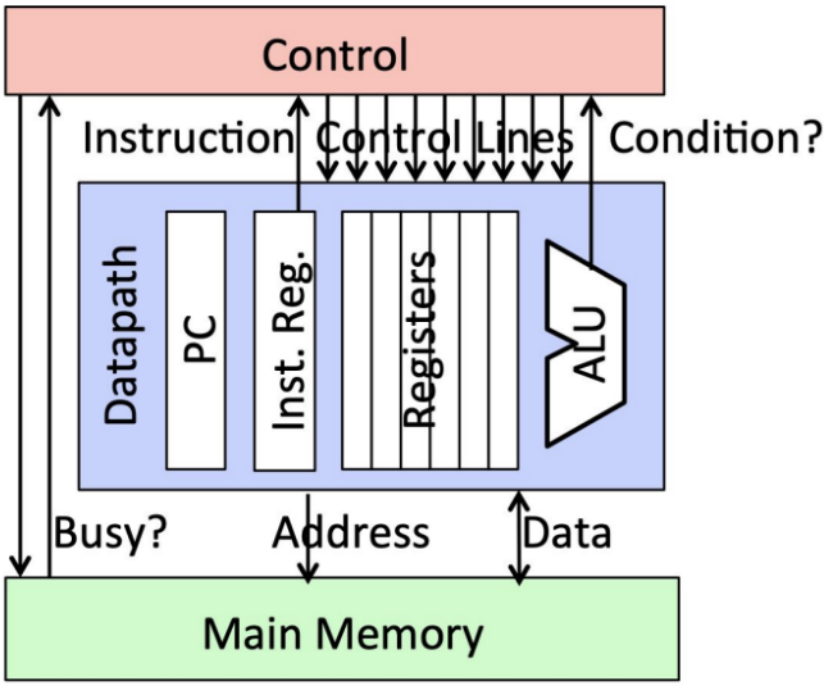
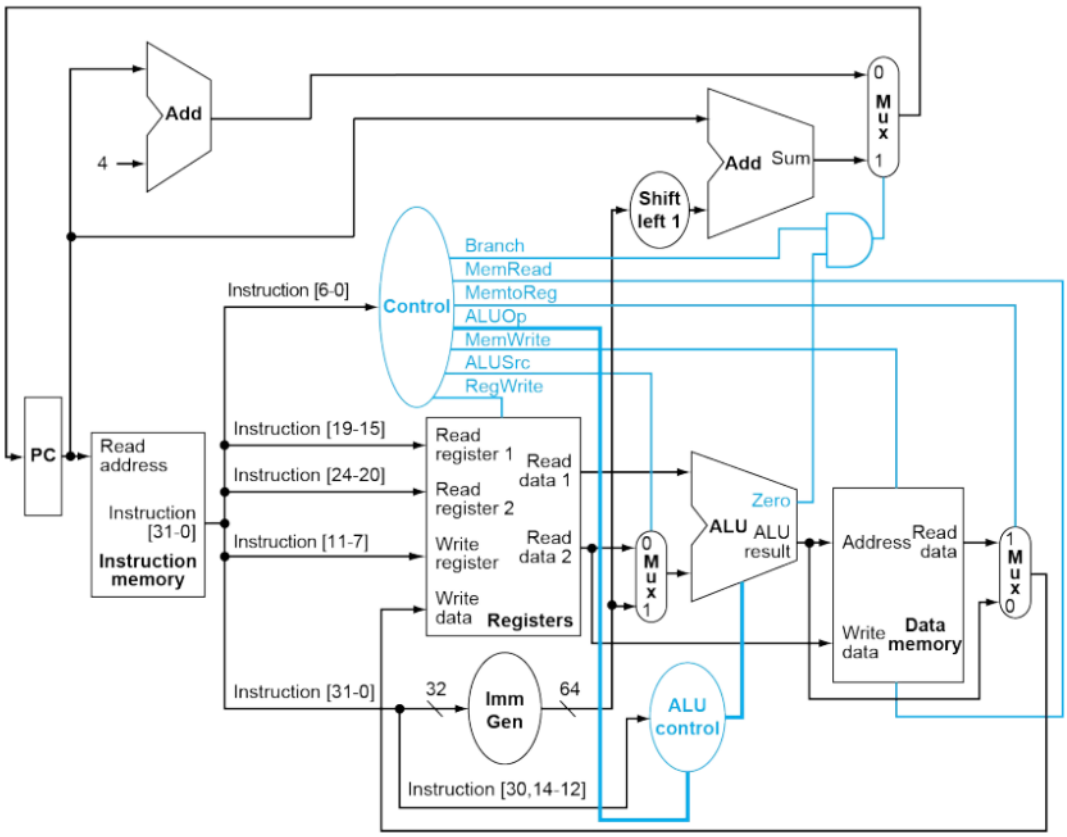
Microarchitecture

- 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].

