ECE 3057: Architecture, Concurrency and Energy in Computation Summer 2019

Lectures 3b & 4: Multi-Cycle

David Anderson

School of Electrical and Computer Engineering
Georgia Institute of Technology

Acknowledgment: Lecture slides adapted from GT ECE 3056 (S. Yalamanchilli and Tushar Krishna) and MIT 6.823 (Arvind and J. Emer)

Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

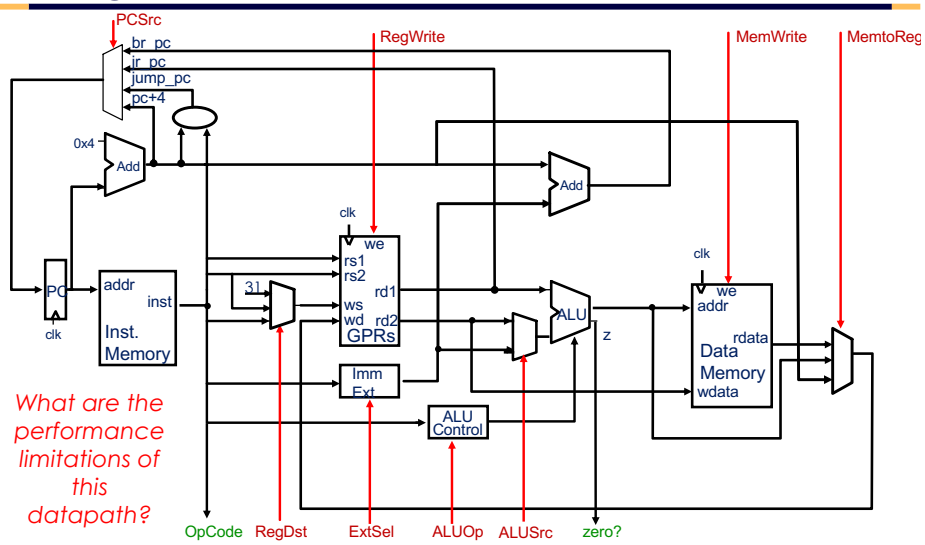
- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

Appendices: A.7, D.3, D.4, D.5

this lecture →

Microarchitecture	CPI	cycle time
Single-cycle unpipelined	1	long
Pipelined	1	short
Multi-cycle/Micro-coded	>1	short

Single Cycle Datapath



What are the performance limitations of this datapath?

Single Cycle Datapath Limitations

Instruction Class	Instruction Fetch 2ns	Register Read 1ns	ALU Operation 2ns	Memory Access 2ns	Register Write 1ns	Total Time	CPI
ALU	✓	✓	✓		✓	6ns	1
Load	✓	✓	✓	✓	✓	8ns	1
Store	✓	✓	✓	✓		7ns	1
Branch	✓	✓	✓			5ns	1

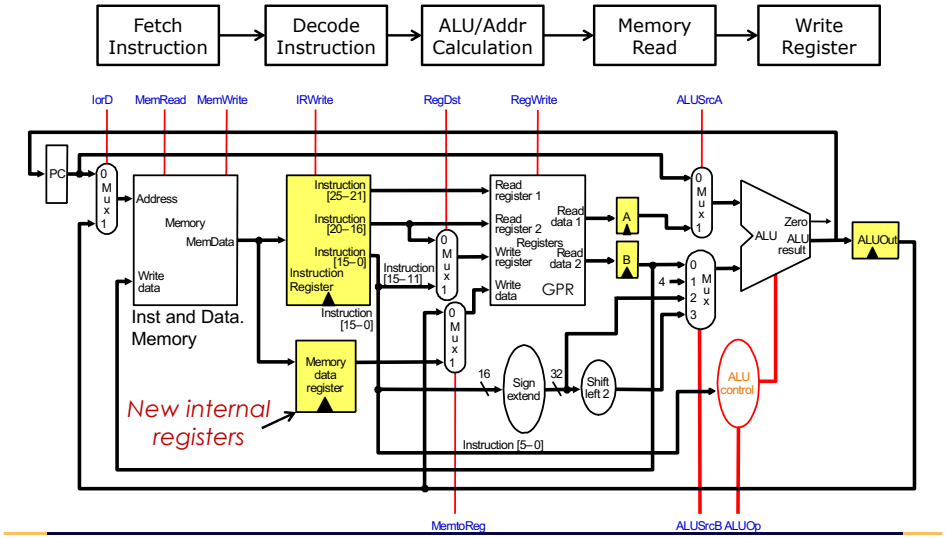
- Cycle Time of Processor?
  - 8ns
  - Single-Cycle Datapath: Design for the worst case!

