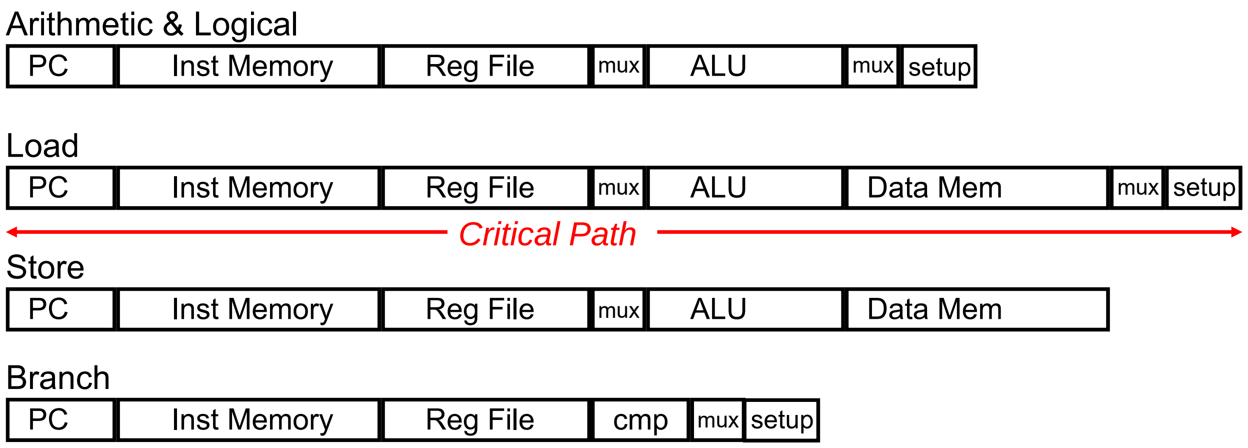
Chapter Four – I (3/4)

Where we are headed

- **Single Cycle Problems:**
  - **The clock cycle is equal to the worst-case delay for all instructions**
  - **Inefficient both in its performance and in its hardware cost**
- **One Solution:**
  - **use a “smaller” cycle time**
  - **have different instructions take different numbers of cycles**
  - **a “multicycle” datapath:**

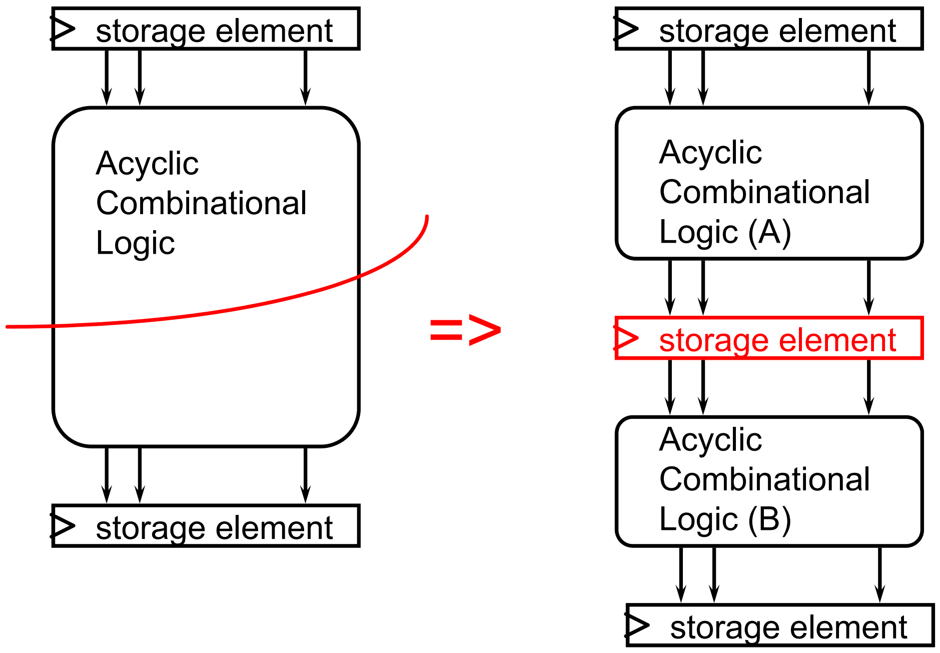
What's wrong with our CPI=1 processor?

- **Long Cycle Time**
- **All instructions take as much time as the slowest**
- **Real memory is not so nice as our idealized memory**
  - **cannot always get the job done in one (short) cycle**



Reducing Cycle Time

- **Cut combinational dependency graph and insert register / latch**
- **Do same work in two fast cycles, rather than one slow one**


storage element
Acyclic Combinational Logic
storage element
=>
storage element
Acyclic Combinational Logic (A)
storage element
Acyclic Combinational Logic (B)
storage element

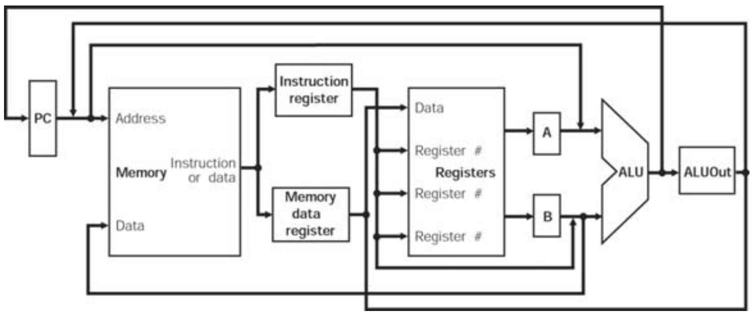
Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control
- Partition data so can handle different instruction execution times.

