Computer Architecture

4. The Processor: Datapath and Control

Young Geun Kim
(younggeun_kim@korea.ac.kr)
Intelligent Computer Architecture & Systems Lab.

Single Cycle Processor

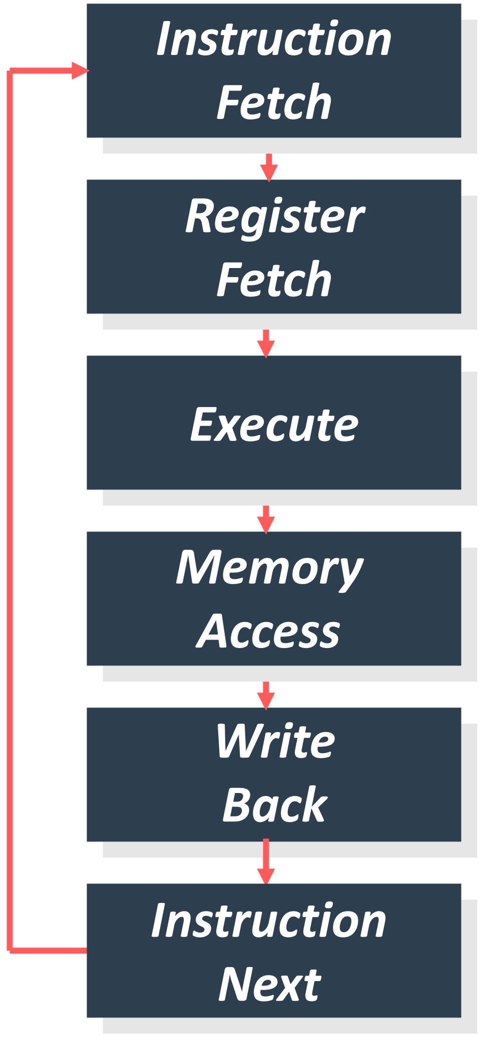
- **Advantages**
  - Single cycle per instruction makes logic and clock simple
- **Disadvantages**
  - Cycle time is determined by the worst-case path
    - Critical path: load instruction
      - Instruction memory -> register file –> ALU -> data memory -> register file
  - Not feasible to adapt to different instructions
    - Different instruction can have different length of time
  - Inefficient utilization of memory and functional units
- **We will improve performance with different approaches**

To Mitigate the Disadvantages

- **Multicycle Implementation**
  - Divide each instruction into a series of steps
  - Each step will take one clock cycle
  - Different instructions can have different CPI
- **Requires a few significant changes to organization**
  - Use registers to separate each stage
- **Example**
  - R-format instruction (4 cycles)
    - (1) Instruction fetch (2) Instruction decode/register fetch (3) ALU operation (4) Register write
  - Load instruction (5 cycles)
    - (1) Instruction fetch (2) Instruction decode/register fetch (3) Address computation (4) Memory read (5) Register write

