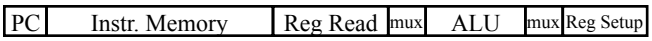
1000
Multi-Cycle CPU

ENGR 3410 – Computer Architecture

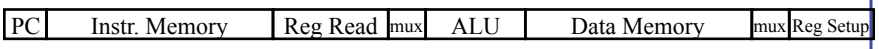
Performance of Single-Cycle Machine

CPI = 1.0, but what about cycle time? Unit reuse (I.e. adder for PC vs. ALU)

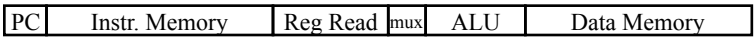
Arithmetic & Logic



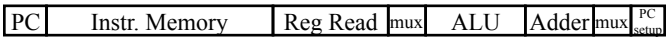
Load



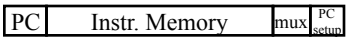
Store



Branch



Jump

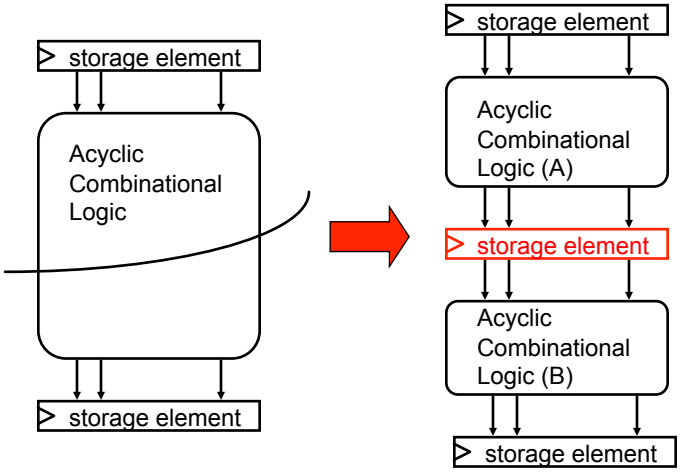


Pipelining

- Latency
- Throughput
- Exercise:
  - Start sheet of paper at one end of a table
  - Each person signs paper
  - Paper arrives at end of table signed by all
  - Start a second sheet
  - …
  - Time for one sheet, time for n sheets, time from one sheet to the next
  - Start a sheet of paper at one end of a table
  - First person signs it, passes it on
  - Second person signs first sheet, first person signs second sheet
  - …
  - Time for one sheet, time for n sheets, time from one sheet to the next

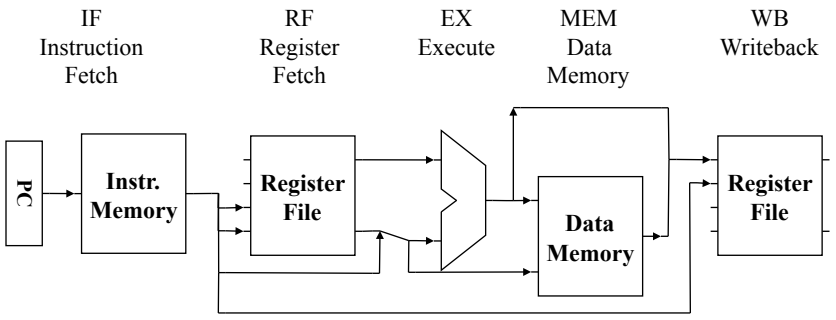
Reducing Cycle Time

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



Multicycle Processor Overview

- Divide datapath into multiple cycles



Multicycle Processor Changes

- Only one memory
  - Shared between instructions and data
- Only one ALU/adder
  - Use ALU for instructions & PC computations
- Add registers to datapath
  - IR: instruction register
  - MDR: Memory Data Register
  - A & B: Values read from register file
  - ALUout: Output of ALU

Cycle 1: Instruction Fetch

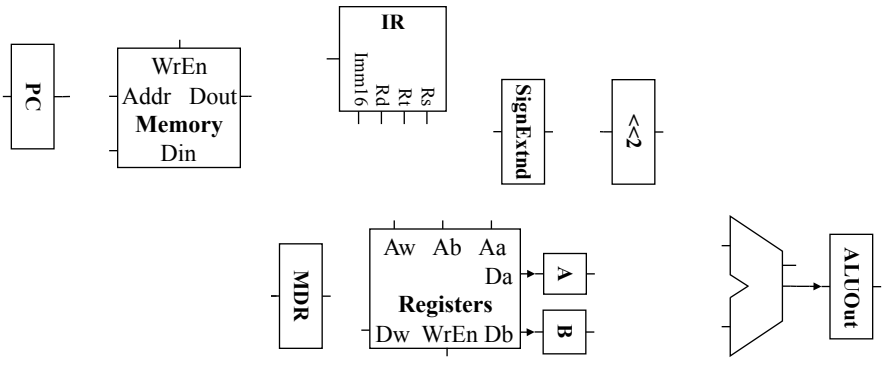
- Put the instruction to execute into the Instruction Register (IR)

  RTL:

- Set the PC to the next instruction (ignore branches)

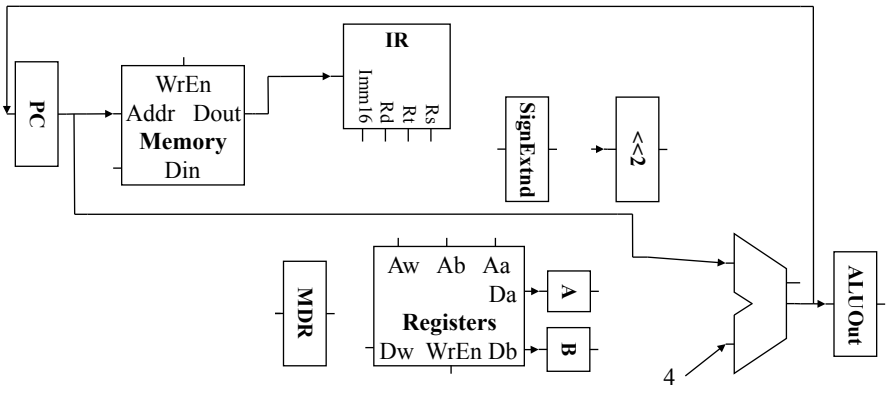
  RTL:

Cycle 1 Datapath



Cycle 1 Datapath

- IR = Mem[PC]
- PC = PC + 4



Cycle 2: Instruction Decode, Register Fetch

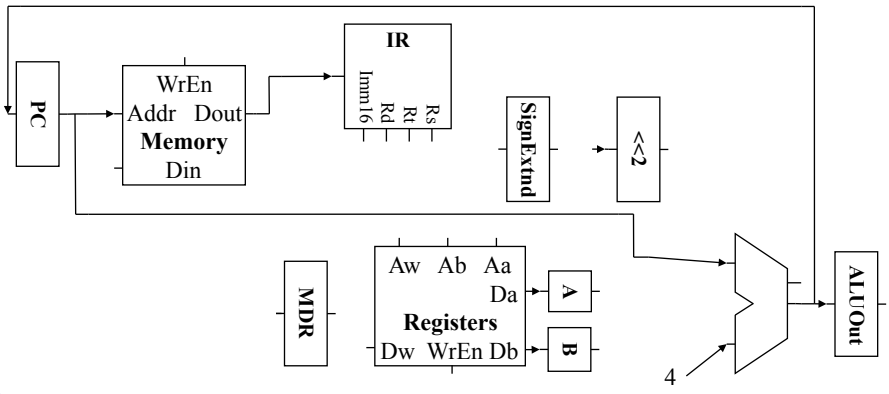
- Store the two GPR operands into registers A and B

  RTL:

- Compute Branch Target (in case it's a branch operation, won't have time later)

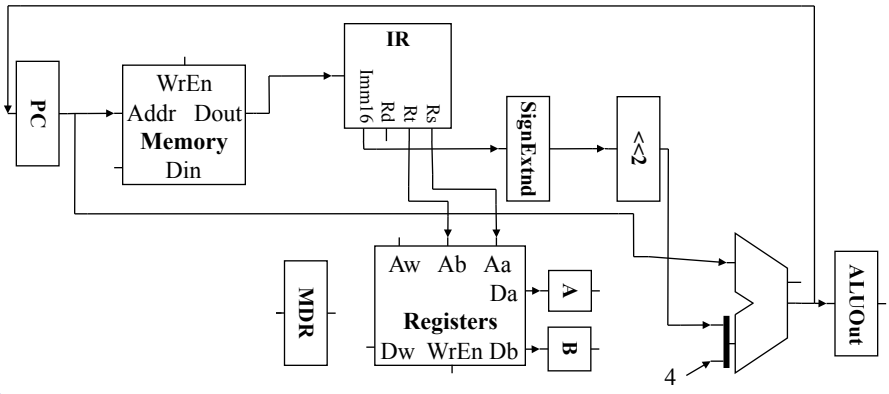
  RTL:

Cycle 1+2 Datapath



Cycle 1+2 Datapath

- A = Reg[Rs]
- B = Reg[Rt]
- ALUOut = PC + (Sign-extend(Imm16) << 2)

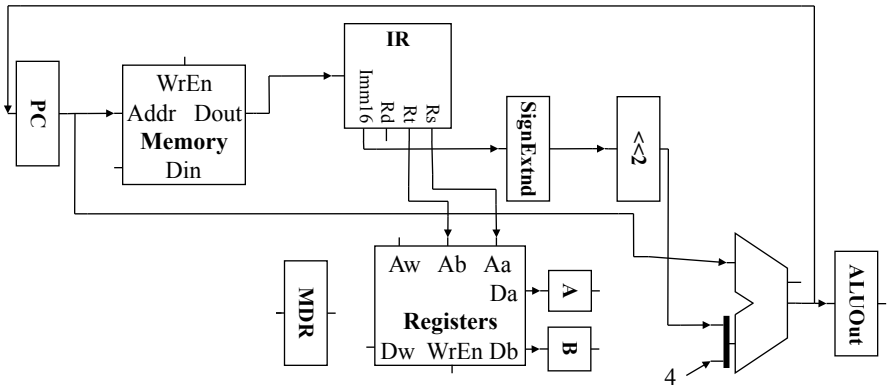


Cycle 3 (Branch)

- Branch (Beq): Branch address in ALUout. Set PC to branch address if A == B

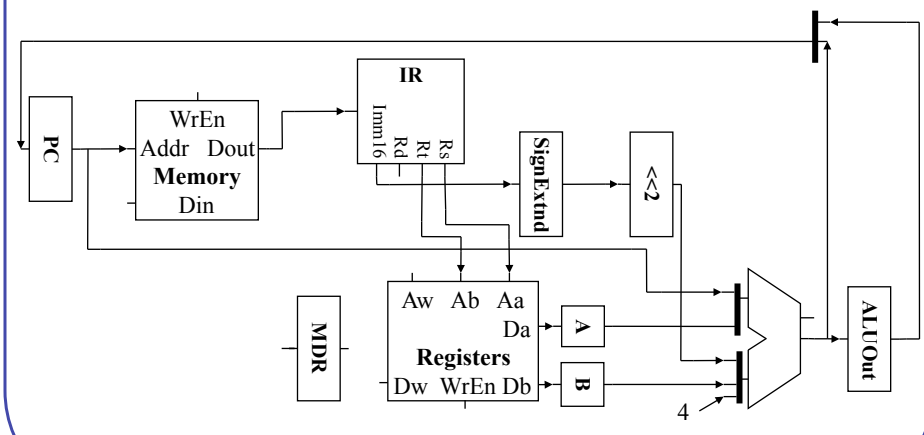
RTL:

Branch Datapath



Branch Datapath

- Zero = (A-B)
- If (zero) PC = ALUout



Cycle 3-4 (Add, Subtract)

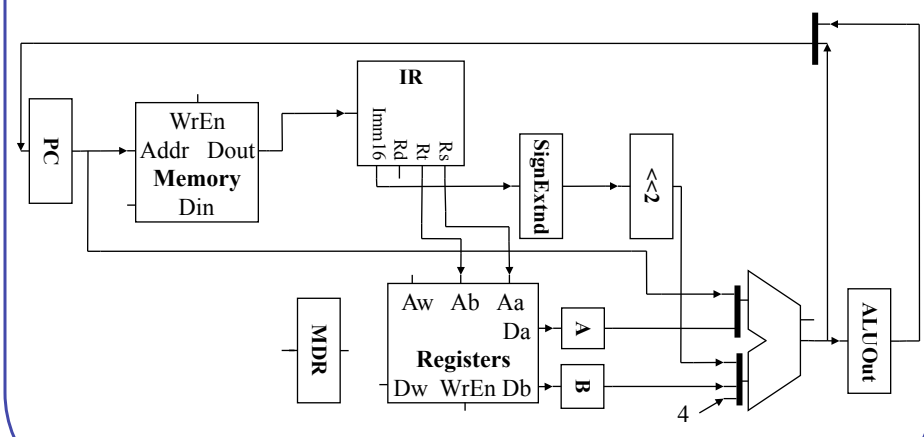
- Cycle 3: compute function in ALU, operands in A & B. Store in ALUout

  RTL:

- Cycle 4: Write value from ALUout to destination register

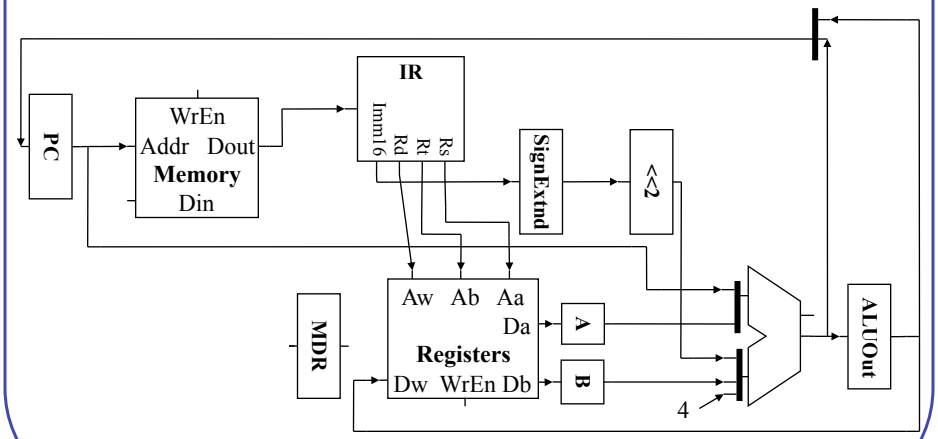
  RTL:

Branch, R-Type Datapath



Branch, R-Type Datapath

- Cycle 3: ALUout = A op B
- Cycle 4: Reg[Rd] = ALUout



Cycle 3-4 (Store)

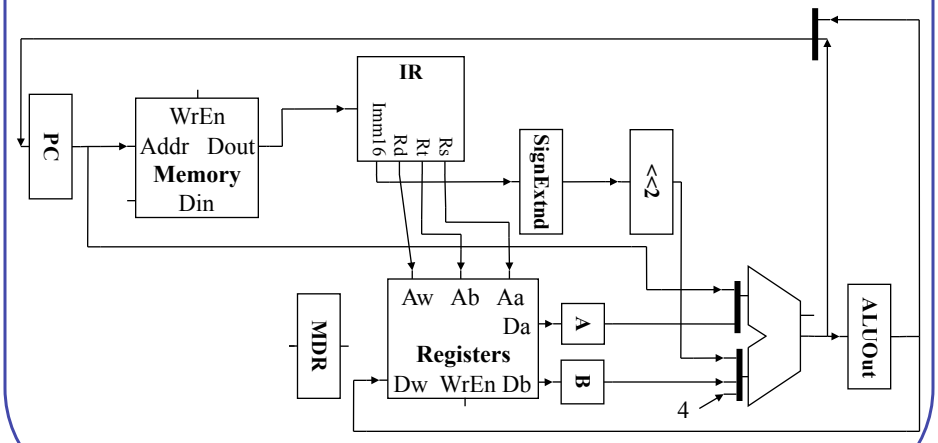
- Cycle 3: compute address from operand A and IR[15-0], put into ALUout

  RTL:

- Cycle 4: Store value from operand B to address specified in ALUout

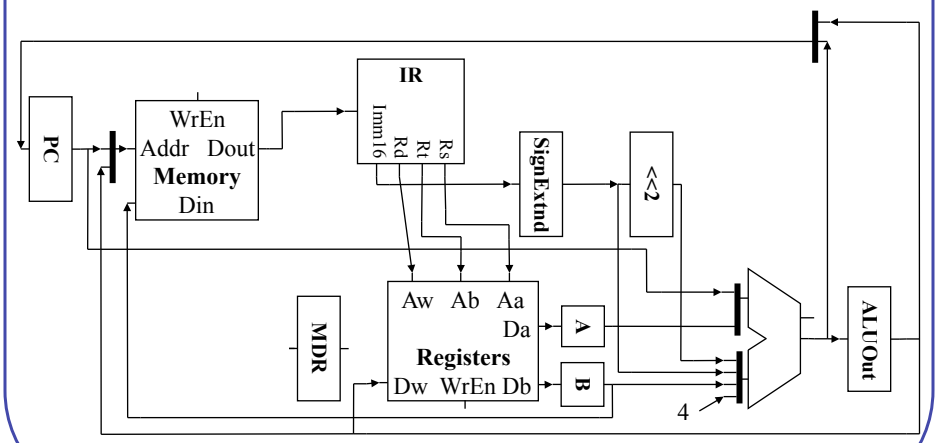
  RTL:

Branch, R-Type, Store Datapath



Branch, R-Type, Store Datapath

- Cycle 3: ALUout = A + sign-extend(Imm16)
- Cycle 4: Mem[ALUout] = B



Cycle 3-5 (Load)

- Cycle 3: compute address from operand A and IR[15-0], put into ALUout

  RTL:

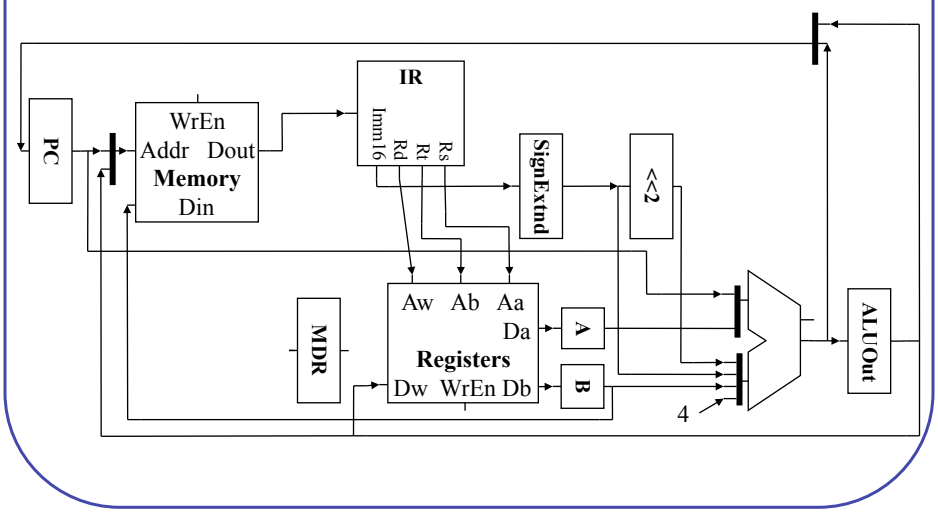
- Cycle 4: Load value from address specified in ALUout to MDR

  RTL:

- Cycle 5: Write value from MDR to destination register

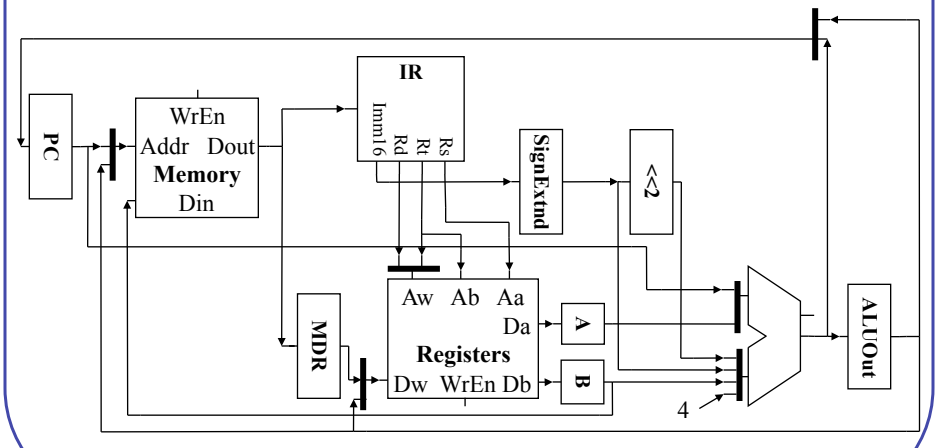
  RTL:

Multicycle Processor Datapath



Multicycle Processor Datapath

- Cycle 3: ALUout = A + sign-extend(Imm16)
- Cycle 4: MDR = Mem[ALUout]
- Cycle 5: Reg[Rt] = MDR



Multicycle Processor Control

- Need to control data path to perform required operations
  - Multiple cycles w/different control values each cycle, so control is an FSM.

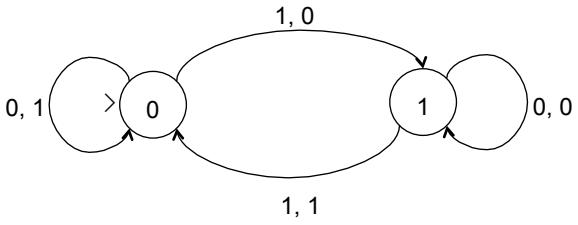
```
Cycle 1:         IR = Mem[PC]; PC = PC + 4
Cycle 2:         A = Reg[RS]; B = Reg[Rt];
                 ALUOut = PC + (Sign-extend(Imm16)<<2)
Cycle 3 Branch: Zero = (A-B); If (zero) PC = ALUout
Cycle 3 R-Type: ALUout = A op B
Cycle 4 R-Type: Reg[Rd] = ALUout
Cycle 3 Store:  ALUout = A + sign-extend(Imm16)
Cycle 4 Store:  Mem[ALUout] = B
Cycle 3 Load:   ALUout = A + sign-extend(Imm16)
Cycle 4 Load:   MDR = Mem[ALUout]
Cycle 5 Load:   Reg[Rt] = MDR
```

Finite State Control

Finite state machine:

InputAlphabet x OutputAlphabet x States x InitialState x TransitionFunction x OutputFunction

① Starts in input state
② Receives input
③ Produces output. (If output depends only on state, Moore Machine. If output depends on state and input, Mealy machine.)
④ Based on current state and input, transition to next state
⑤ Go to Step 2

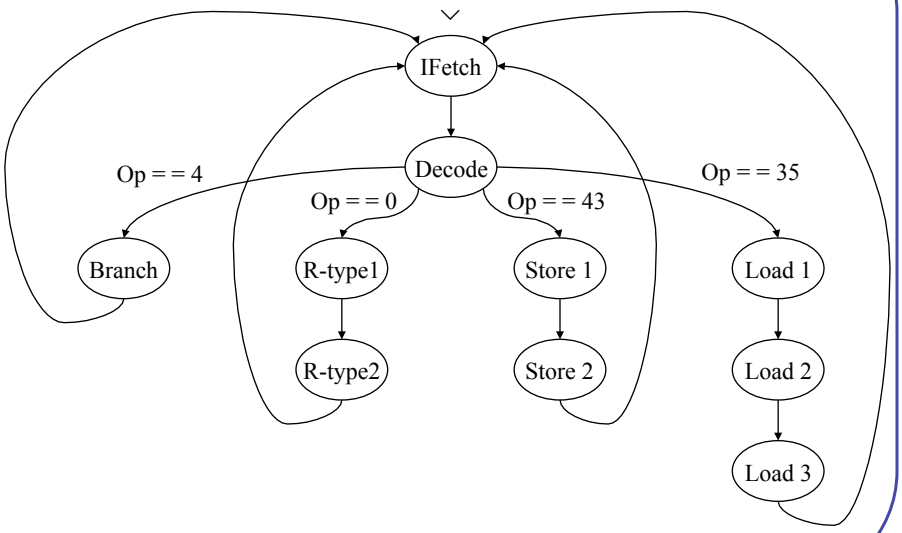


Control FSM

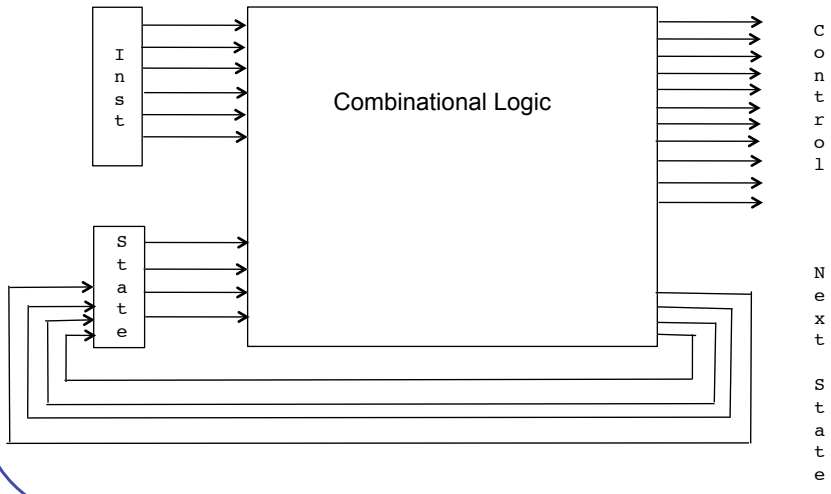
- Opcodes: `LW 35, SW 43, BEQ 4, add/sub 0`