Multi-Cycle Datapath

Instruction Class	Instruction Fetch 2ns	Register Read 1ns	ALU Operation 2ns	Memory Access 2ns	Register Write 1ns	Total Time	CPI
ALU	✓	✓	✓		✓	8ns	4
Load	✓	✓	✓	✓	✓	10ns	5
Store	✓	✓	✓	✓		8ns	4
Branch	✓	✓	✓			6ns	3

- Break up the instructions into steps, each step takes a clock cycle
  - Balance the amount of work to be done
  - Cycle time? 2ns
- At the end of a cycle
  - Store values for use in later cycles
  - Introduce additional "internal" registers

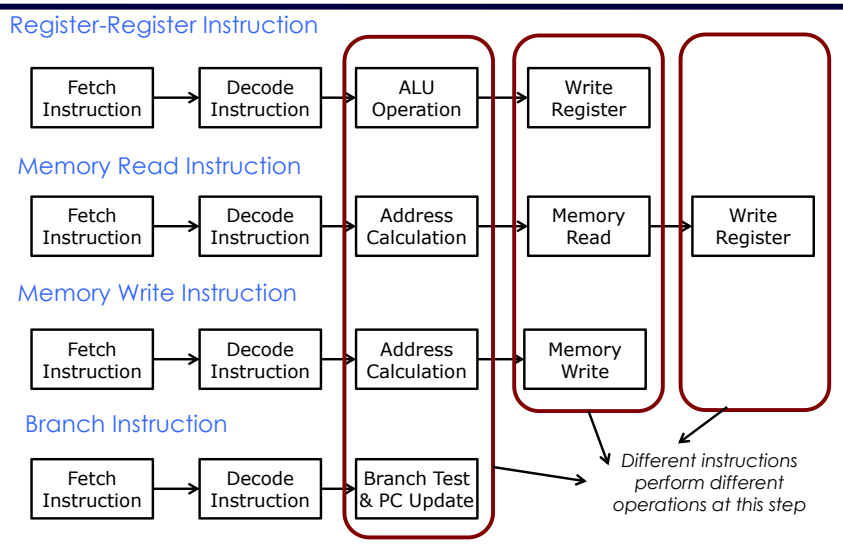
Multi-Cycle Datapath



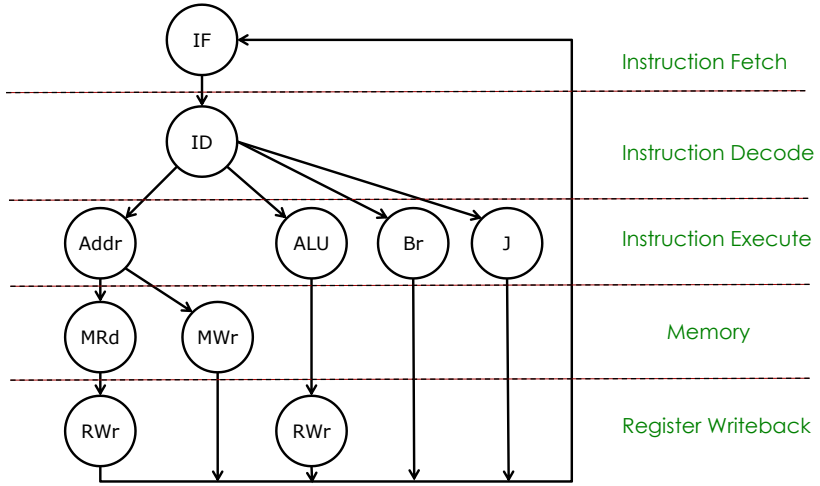
Five Execution Steps

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

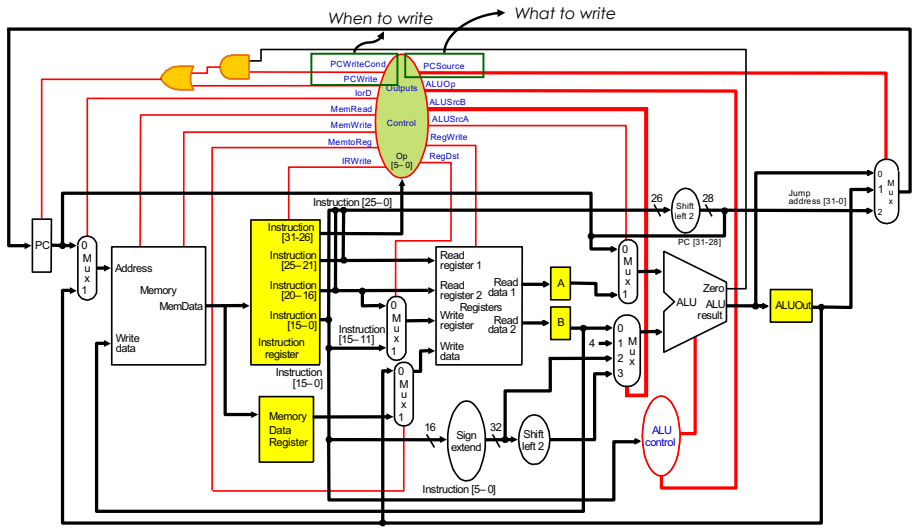
Instruction Execution Steps



Functional Behavior



Multi-cycle Datapath & Control



Control Signals

Stage	Op Code	IorD	ALU Op	ALU SrcA	ALU SrcB	Reg Wr	Reg Dst	Mem Wr	Mem Rd	Mem toReg	IR Wr	PCWr Cond	PC Wr	PC Src
IF														
ID														
EX	ALU													
	LW/ SW													
	BEQ/ BNE													
	J													