Multicycle Implementation

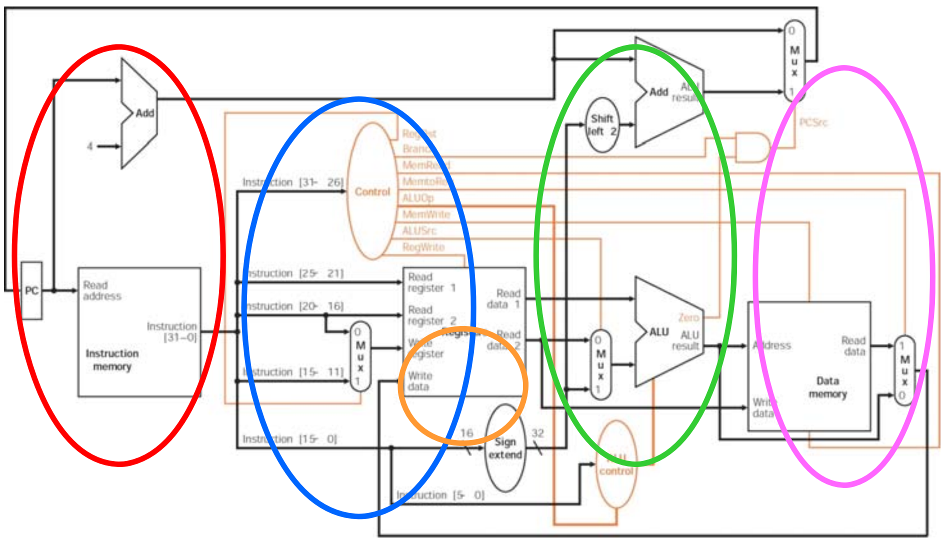
- Break each instruction into a series of steps
  - Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Each step in the execution will take 1 clock cycle
  - Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Allows a functional unit to be used more than once per instruction
  - Reduce the amount of hardware required

- Single memory unit
- Single ALU
- Registers after every major functional unit



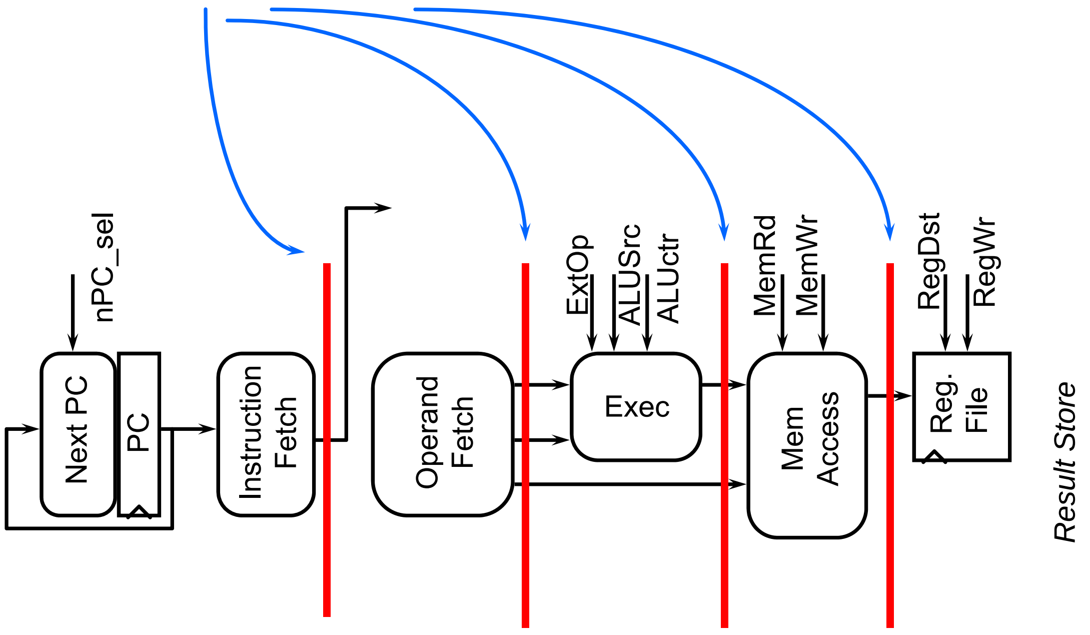
Five Execution Steps

- Instruction Fetch (IF)
- Instruction Decode (ID) and Register Fetch
- Execution, Memory Address Computation, or Branch Completion (EX)
- Memory Access (MEM) or R-type instruction completion
- Write-back step (WB)



Partitioning the CPI=1 Datapath

- Add registers between smallest steps



Allow the instruction to take multiple cycles.

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language“

```
IR ← Memory[PC];
PC ← PC + 4;
```

Updating the PC at this stage can be done using ALU.

Step 2: Instruction Decode and Register Fetch

- Read registers `rs` and `rt` in case we need them
- Compute the branch address in case the instruction is a branch
  RTL:

```
A ← Reg[IR[25:21]];
B ← Reg[IR[20:16]];
ALUOut ← PC + (sign_extend(IR[15:0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3: (Instruction dependent operations)

- ALU is performing one of three functions, based on instruction type
  - Memory Reference:
    ```
    ALUOut ← A + sign-extend(IR[15:0]);
    ```
  - R-type:
    ```
    ALUOut ← A op B;
    ```
  - Branch:
    ```
    if (A==B) PC ← ALUOut;
    ```
- Jump instruction

```
PC ← PC [31:28] || (IR[25:0] << 2);
```

Step 4: (R-type or memory-access)

- Loads and stores access memory

```
MDR ← Memory[ALUOut];
       or
Memory[ALUOut] ← B;
```

- R-type instructions finish

```
Reg[IR[15:11]] ← ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Step 5: Write-back step

- **Load instructions finish**

```
Reg[IR[20:16]] ← MDR;
```

What about all the other instructions?