Multicycle Divisions

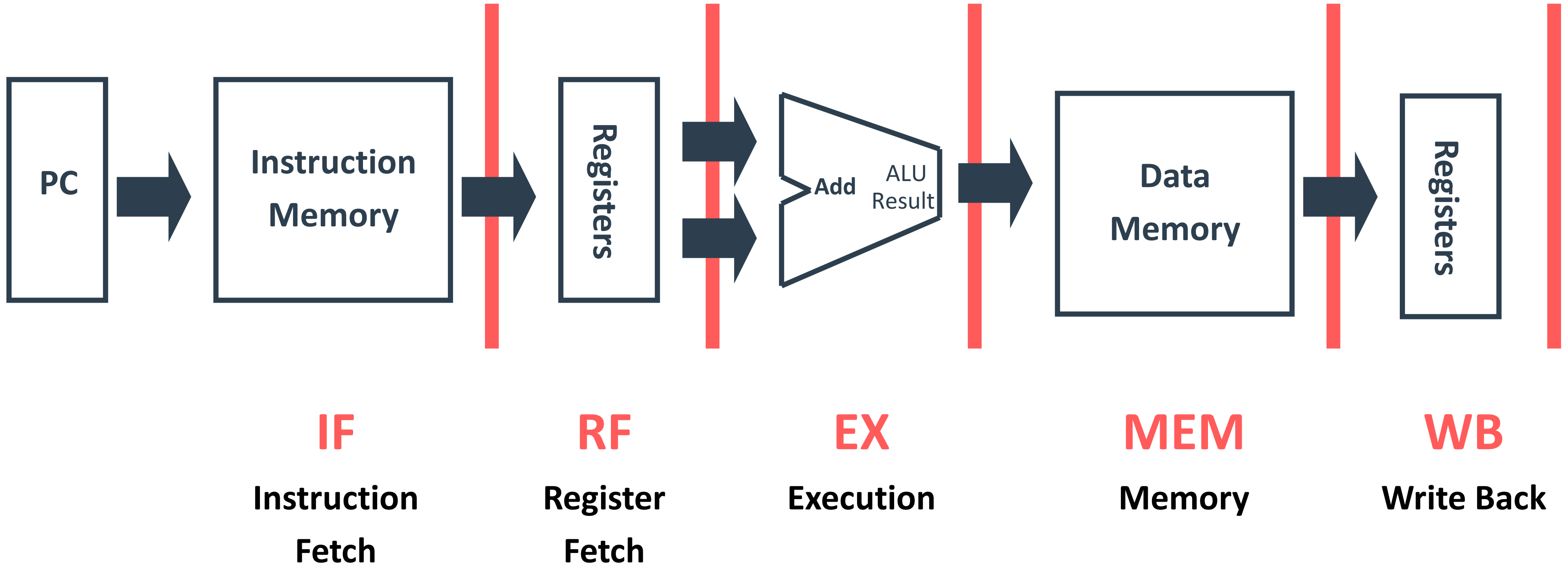
- Divide datapath into steps (1 cycle each)
- Instructions can range from 3 - 5 stages in our multicycle



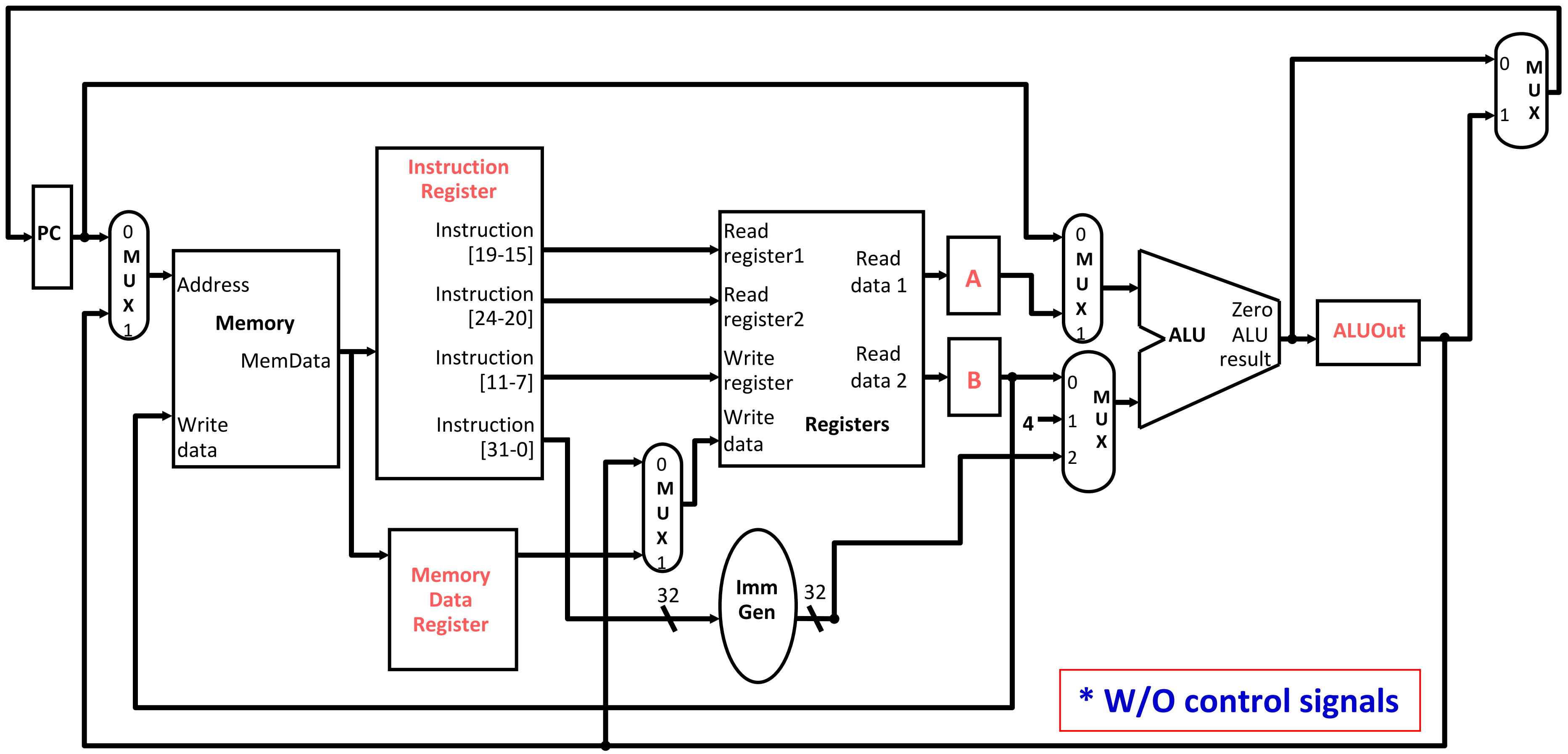
- *Obtain instruction from instruction memory*
- *Locate and obtain operand data*
- *Compute result value or status*
- *Load and/or store values from/to memory*
- *Deposit results in storage for later use*
- *Determine the following instruction*