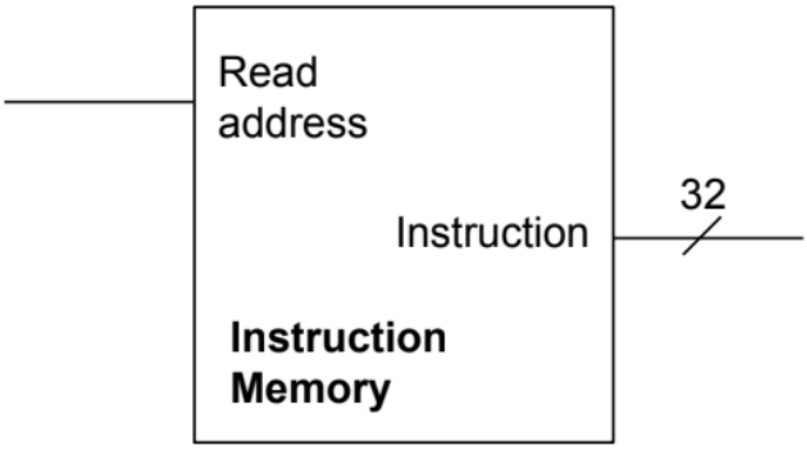
We will go through how this works, step by step. But first, how are instructions executed?

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

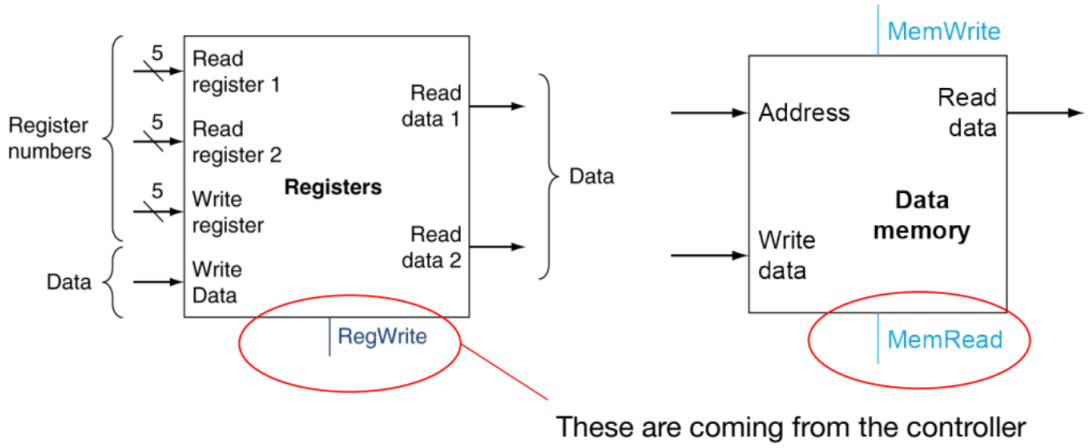
- Are all memory cells registers, SRAM, DRAM, etc.?
- No!
- **Latches and Flip-Flops (aka Registers)**
  - Very fast (1 cycle), parallel access.
  - Very expensive (one bit costs tens of transistors).
- **Static RAM (SRAM)**
  - Relatively fast (10 cycles), only one data word at a time.
  - Expensive (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?