Summary:

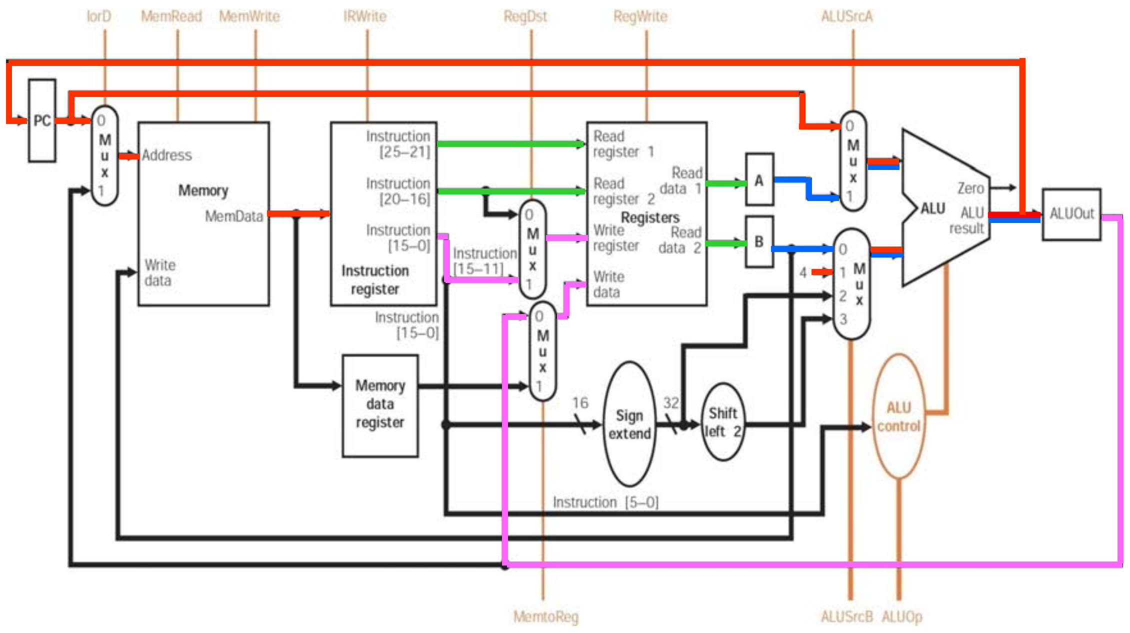
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

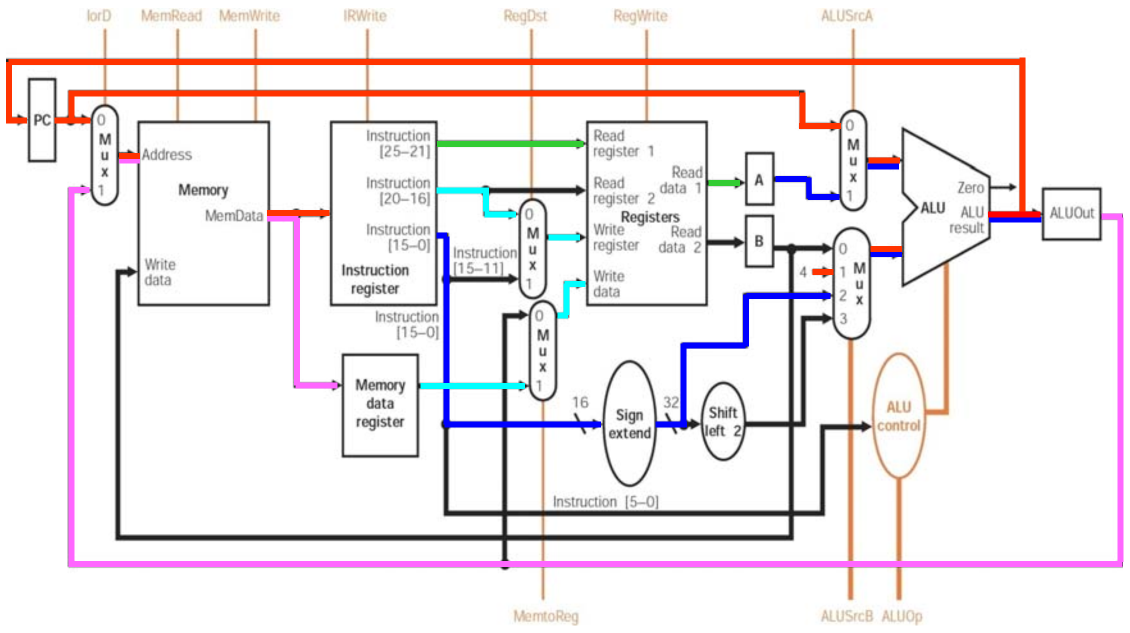
Instruction decode/register fetch	A = Reg [IR[25:21]]
	B = Reg [IR[20:16]]
	ALUOut = PC + (sign-extend (IR[15:0]) << 2)			

Execution, address computation, branch/jump completion	ALUOut = A op B	ALUOut = A + sign_extend(IR[15:0])	if (A ==B) then PC = ALUOut	PC = PC [31:28] II (IR[25:0]<<2)
Memory access or R-type completion	Reg [IR[15:11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20:16]] = MDR		

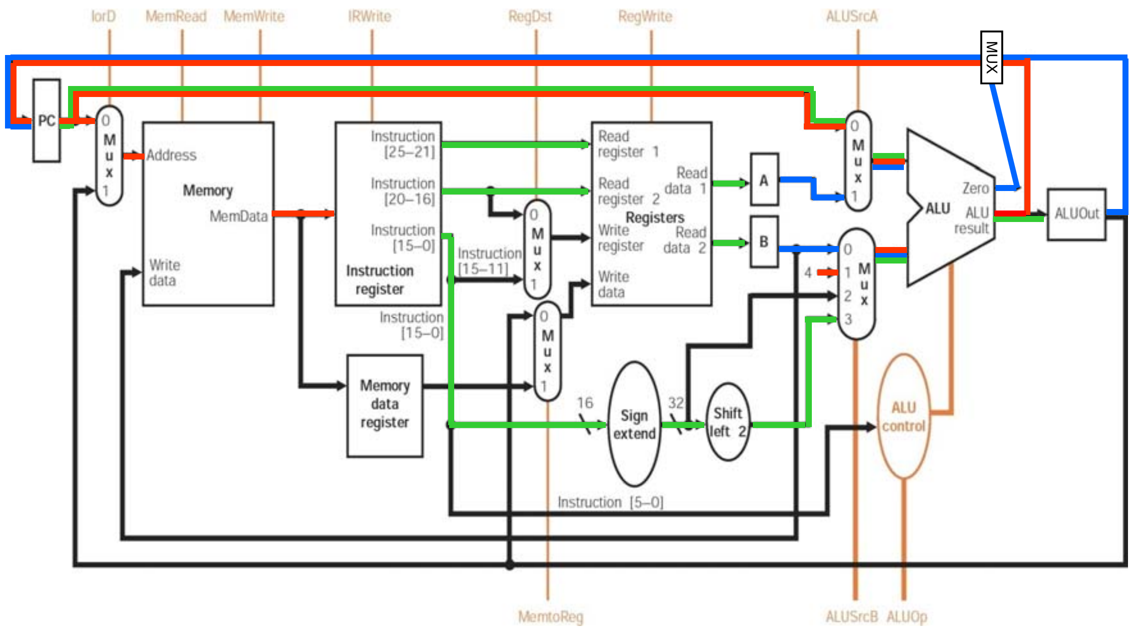
Multicycle Datapath for R-type Instruction