Multicycle Divisions (Cont'd)

- From datapath point of view



Multicycle Implementation Overview



* W/O control signals

- **Single memory unit for instructions and data**
  - Registers used to store output during instruction execution
- **Single ALU used for arithmetic/logic, memory addresses, and next instructions**

Specific Changes

- **A $\boxed{\text{single memory}}$ unit is used for both instructions and data**
- **There is a $\boxed{\text{single ALU}}$ rather than an ALU and additional adder**

  single cycle 과 달리, 추가적인 Adder가 필요 없음.

- **One or more registers are added after every major functional unit to hold output**
  - Instruction register
    - Needs to have Write line since instruction kept across multiple cycles
  - Memory data register (MDR)
  - A and B registers
  - ALUOut register

Stage Description of Multicycle Design

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

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

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step1: Instruction Fetch

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

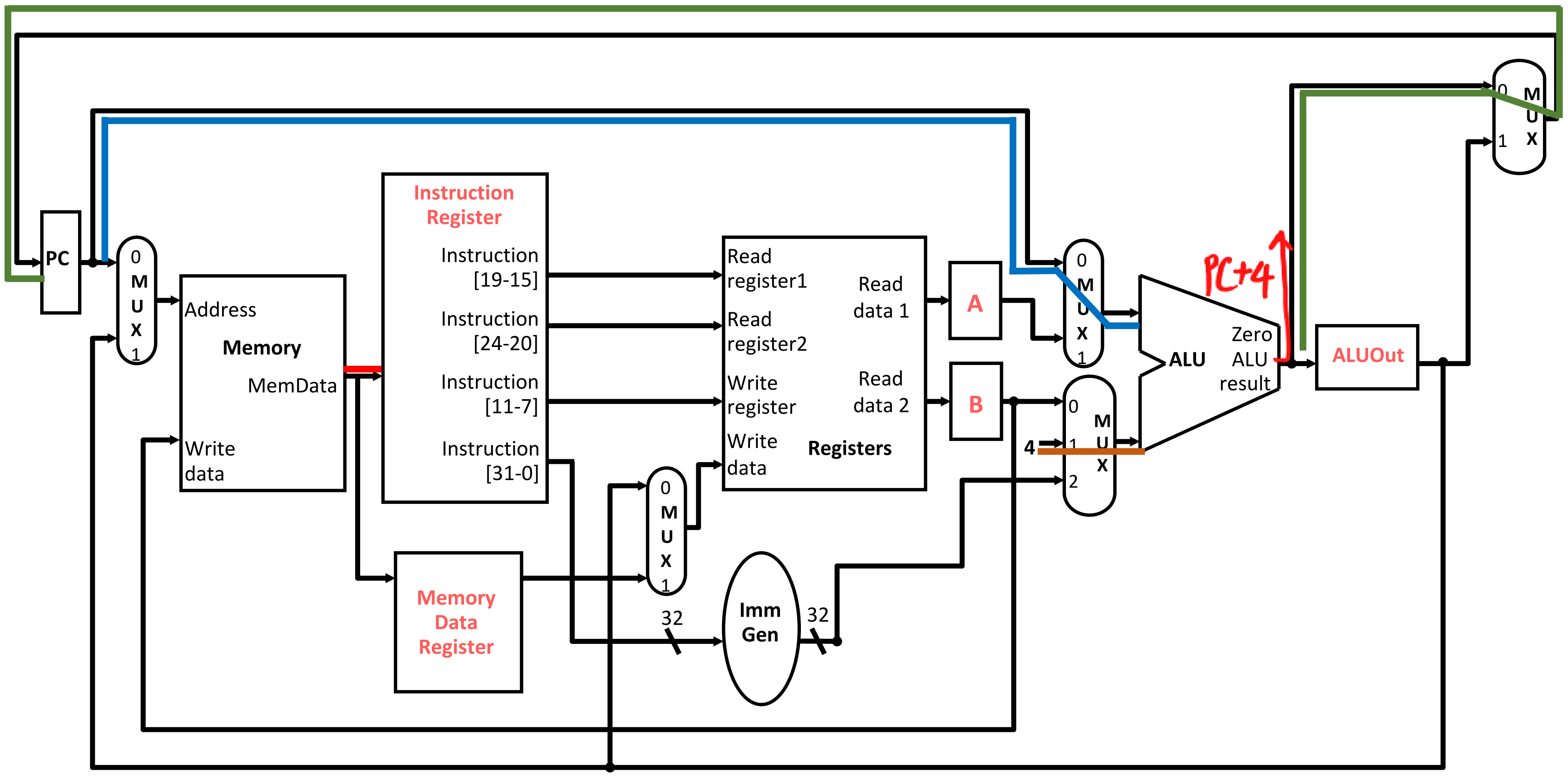
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step1: Instruction Fetch (Cont'd)