MEM	LW													
	SW													
WB	ALU													
	LW													

ALU Control

3 modes

ALUOp	Operation	Opcodes	funct
00	Add	LW/SW, All IF	XXXXXX
01	Subtract	BEQ/BNE	XXXXXX
10	funct	add	100000
		subtract	100010
		AND	100100
		OR	100101
		set-on-less-than	101010

Next PC Address Calculation

Branch Test

ALU Instruction

Instruction Fetch (IF) Stage

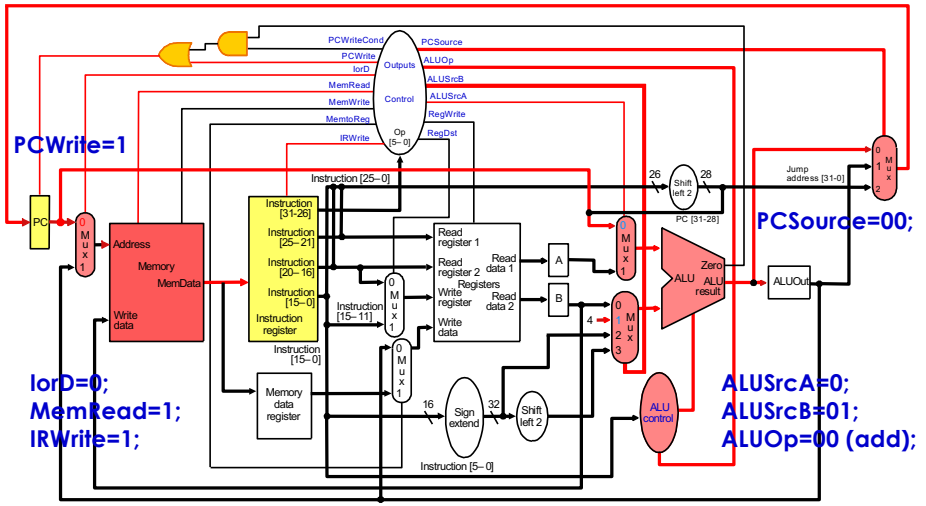
- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

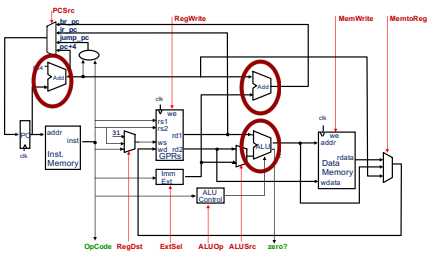
- What will be the values of the control signals?

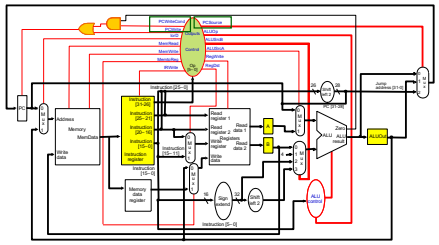
Instruction Fetch (IF) Stage

IR = Memory[PC]; PC+=4



Single-Cycle vs Multicycle Datapaths





Why did we need separate ALUs for PC calculation, LW/SW addr calculation and ALU ops in Single-cycle Datapath but reuse same ALU in Multi-cycle Datapath?

In Single-Cycle, a BEQ instruction needs to use all three ALUs at same time, but in Multi-cycle, same ALU can be used in different cycles

How would you implement the single-cycle datapath with one ALU?