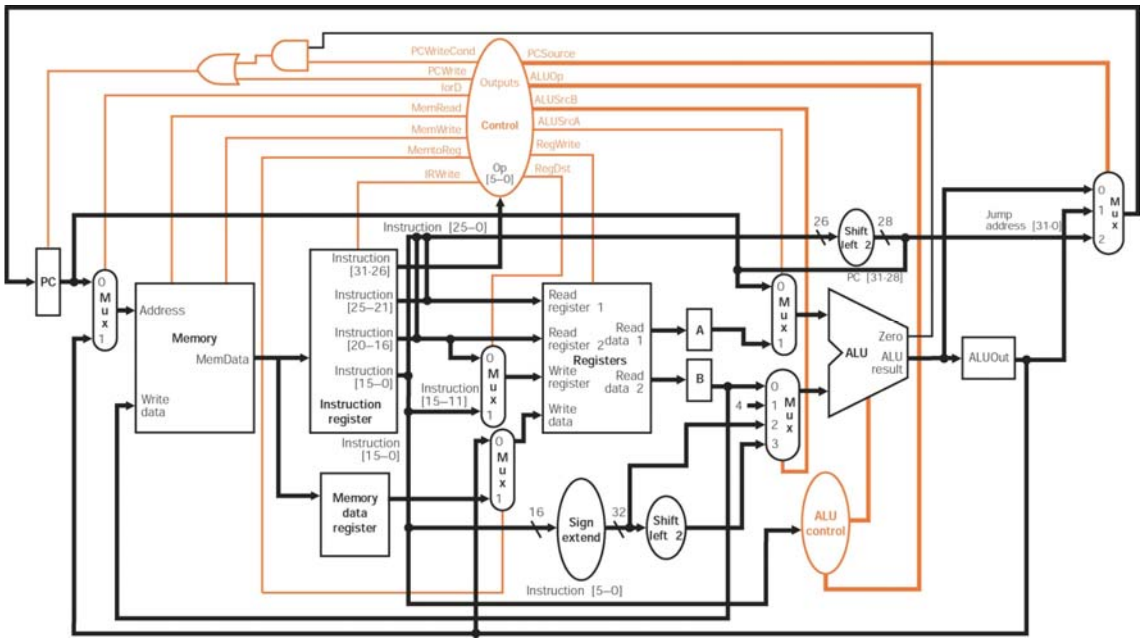
Multicycle Datapath for Load Instruction



Multicycle Datapath for Branch Instruction



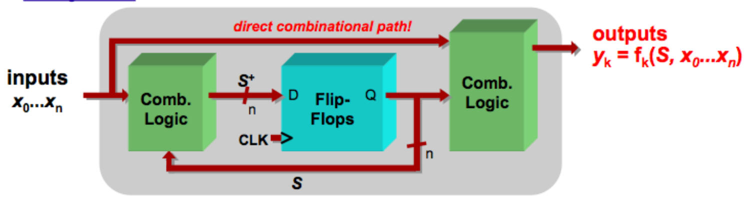
Multicycle Datapath with Control Lines



Control for Multicycle Implementation

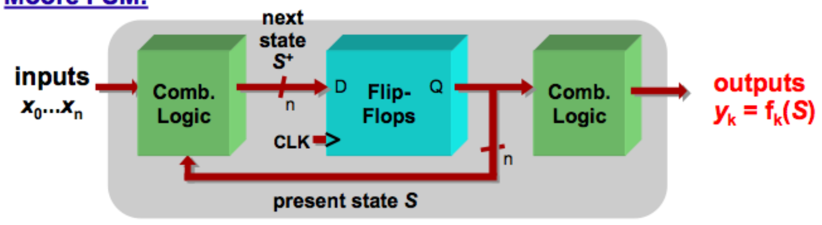
- Two techniques
  - Finite State Machine
  - Microprogramming

Mealy FSM:

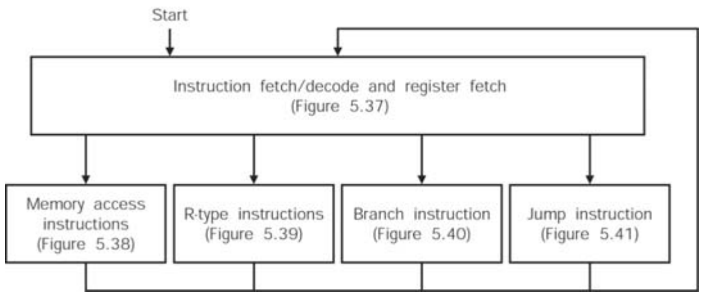


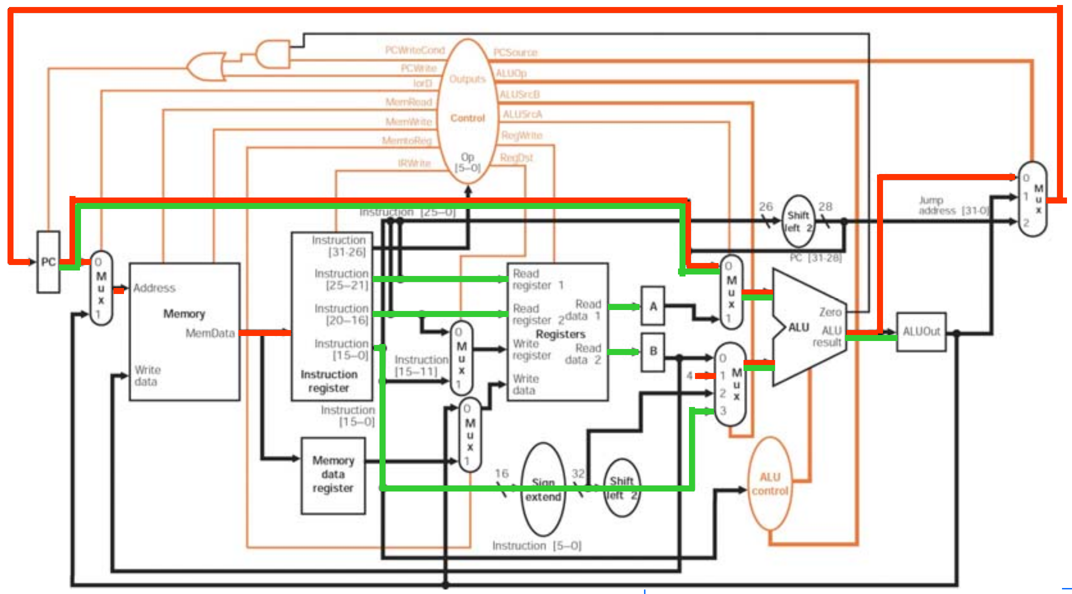
- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)
  - a Moore machine (output based only on current state)
  - a Mealy machine (output based on current state & inputs)

Moore FSM:

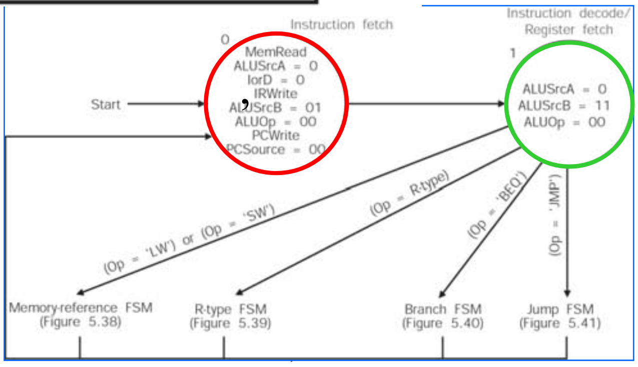


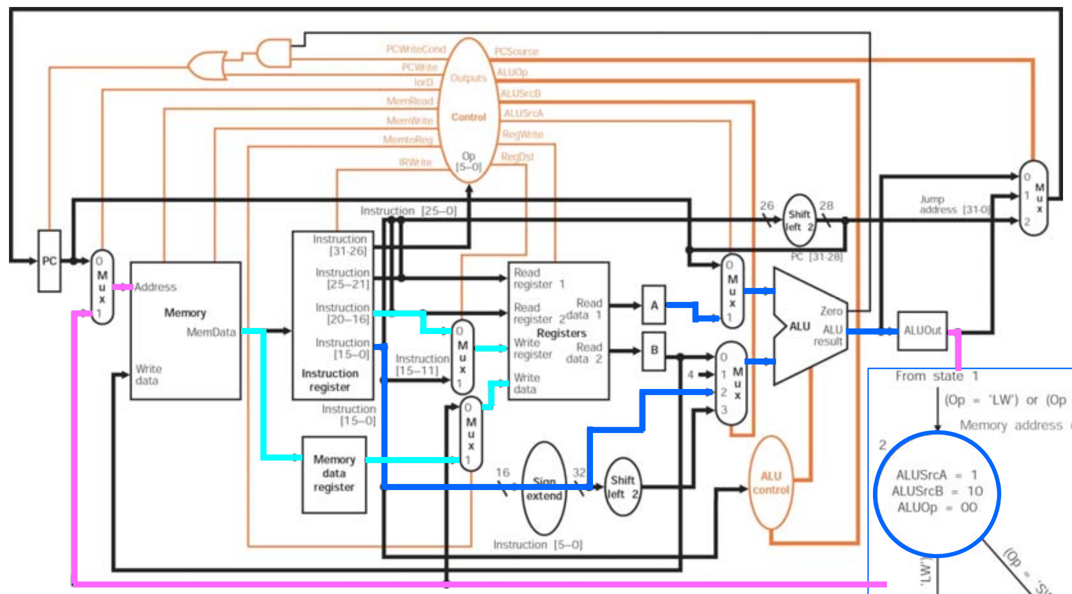
Control for Multicycle Implementation



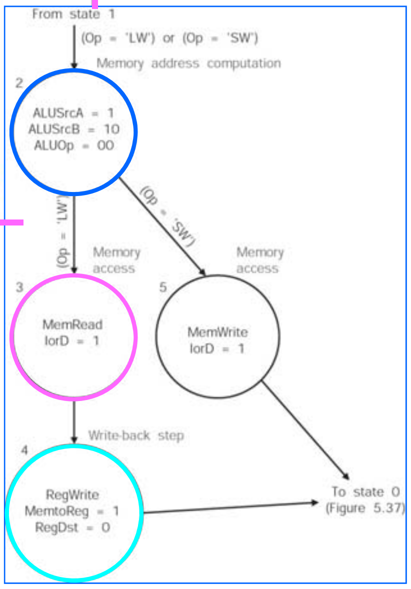


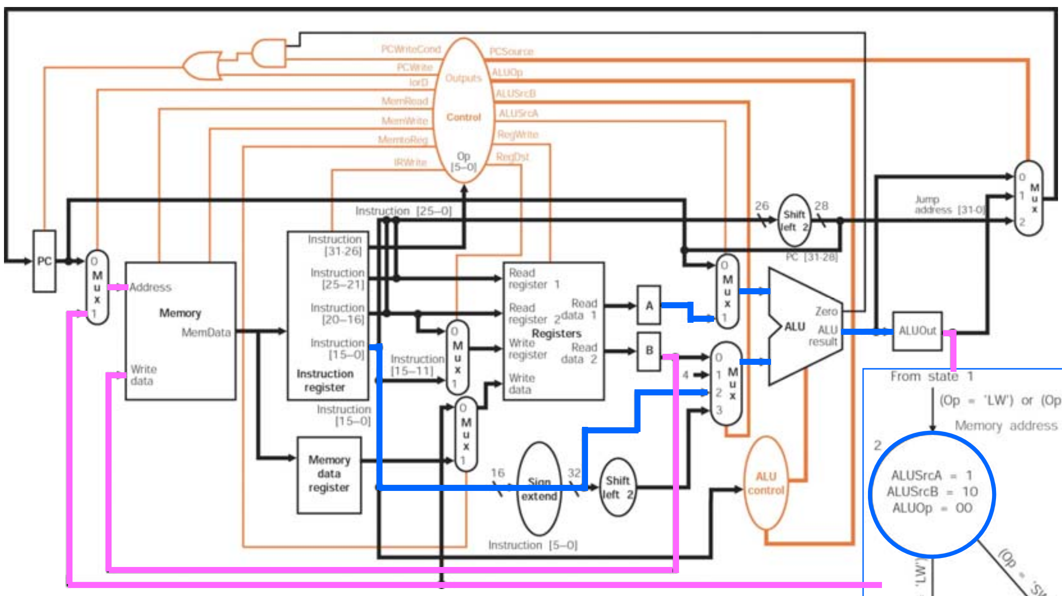
Instruction Fetch and Decode

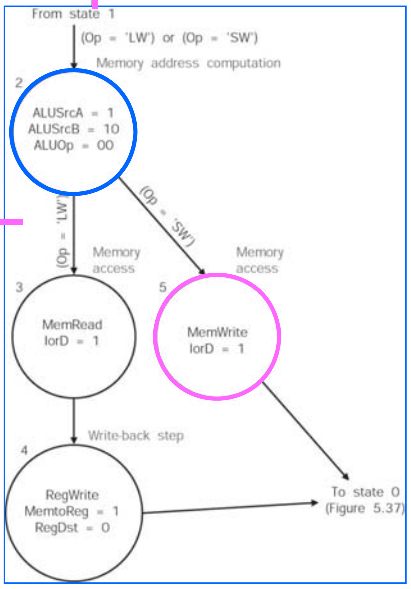




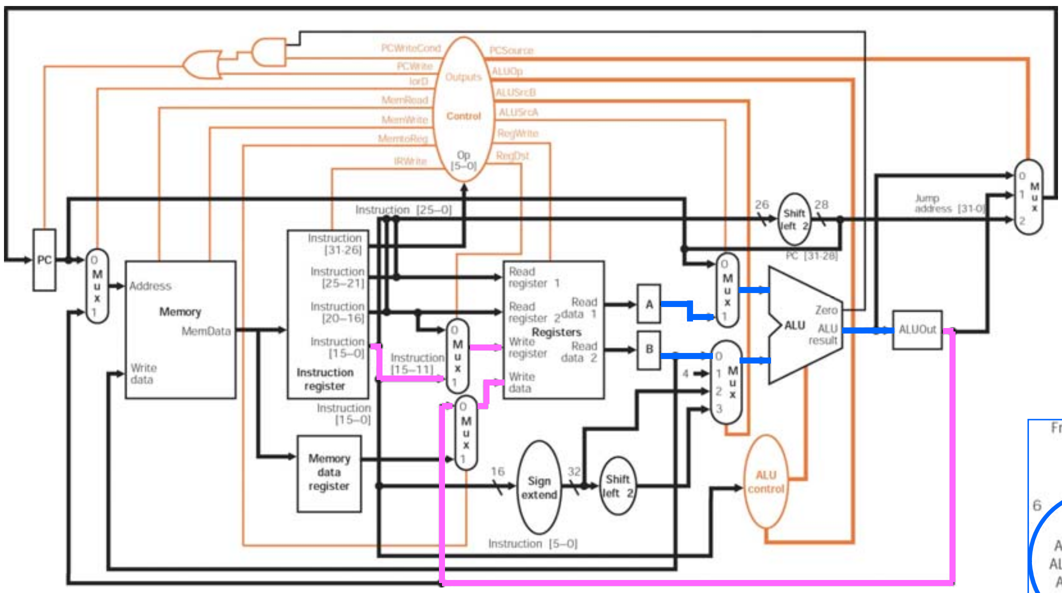
Memory Reference Instructions (load)