- **RTL Description**
  IR = Mem[PC]
  PC = PC + 4



Step2: Instruction Decode and Register Fetch

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

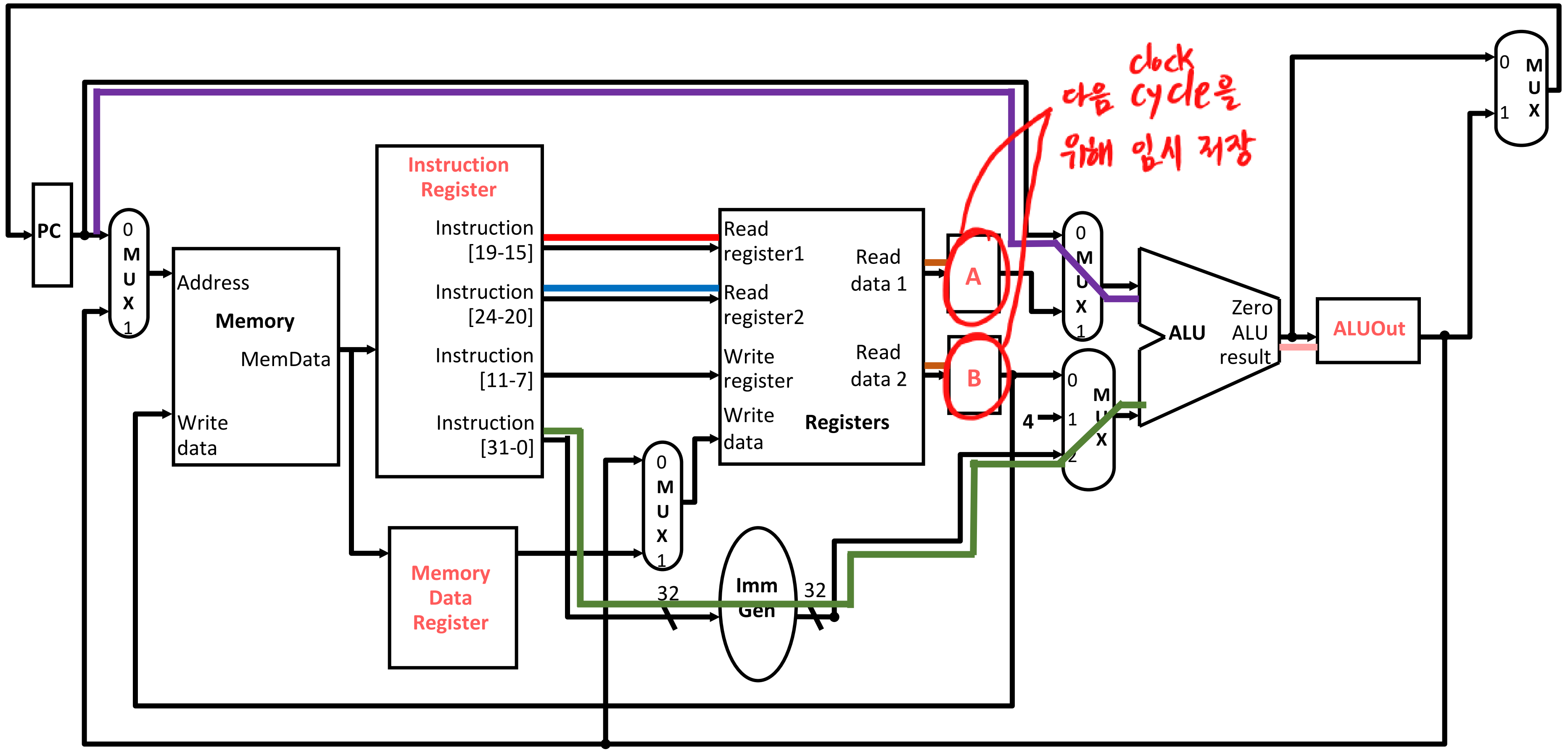
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step2: Instruction Decode and Register Fetch (Cont'd)