ALU will form a "structural hazard". Need to spend 3 cycles for BEQ

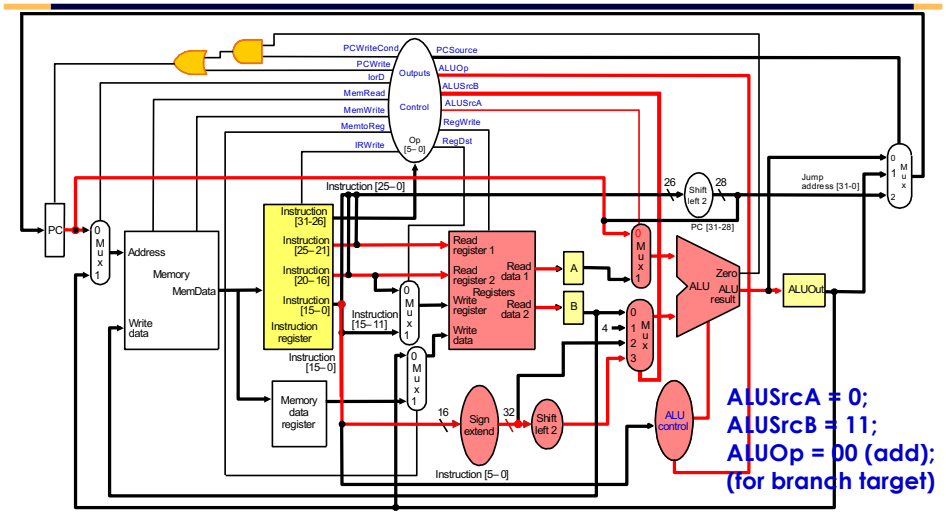
Instruction Decode and Register Fetch (ID) Stage

- Decode the Instruction
  - Still do not have any idea what instruction it is
- Read registers rs and rt in case we need them
- Compute the branch address (used in next cycle 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);
```

Decode & Reg Fetch (ID) Stage:

Assign A and B; Calculate Branch Address



Execute, memory or branch (EX): instruction dependent

- **R-type**:

```
// Instruction specified ALU Operation

ALUOut = A op B;
```

- **Memory Reference**:

```
// Addr Calculation

ALUOut = A + sign-extend(IR[15-0]);
```

- **Branch**:

```
// Branch Test & PC Update

if (A==B) PC = ALUOut;
```

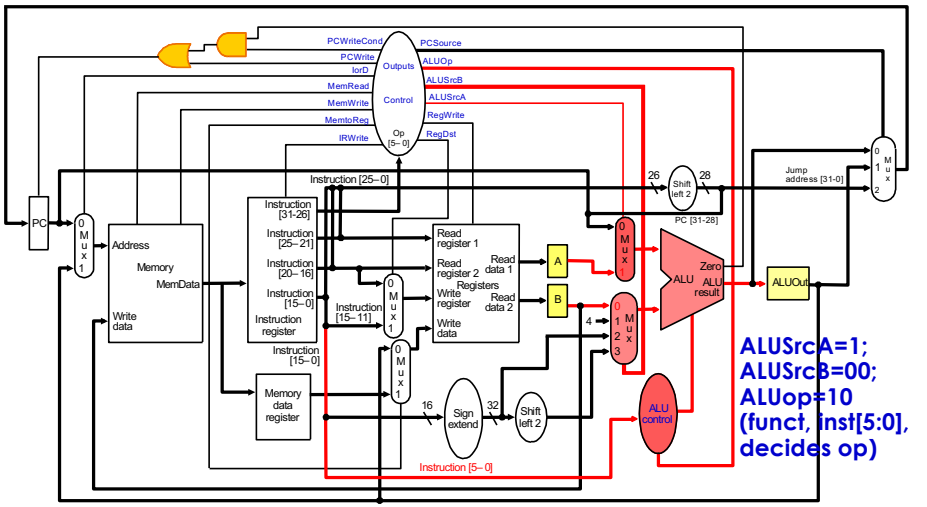
- **Jump**:

```
// Jump PC Calculation

PC = {PC[31:28] || IR[25:0] || 2 'b00};
```

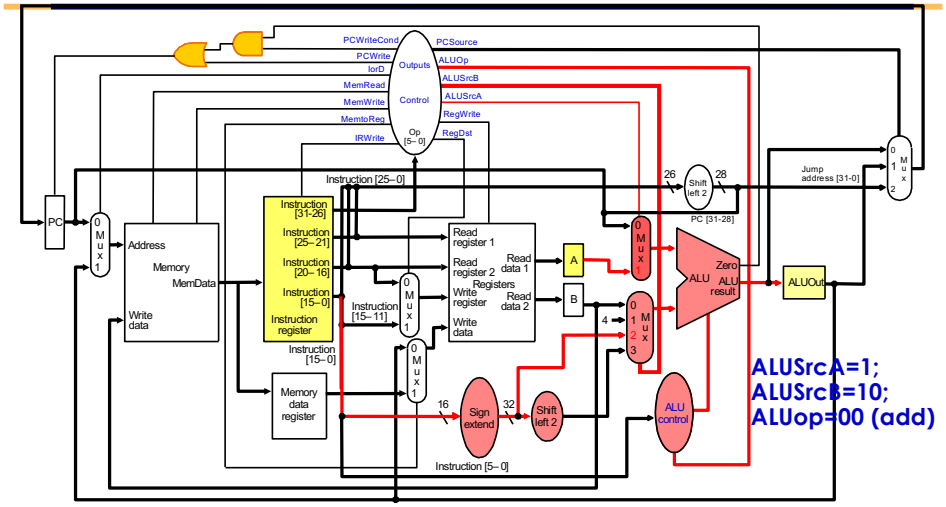

Execute: R-Type

ALUOut = A op B



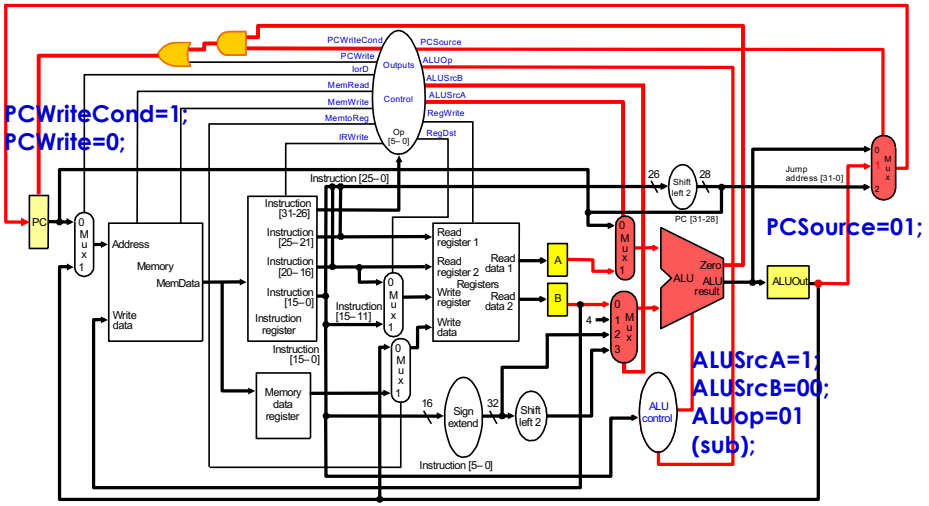
Execute: Memory Type

ALUOut = A + sign-extend(IR[15-0]);



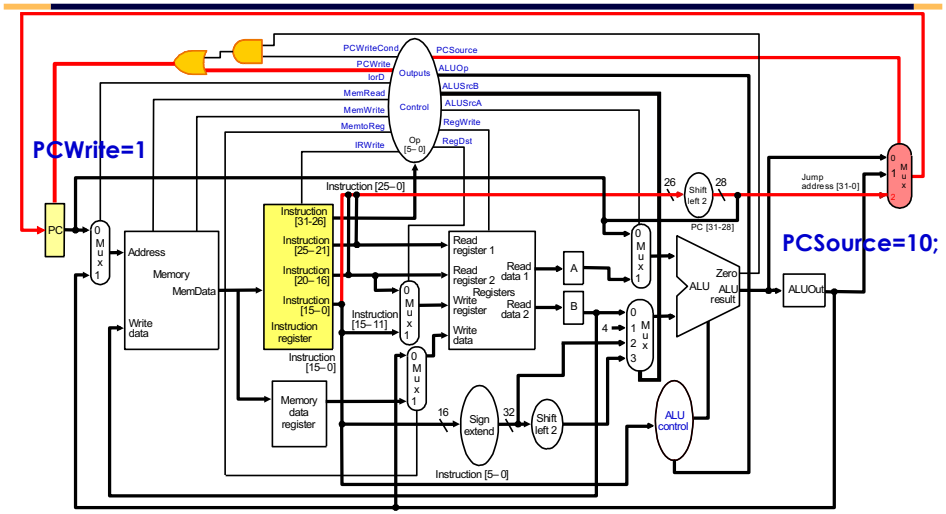
Execute: Branch Type

if (A==B) PC = ALUOut



Execute: Jump Type

PC = {PC[31:28] | | IR[25:0] | | 2'b00};



Memory Access (MEM)

- Load

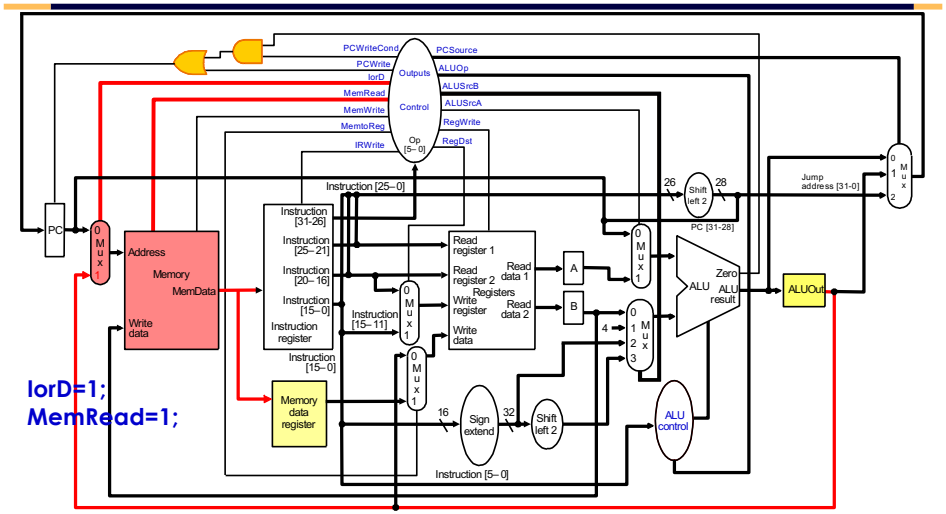
```
MDR = Memory[ALUOut];
```

- Store