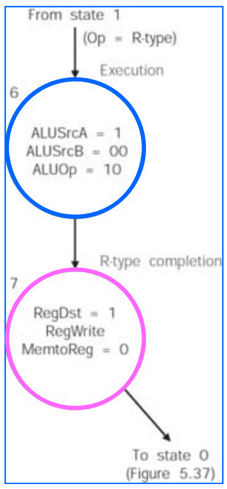




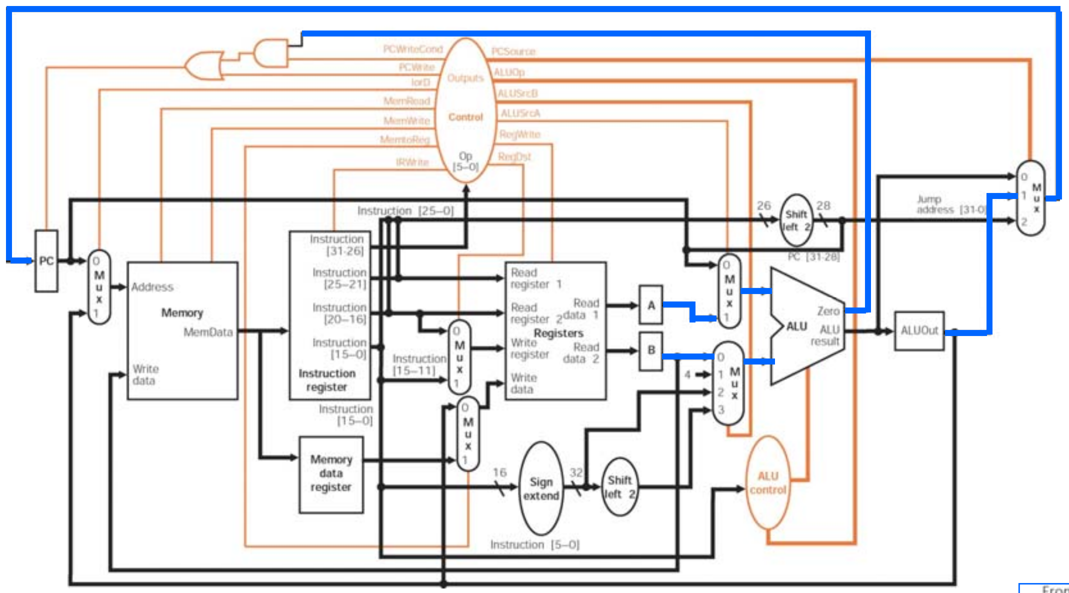


Memory Reference Instructions (store)