- **RTL Description**
  A = Reg[IR[19-15]]
  B = Reg[IR[24-20]]
  ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))



Step2 Special Note

- **Control Lines**
  - Not dependent on instruction type
  - Instruction is still being decoded at this step
  - ALU used to calculate branch destination just in case we decode a branch instruction

Step3: R-Type Execution

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

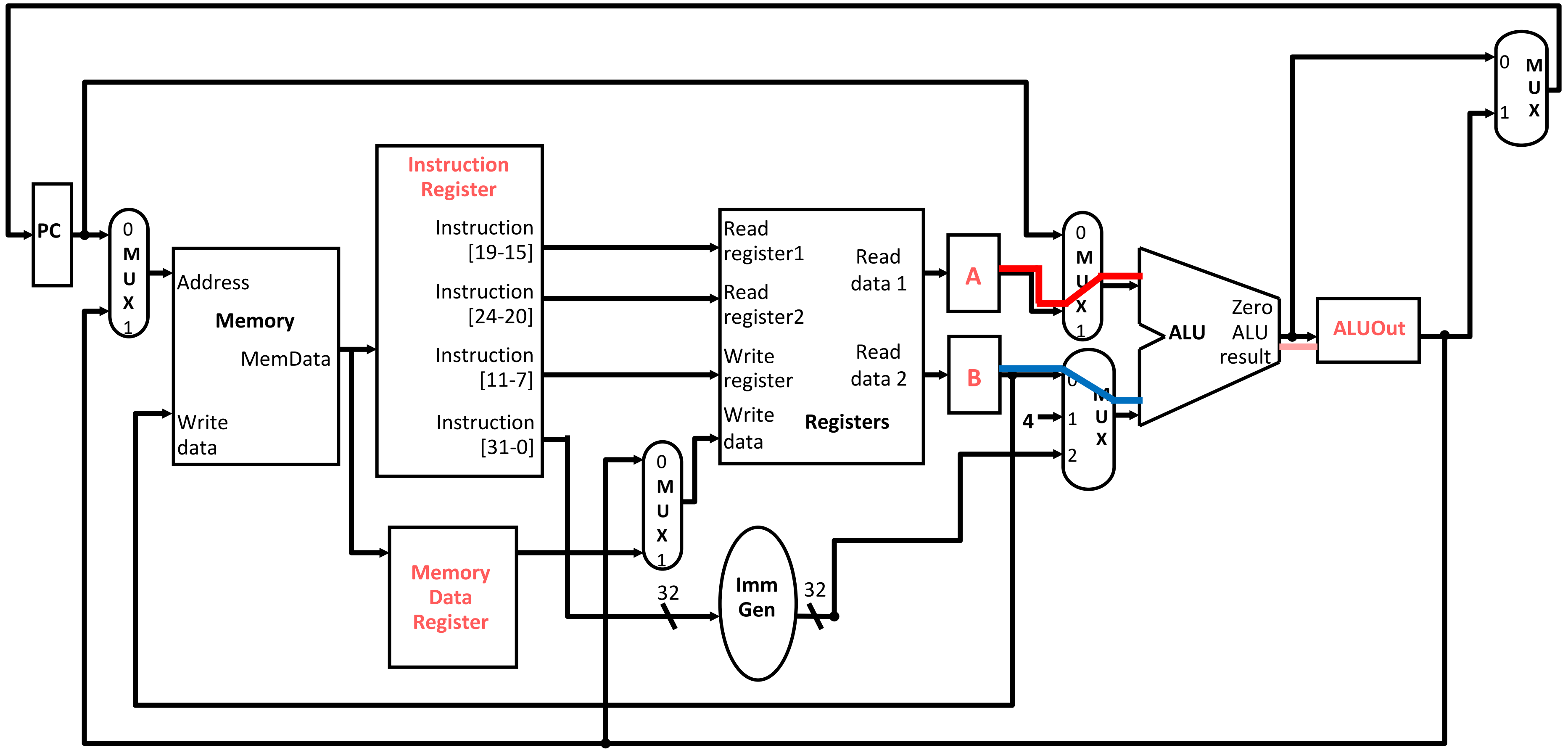
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/
Register Fetch
	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step3: R-Type Execution