```
Memory[ALUOut] = B;
```

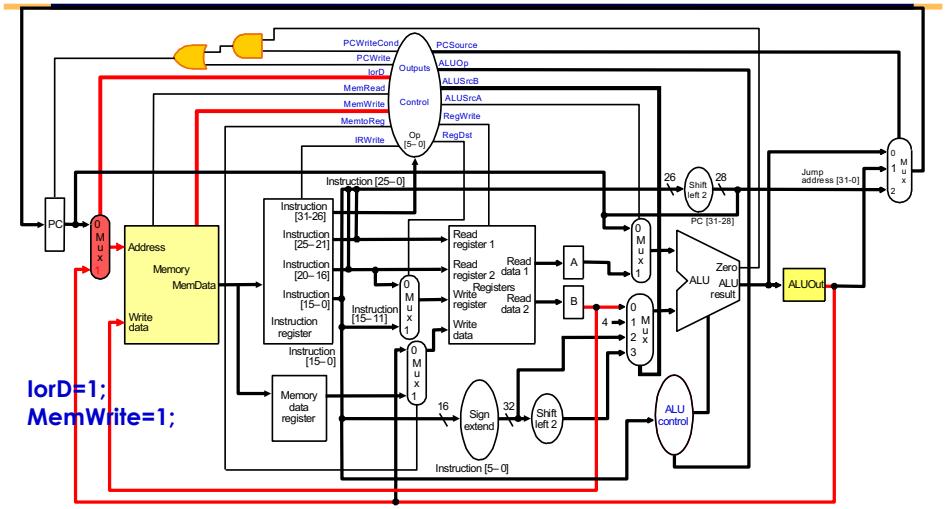

MEM: Load

MDR = Memory[ALUOut]



MEM: Store

Memory[ALUOut] = B



Write-back (WB)

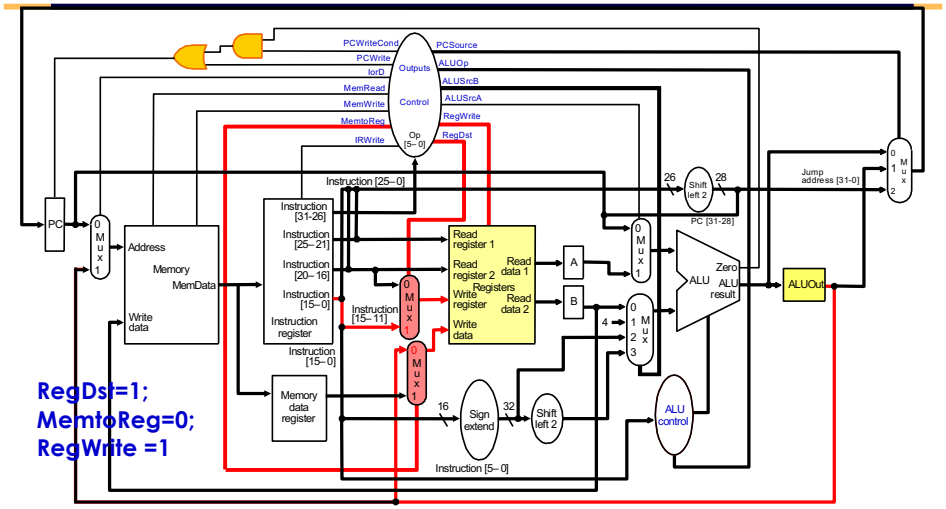
- R-type Instruction Completion