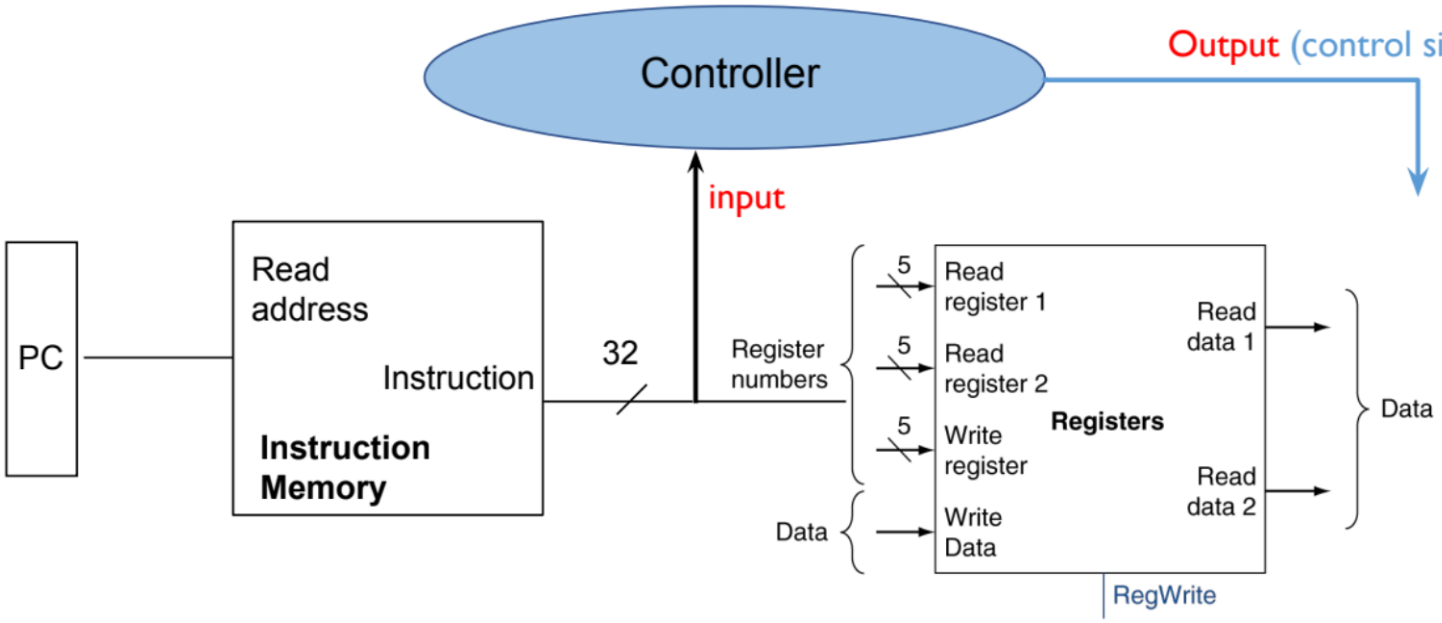
- From Register to DRAM to disk:
  - From *smaller* and *faster* → *bigger* and *slower*
  - From *more expensive* and *power hungry* → *cheaper* and *power efficient*
  - The type we need depends on the purpose we need it for. Use registers for frequently used data and slower and bigger elements for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



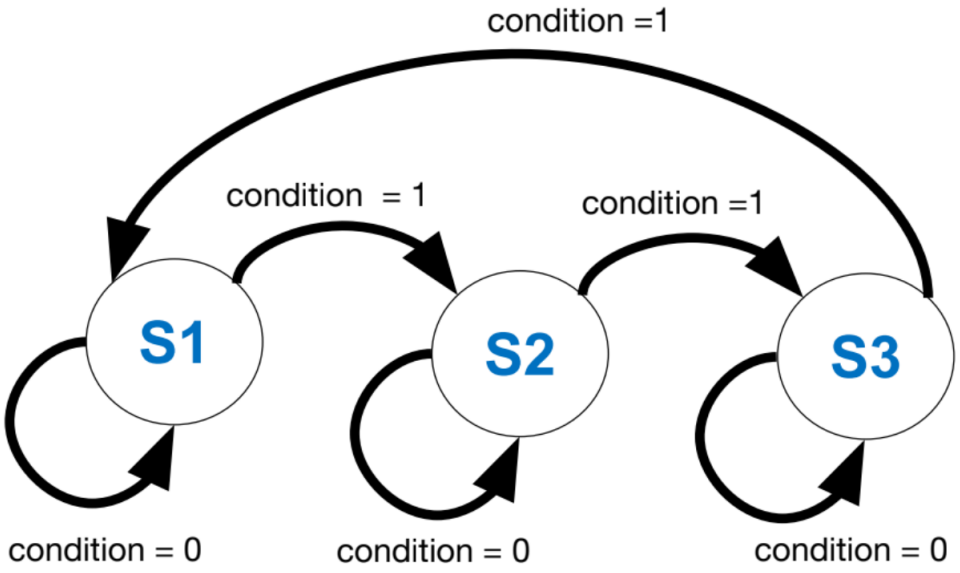
The controller directs the operations for each instruction.