- **RTL Description**
  ALUOut = A op B



Step4: R-Type Completion Step

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

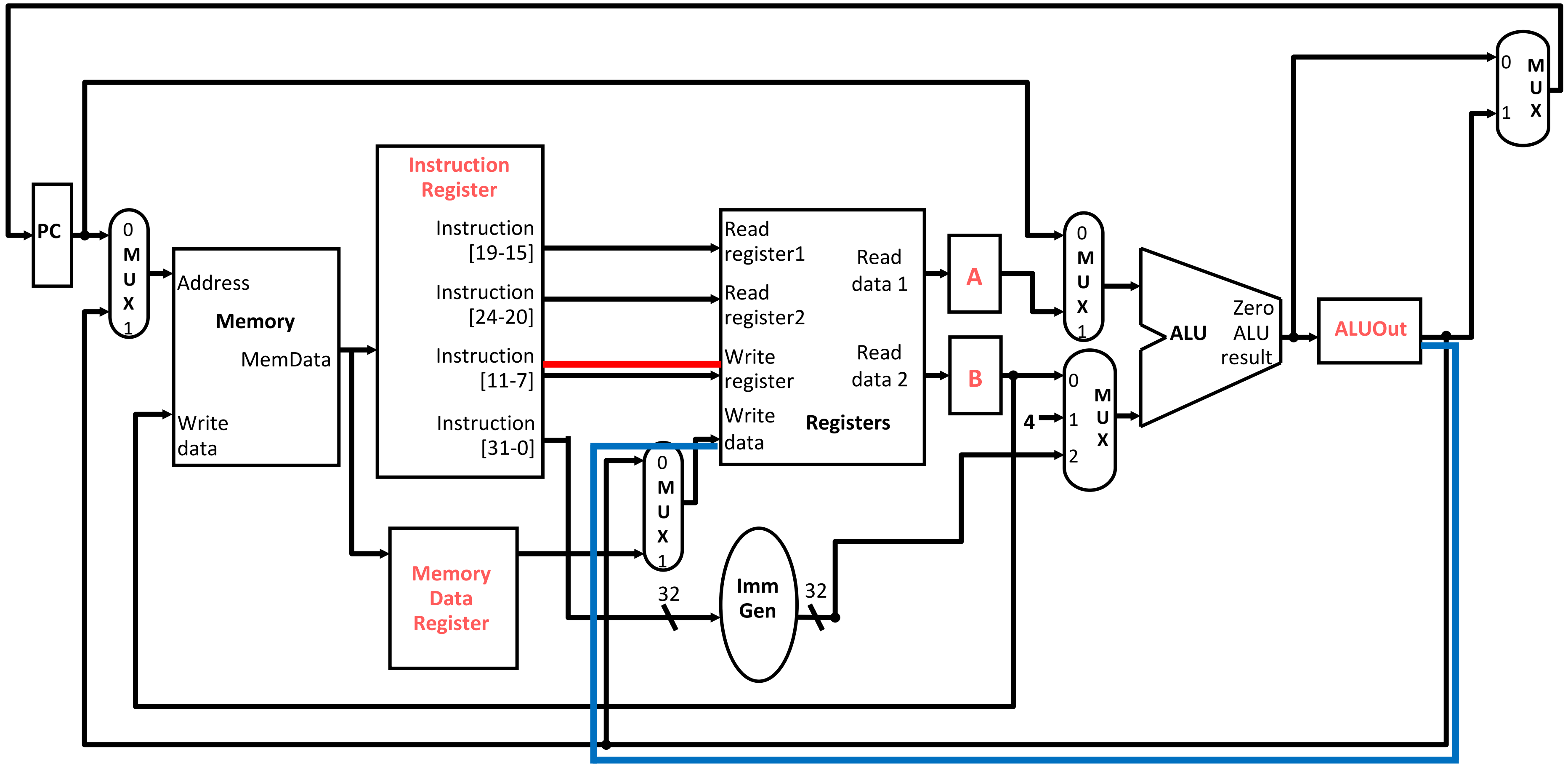
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step4: R-Type Completion Step