Control FSM

- Opcodes: `LW 35, SW 43, BEQ 4, add/sub 0`



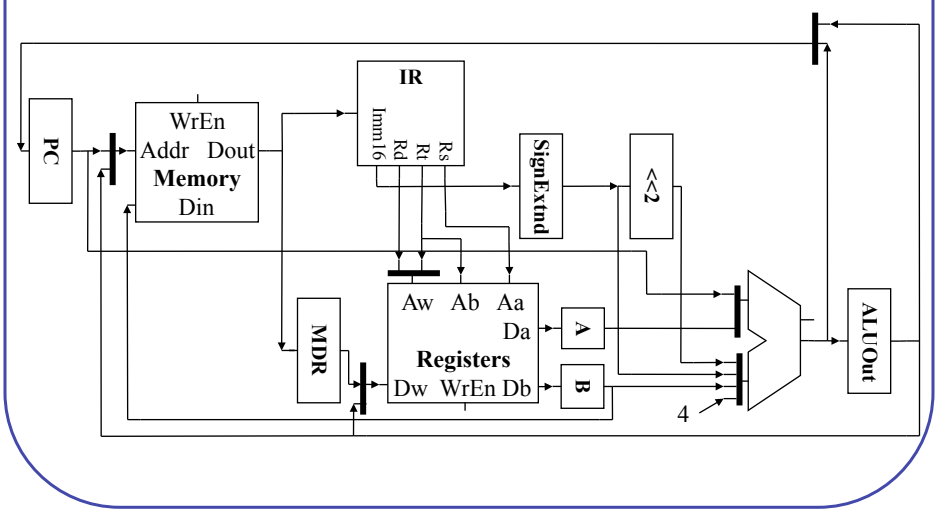
Implementing Finite State Control



Combinational Logic Choices

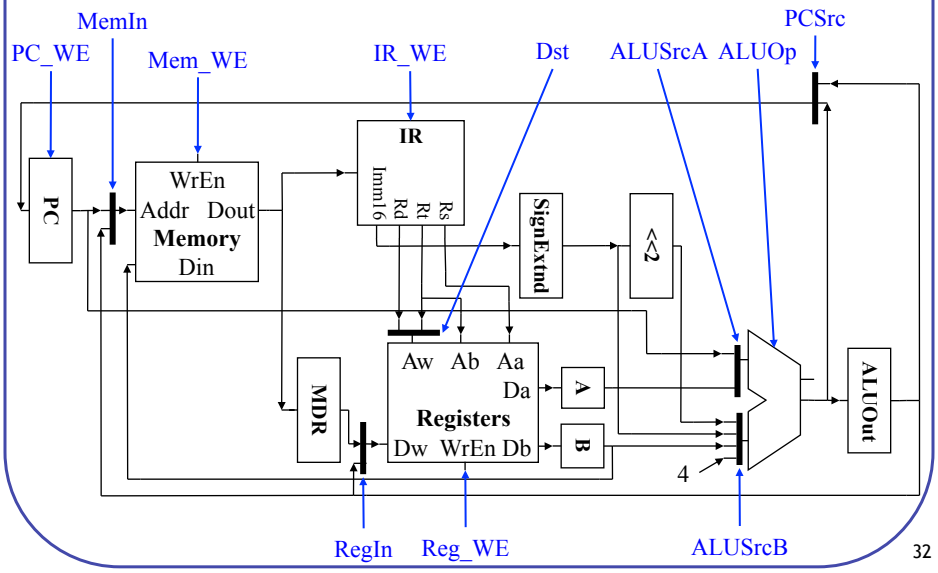
- Discreet logic: compute each output bit and next state bit
- 2-level logic device: PLA
- ROM: A direct implementation of a complete truth table. Append input bits and state bits and wire to address pins of ROM. Wire output bits to control and next state buses. Store the truth table in the ROM.

Datapath Control Signals



Datapath Control Signals

•



Multicycle Datapath Control

	WE				ALU						
State	PC	Mem	Reg	IR	SrcA	SrcB	Op	Dest	MemIn	RegIn	PCSrc
1											
2											
3Br											
3Rt											
4Rt											
3St											
4St											
3Lo											
4Lo											
5Lo											
					A	<<2		Rt[20:16]	PC	MDR	ALU
					PC	SE		Rd[15:11]	ALUout	ALUout	ALUout
						B					
						4					

```
1:
IR = Mem[PC]
PC = PC + 4

2:
A = Reg[Rs]
B = Reg[Rt]
ALUOut =

   PC + (SE(imm16) << 2)

3Br:
Zero = (A-B)
If (zero) PC = ALUout

3Rt:
ALUout = A op B

4Rt:
Reg[Rd] = ALUout

3St:
ALUout = A + SE(Imm16)

4St:
Mem[ALUout] = B

3Lo:
ALUout = A + SE(Imm16)

4Lo:
MDR = Mem[ALUout]

5Lo:
Reg[Rt] = MDR
```

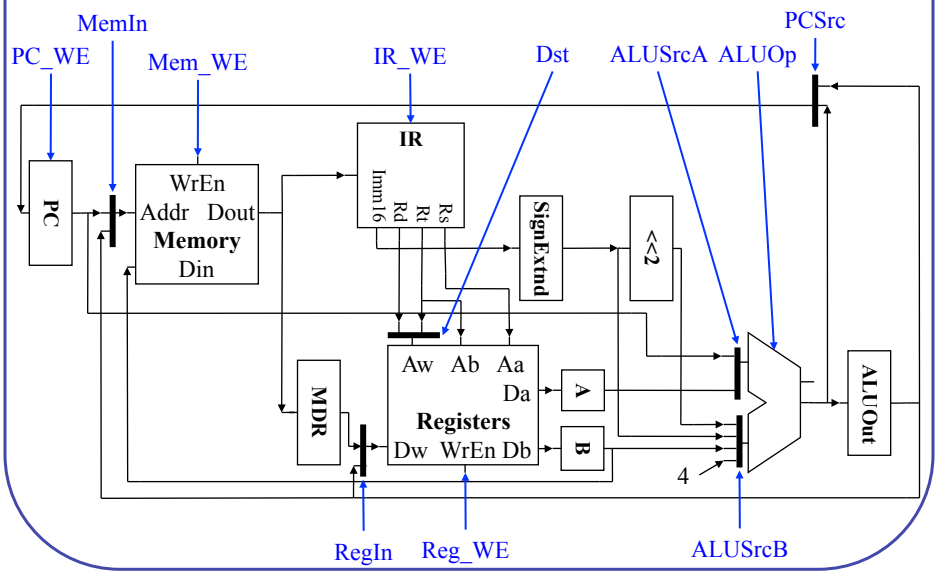
Multicycle Datapath Control

	WE				ALU						
State	PC	Mem	Reg	IR	SrcA	SrcB	Op	Dest	MemIn	RegIn	PCSrc
1	1	0	0	1	PC	4	+	X	PC	X	ALU
2	0	0	0	0	PC	<<2	+	X	X	X	X
3Br	(zero)	0	0	0	A	B	-	X	X	X	ALUout
3Rt	0	0	0	0	A	B	(IR)	X	X	X	X
4Rt	0	0	1	0	X	X	X	Rd	X	ALUout	X
3St	0	0	0	0	A	SE	+	X	X	X	X
4St	0	1	0	0	X	X	X	X	ALUout	X	X
3Lo	0	0	0	0	A	SE	+	X	X	X	X
4Lo	0	0	0	0	X	X	X	X	ALUout	X	X
5Lo	0	0	1	0	X	X	X	Rt	X	MDR	X
					A	<<2		Rt[20:16]	PC	MDR	ALU
					PC	SE		Rd[15:11]	ALUout	ALUout	ALUout
						B					
						4					

```
1:
IR = Mem[PC]
PC = PC + 4

2:
A = Reg[Rs]
B = Reg[Rt]
ALUOut = PC
  + (SE(imm16)<<2)

3Br:
Zero = (A-B)
If (zero) PC = ALUout

3Rt:
ALUout = A op B

4Rt:
Reg[Rd] = ALUout

3St:
ALUout = A + SE(Imm16)

4St:
Mem[ALUout] = B

3Lo:
ALUout = A + SE(Imm16)

4Lo:
MDR = Mem[ALUout]

5Lo:
Reg[Rt] = MDR
```