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!

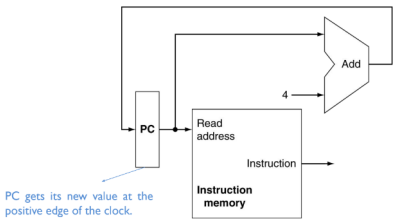
- Lifecycle of each instructon in the controller:
  - *Initial State:* reading the instruction and decoding it.
  - *Next State:* each instruction has its own state.
  - *Future State:* depending on the instruction, there could be several states (e.g., reading registers, loading memory, arithmetic operations, 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

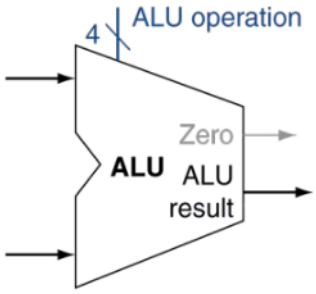
Creating nextPC

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!

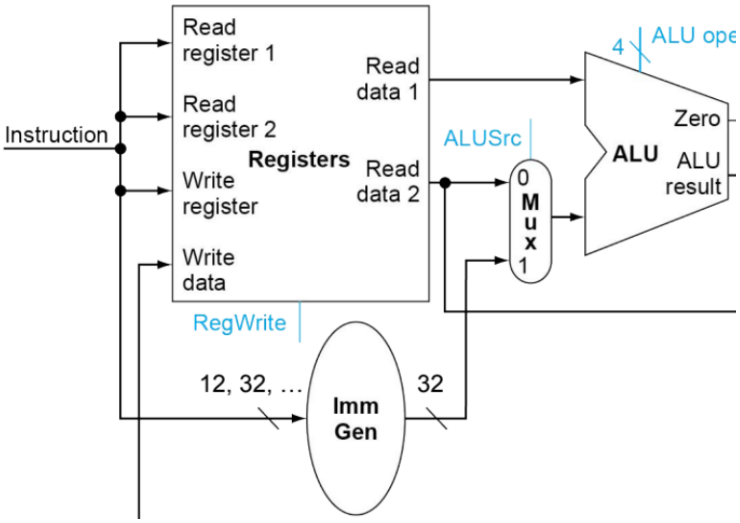
ALU



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

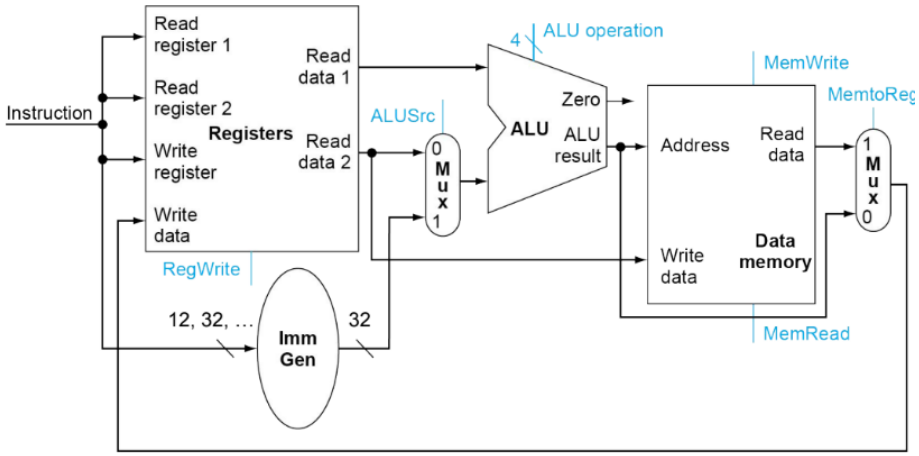
We need to connect the registers to the ALU.



- 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.