- **RTL Description**
  Reg[IR[11-7]] = ALUOut



Step3: Branch Completion Step

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

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

Instruction Decode/
Register Fetch
	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

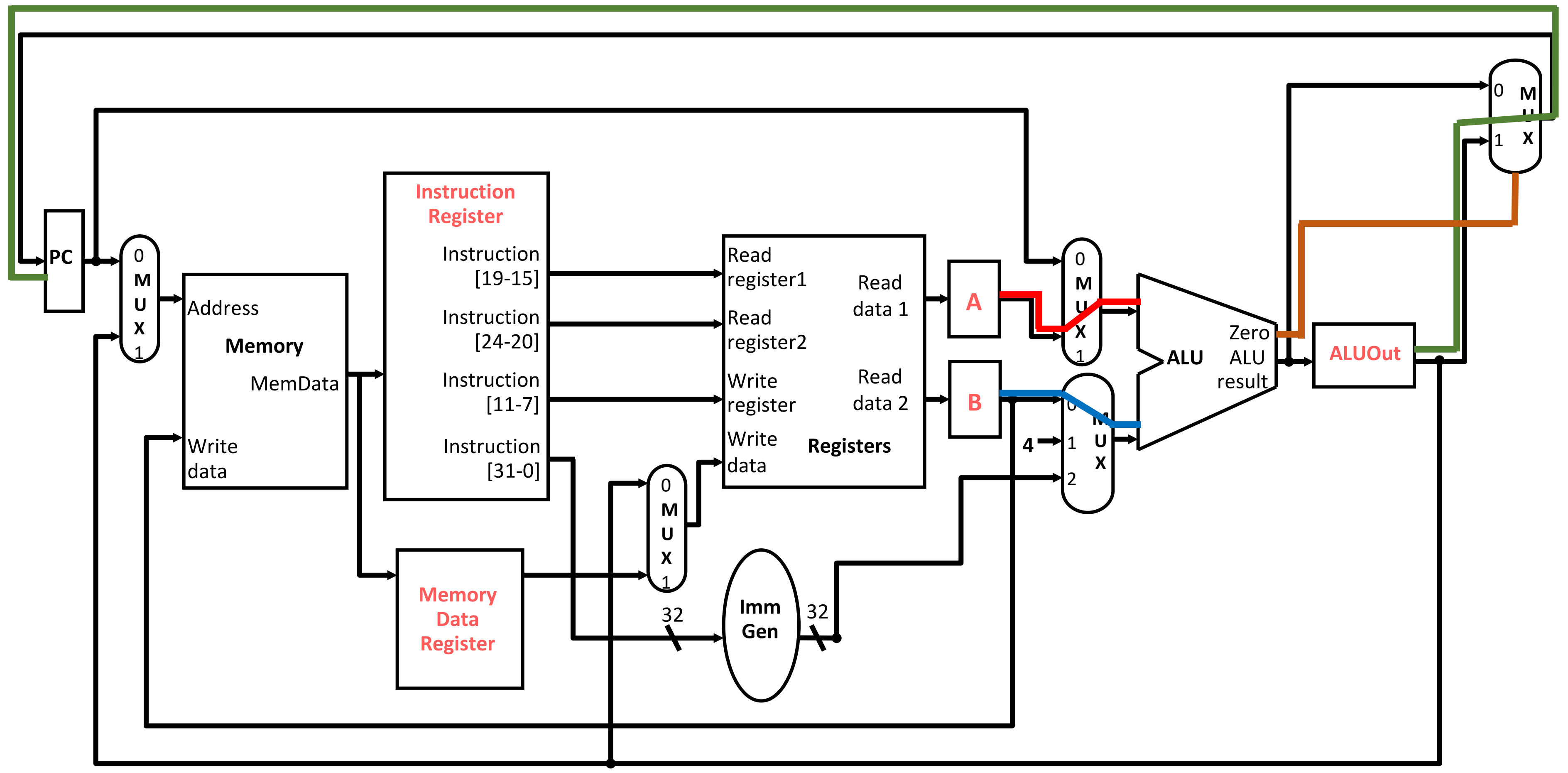
Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step3: Branch Completion Step (Cont'd)