```
Reg[IR[15-11]] = ALUOut;
```

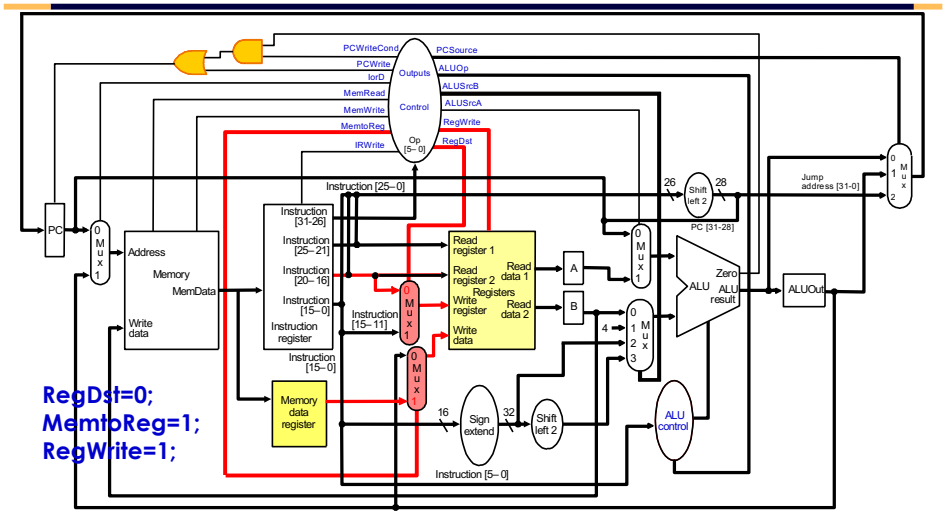
- Load writes value to Register