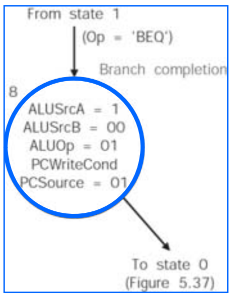


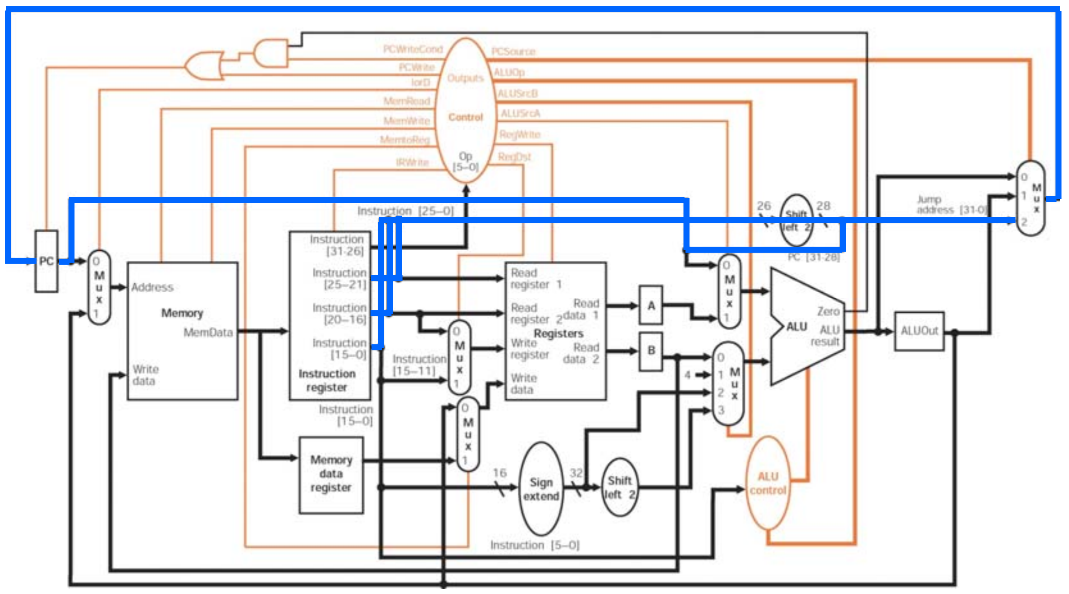


R-type Instructions

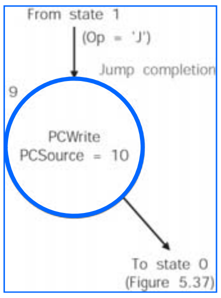


Branch Instructions





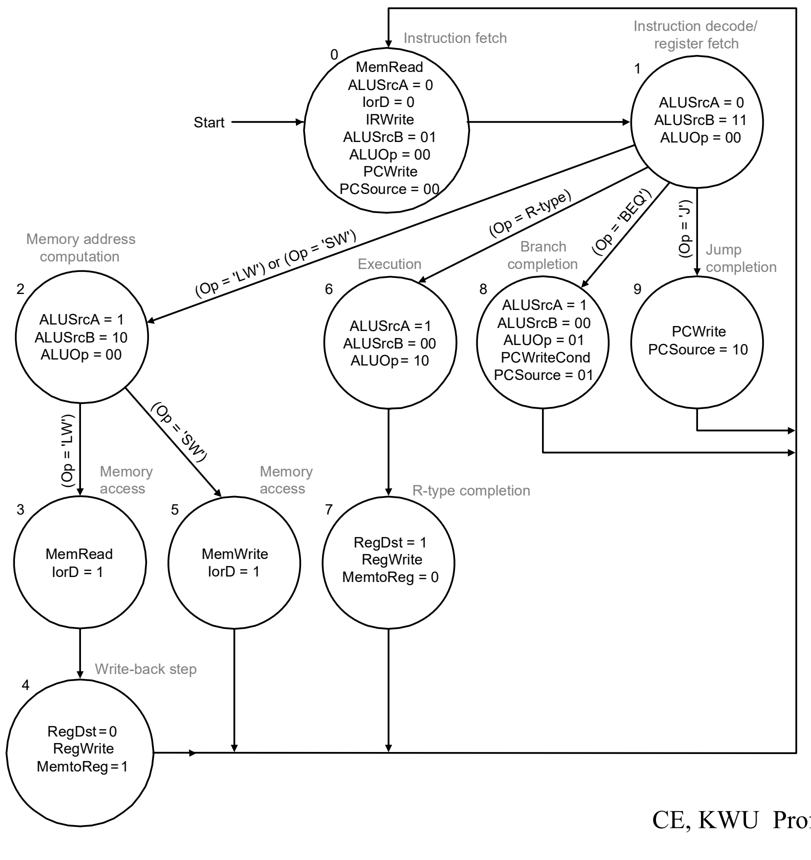
Jump Instructions



Implementing the Control

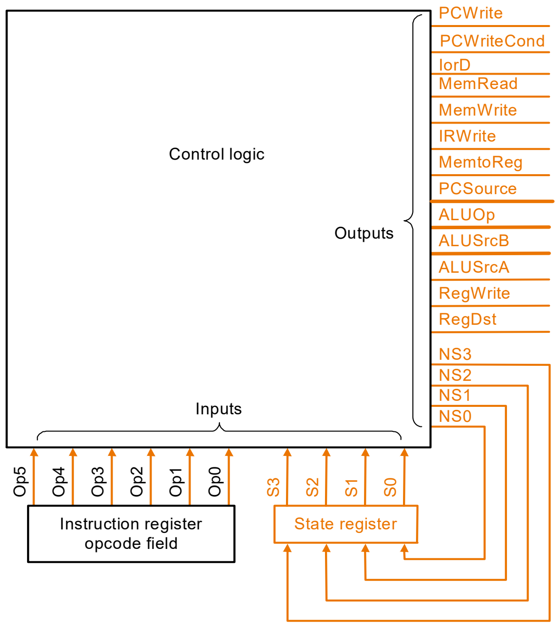
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Complete Finite State Machine Control

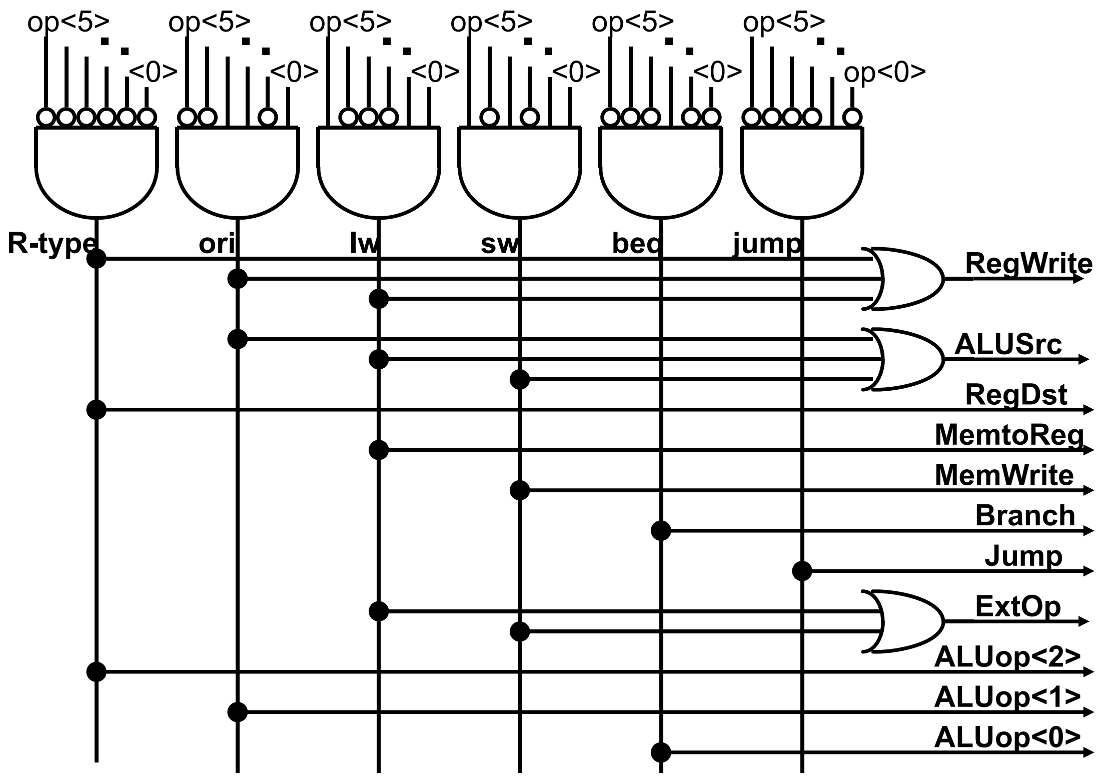


Finite State Machine for Control

- **Implementation:**

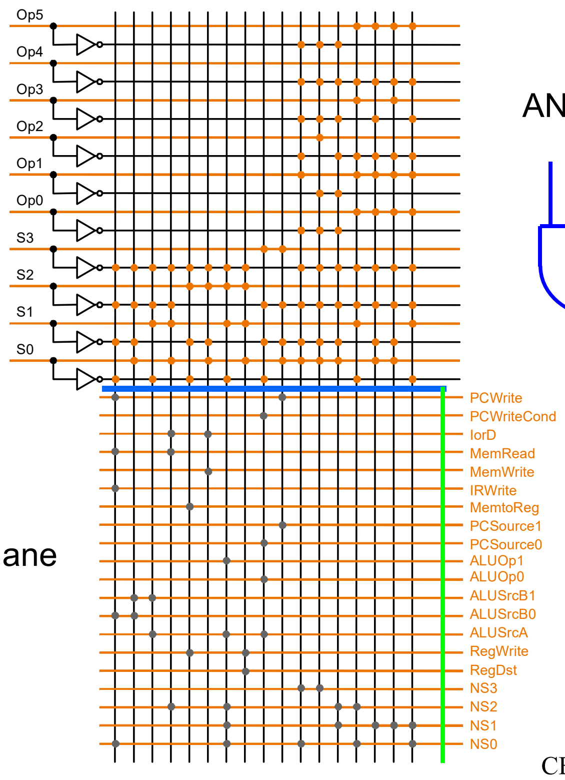


Recap: PLA Implementation of the Main Control

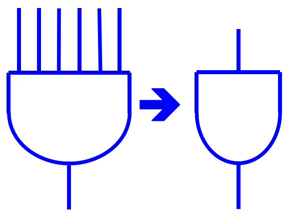


PLA Implementation

- **If I picked a horizontal or vertical line could you explain it?**



AND Plane



OR Plane