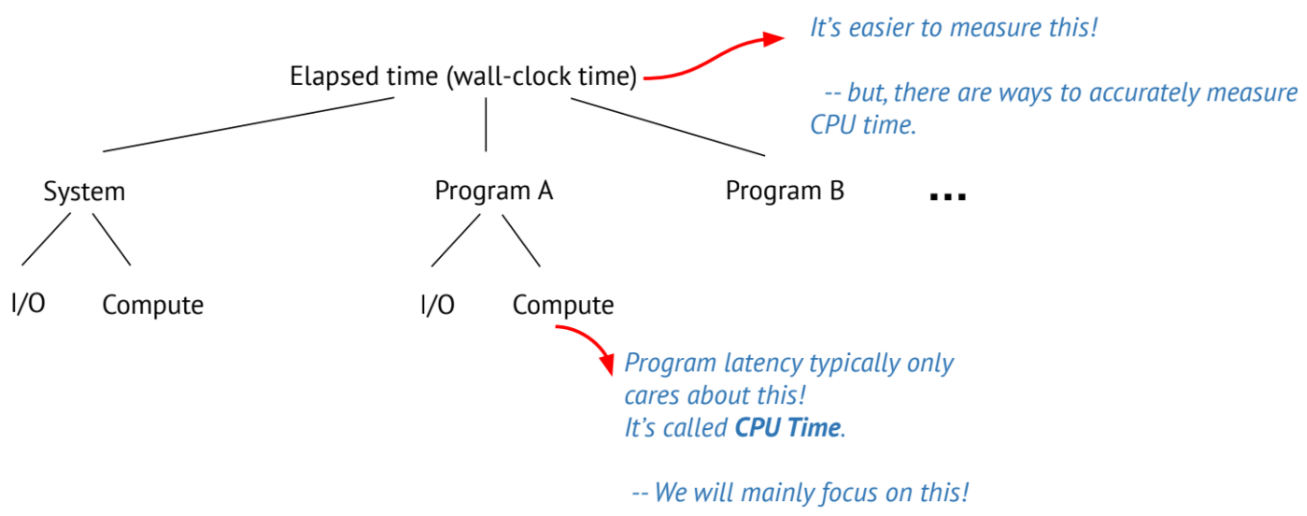
(Confusing) Terminology:

- Improved by 2.5x = improved by 150% = improved by 2.5 times
- For *bigger-is-better* metrics, "improved" means **increase**, but for *smaller-is-better* metrics, "improved" means **decrease**

What is performance?

- **Time** is the measure of computer performance:
  - The computer that performs the **same amount of work** in the **least time** is the fastest.

Measuring Time in a Computer



Elapsed time (wall-clock time)
It's easier to measure this!
-- but, there are ways to accurately measure CPU time.
System
Program A
Program B
...
I/O
Compute
I/O
Compute
Program latency typically only cares about this! It's called **CPU Time**.
-- We will mainly focus on this!

- How to measure CPU Time?
  - Directly: just run the program and measure time! –*too costly*
  - Simulation: create a piece of software that accurately simulates the behavior of a real processor. –*architects love this! But, there is a trade-off between simulation accuracy and complexity!*
- What types of applications should be used?
  - Different applications have different characteristics.
  - Solution: Benchmarking

Benchmark Software

- A set of programs that were carefully designed to measure a specific metric (usually CPU's performance).
- The purpose of benchmarking (depends on who you talk to)
  - **The Architect:** Prove my gizmo is great!
  - **Marketing:** Make us look good to sell $$ and crush our competition.
  - **The Users:** Be our proxy, run our applications on new systems so we don't waste our money or our time.

"Iron Law" of Processor Performance

$$\text{CPU Time} = \text{InstructionCount} \times \text{CyclePerInstruction} \times \text{CycleTime}$$

- InstructionCount
  - IC
  - Total number of instructions that we are executing (i.e., how large is the program)
  - To optimize performance (i.e., reducing CPU Time) IC should be decreased by changing the algorithm or using a better compiler/ISA.
- CyclePerInstruction

  - CPI (also widely used is I/CPI=IPC, or MIPS = million instructions per second)
  - Cycles needed to execute a given instruction
  - CPI differs for different instructions (e.g., MULT vs. ADD) to optimize performance
  - CPI should be decreased by using different *architectural* techniques

- CycleTime
  - CT=I/F_clk
  - Faster clock rate decreases CT.
  - Device and architecture-level techniques can be used to improve CT.
  - Dennard and Moore's Laws

MultiCycle/MultiStage

- Break down the datapath into multiple stages
- Each stage can be executed in one cycle.
- Using these stages we can break down the critical path, and hence increase the clock frequency.
- Intermediate values are stored in registers.

How does *multicycle* improve performance?

- This reduces CycleTime, but also increases CPI. What about overall performance?