```
Reg[IR[20-16]]= MDR;
```


WB for R-Type Instruction



WB for Load Instruction



Summary of Actions

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch (IF)	IR = Memory[PC]			
	PC = PC + 4			
Instruction decode/ register read (ID)	A = Reg [IR[25-21]]			
	B = Reg [IR[20-16]]			
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/jump completion (EX)	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 (MEM)		Load: MDR = Memory[ALUOut], or		
		Store: Memory [ALUOut] = B		
Memory read or R-Type completion	Reg [IR[15-11]] = ALUOut	Load: Reg[IR[20-16]] = MDR		

Summary of Control Signals

Stage	Op Code	IorD	ALU Op	ALU SrcA	ALU SrcB	Reg Wr	Reg Dst	Mem Wr	Mem Rd	Mem toReg	IR Wr	PCWr Cond	PC Wr	PC Src
IF		0	00	0	01	X	X	X	1	X	1	X	1	00
ID		X	00	0	11	X	X	X	X	X	X	X	X	X
EX	ALU	X	10	1	00	X	X	X	X	X	X	X	X	X
	LW/ SW	X	00	1	10	X	X	X	X	X	X	X	X	X
	BEQ/ BNE	X	01	1	00	X	X	X	X	X	X	01	0	01
	J	X	X	X	X	X	X	X	X	X	X	X	1	10
MEM	LW	1	X	X	X	X	X	X	1	X	X	X	X	X
	SW	1	X	X	X	X	X	1	X	X	X	X	X	X
WB	ALU	X	X	X	X	1	1	X	X	0	X	X	X	X
	LW	X	X	X	X	1	0	X	X	1	X	X	X	X

Multi-Cycle Datapath Summary

- Programs take only as long as they need to
  - Variable timing per instruction
  - Pick a base cycle time
- Re-use hardware
  - Avoid unnecessary duplication of hardware

Question

- How many cycles will it take to execute this code?

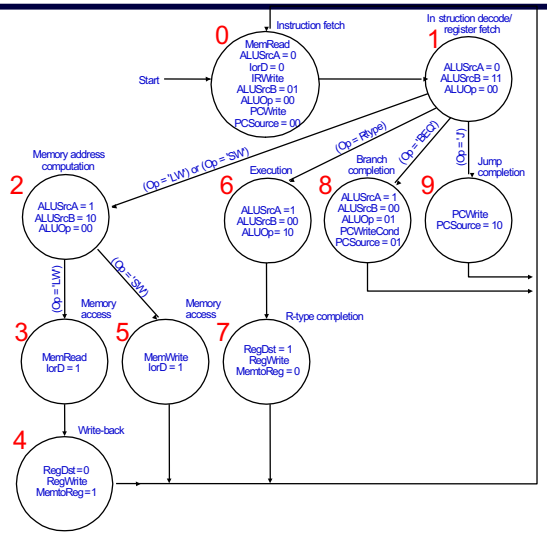
```
        lw $t2, 0($t3)
        lw $t3, 4($t3)
        beq $t2, $t3, Label     #assume not taken
        add $t5, $t2, $t3
        sw $t5, 8($t3)
Label:  ...
```

  - 21 cycles
- What is going on during the 8th cycle of execution?
  - Address for second lw instruction being computed by ALU
- In what cycle does the actual addition of `$t2` and `$t3` takes place?
  - Cycle 16

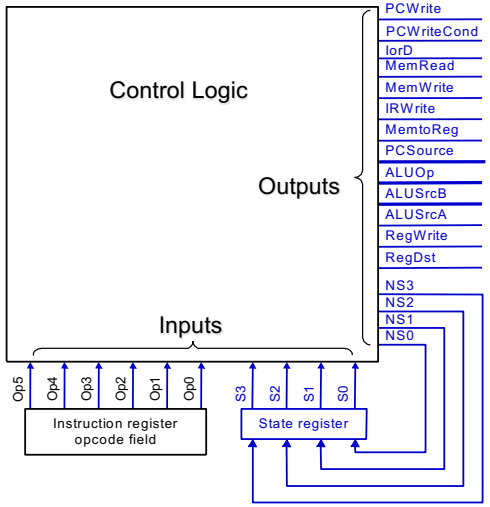
Implementing the Control

- Additional Reading: D.3
- Value of control signals is dependent upon:
  - What instruction is being executed
  - Which step is being performed
- Use the information we have accumulated to specify a finite state machine (FSM)
  - Implementation can be derived from specification
- Implementation choices
  - Specify the finite state machine graphically, or
  - Use microprogramming

Graphical Specification of FSM

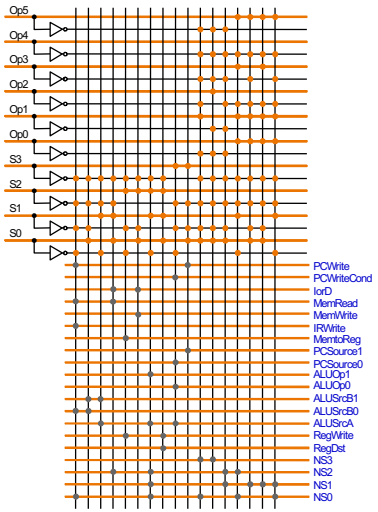


FSM for Control



Alternatives for implementing the control logic?

PLA Implementation



Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

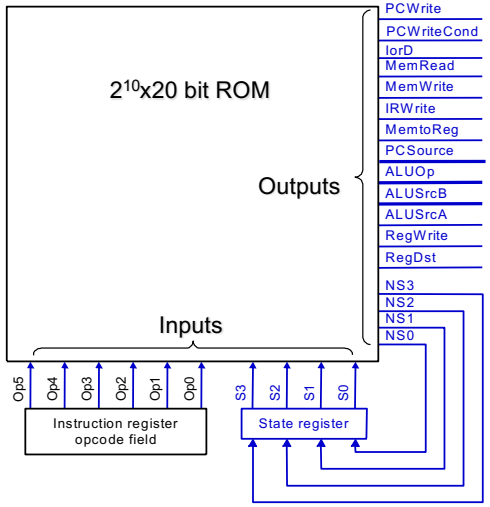
a. The truth table for the NS3 output, active when the next state is 8 or 9. This signal is activated when the current state is 1.

control signal and next state truth tables

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

ROM Implementation

Microprogramming - *Maurice Wilkes, 1954*



- Embed the control logic state table in a memory array
- Load at boot time

Microprogramming – some history

- Microprogramming thrived in 70's
  - Significantly faster ROMs than DRAMs were available
  - For complex instruction sets (CISC), datapath and controller were *cheaper and simpler*
  - *New instructions*, e.g., floating point, could be supported without datapath modifications
  - *Fixing bugs* in the controller was easier
  - ISA compatibility across various models could be achieved easily and cheaply
- Except for the cheapest and fastest machines, all computers were microprogrammed

Microprogramming – some history

- Early 80s
  - Evolution bred more complex micro-machines
  - Complex instruction sets led to the need for subroutine and call stacks in µcode
  - Need for fixing bugs in control programs was in conflict with read-only nature of µROM
  - With the advent of VLSI technology assumptions about ROM & RAM speed became invalid -> more complexity
  - Better compilers made complex instructions less important.
  - Use of numerous micro-architectural innovations, e.g., pipelining, caches and buffers, made multiple-cycle execution of reg-reg instructions unattractive

Microprogramming – 90s to today

- Microcode plays an assisting role in most modern CISC ISAs (e.g., x86 in AMD and Intel)
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine
- Patchable microcode common for post-fabrication bug fixes, e.g. Intel processors load µcode patches at bootup

Next Topic

Pipelined Datapath