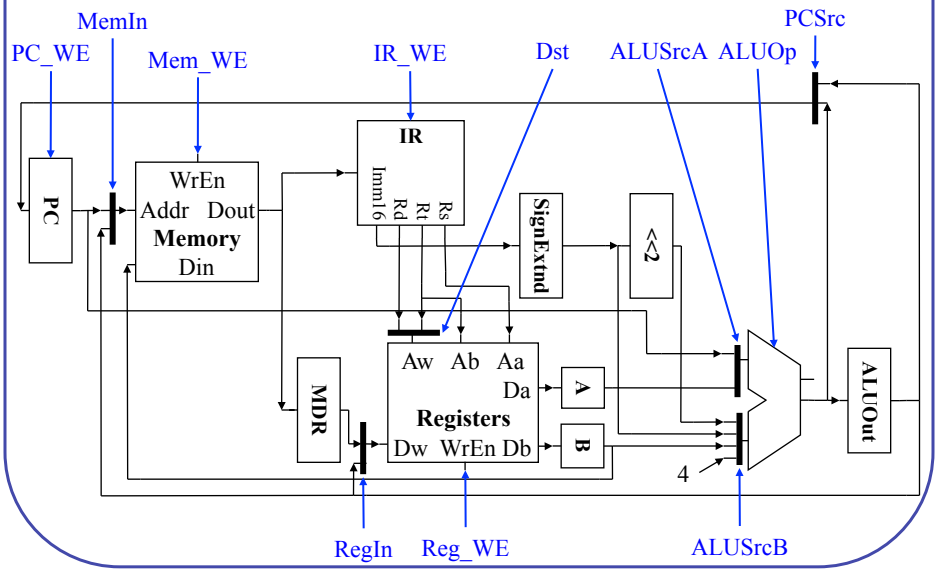
Cycle 1 Control

- `IR = Mem[PC]; PC = PC + 4`



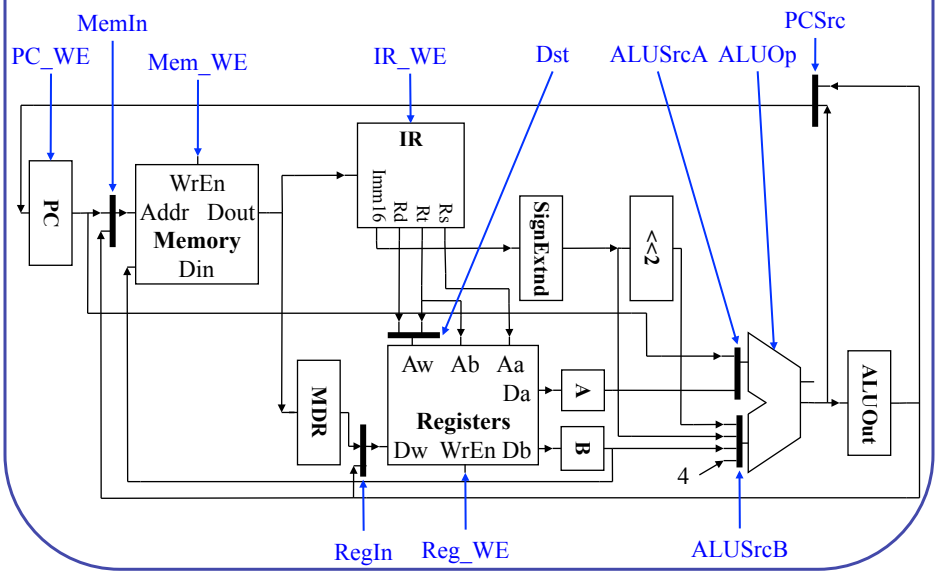
Cycle 2 Control

- `A=Reg[Rs]; B=Reg[Rt]; ALUOut = PC+(Sign-extend(Imm16) <<2 )`



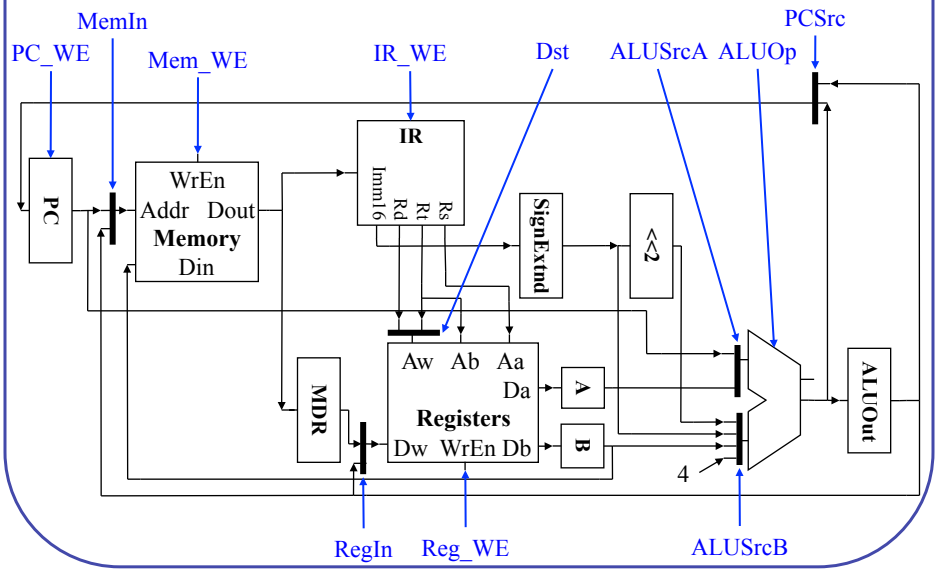
Cycle 3 (Branch) Control

- `Zero = (A-B); If (zero) PC = ALUout`



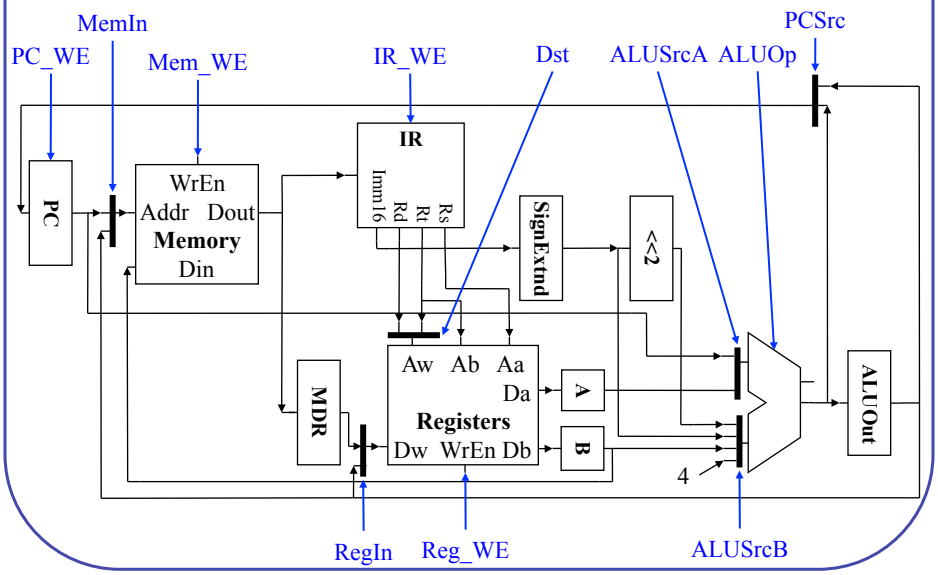
Cycle 3 (R-Type) Control

- `ALUout = A op B`



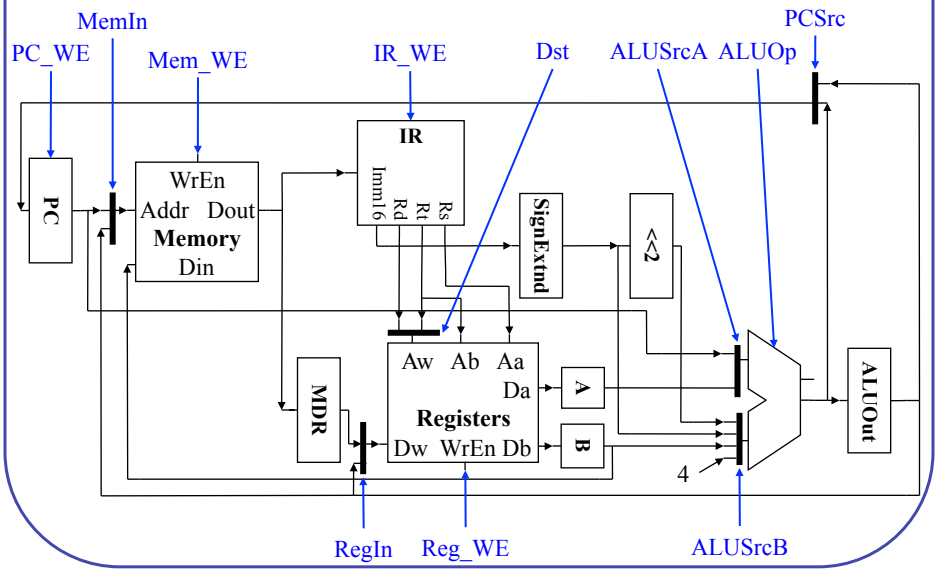
Cycle 4 (R-Type) Control

- `Reg[Rd] = ALUout`



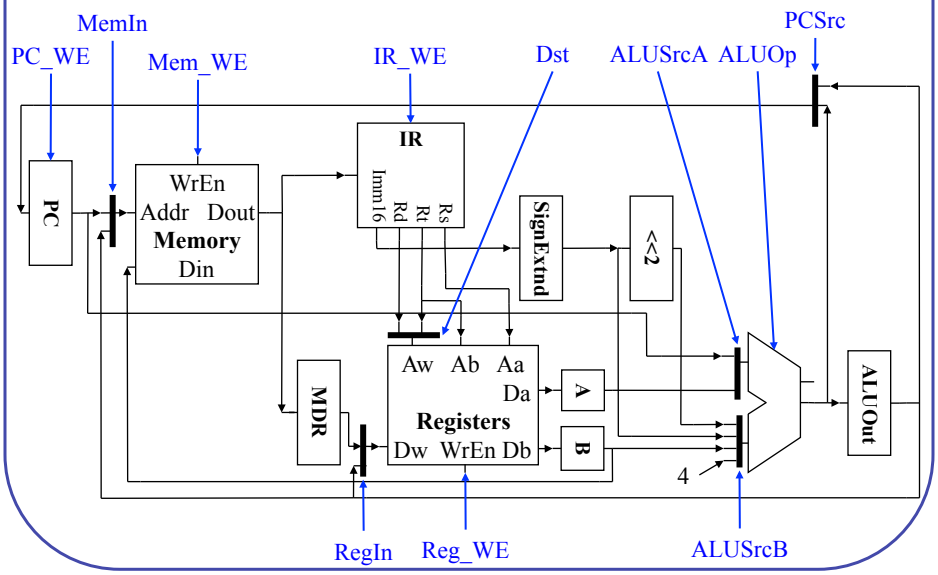
Cycle 3 (Store) Control

- `ALUout = A + sign-extend(Imm16)`



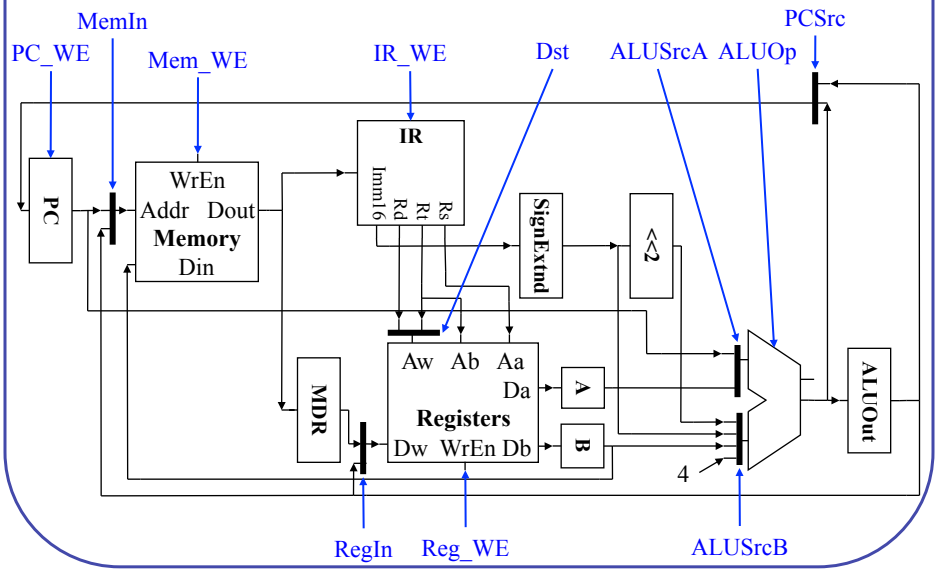
Cycle 4 (Store) Control

- `Mem[ALUout] = B`



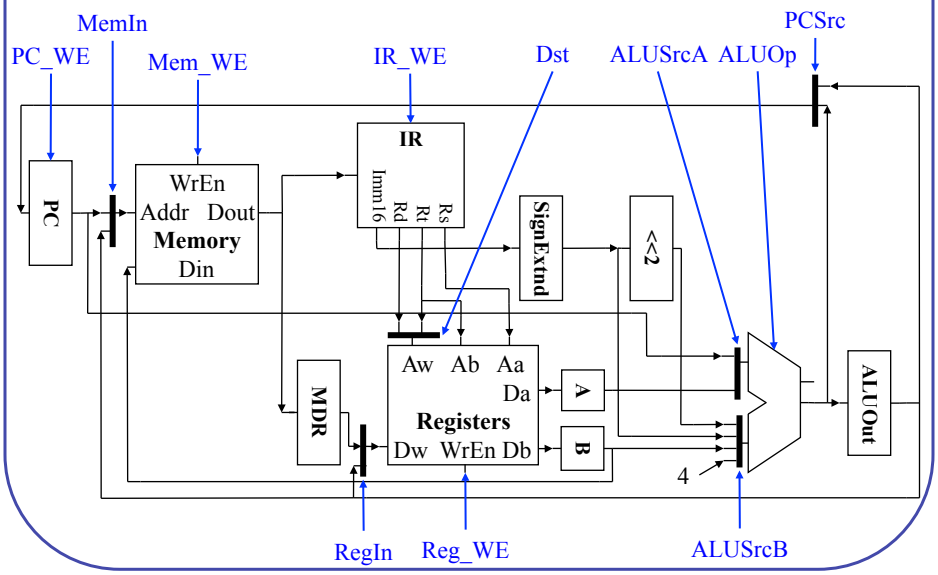
Cycle 3 (Load) Control

- `ALUout = A + sign-extend(Imm16)`



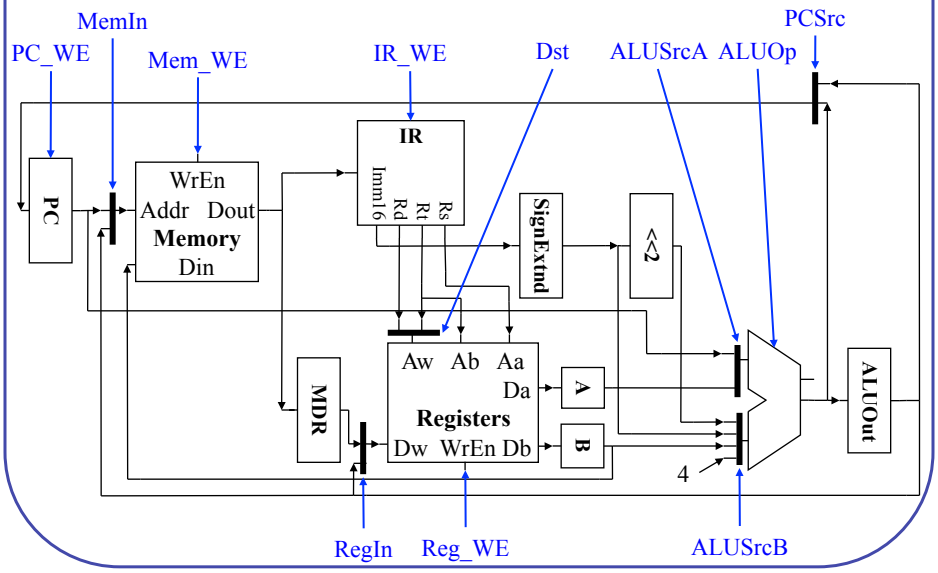
Cycle 4 (Load) Control

- `MDR = Mem[ALUout]`

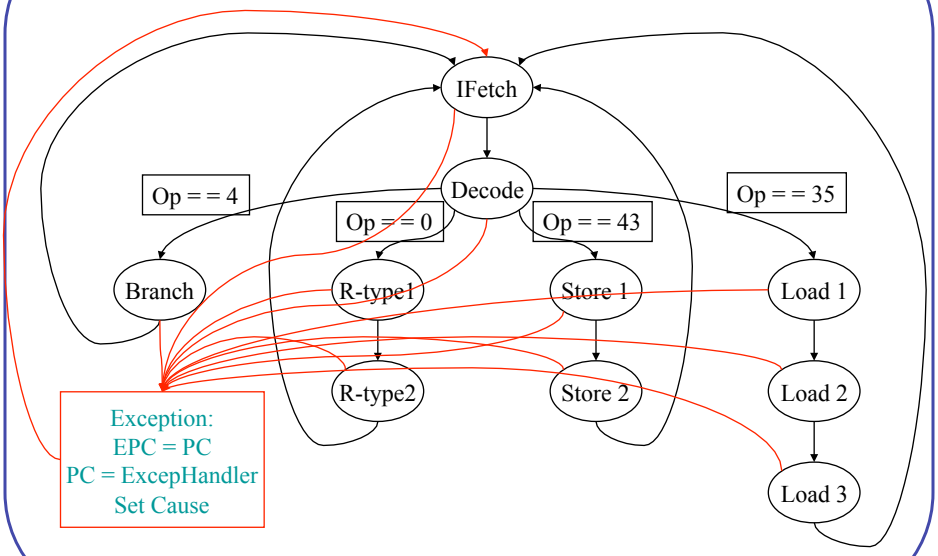


Cycle 5 (Load) Control

- `Reg[Rt] = MDR`



Advanced: Exceptions (cont.)



Multicycle CPI

- Compute the CPI of the machine, given the frequencies specified

$$CPI = \sum_{types} (Cycles_{type} * Frequency_{type})$$

Instruction Type	Type Cycles	Type Frequency	Cycles * Freq
ALU		50%	
Load		20%	
Store		10%	
Branch		20%	
		CPI:	

Multicycle CPI

- Compute the CPI of the machine, given the frequencies specified

$$CPI = \sum_{types} (Cycles_{type} * Frequency_{type})$$

Instruction Type	Type Cycles	Type Frequency	Cycles * Freq
ALU	4	50%	2.0
Load	5	20%	1.0
Store	4	10%	0.4
Branch	3	20%	0.6
		CPI:	4.0

Multicycle Summary

- By splitting the single-cycle datapath up we achieve:

Multicycle Summary

- By splitting the single-cycle datapath up we achieve:
  - Faster clock cycle
  - Variable duration instructions
  - (big CPI, but clock cycle more than compensates)
  - Hardware reuse (ALU, Memory)
- But, more complex control