- **RTL Description**

  ~~IR = Mem[PC]~~

  ~~PC = PC + 4~~



Step3: Memory Execution

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

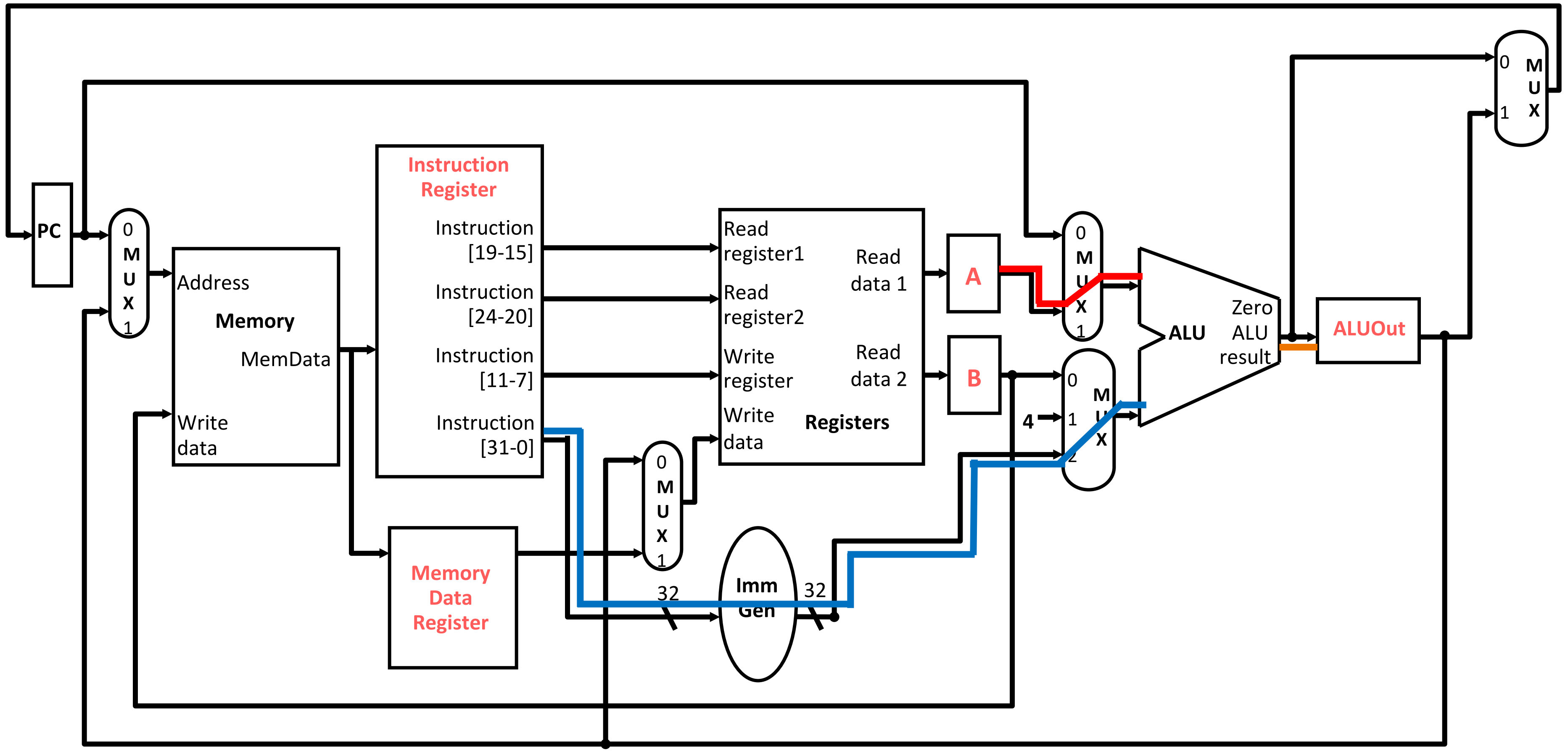
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step3: Memory Execution (Cont'd)

- **RTL Description**
  ALUOut = A + sign-extend (imm12) 메모리 내 주소값



Step4: Load Memory Access Step

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

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

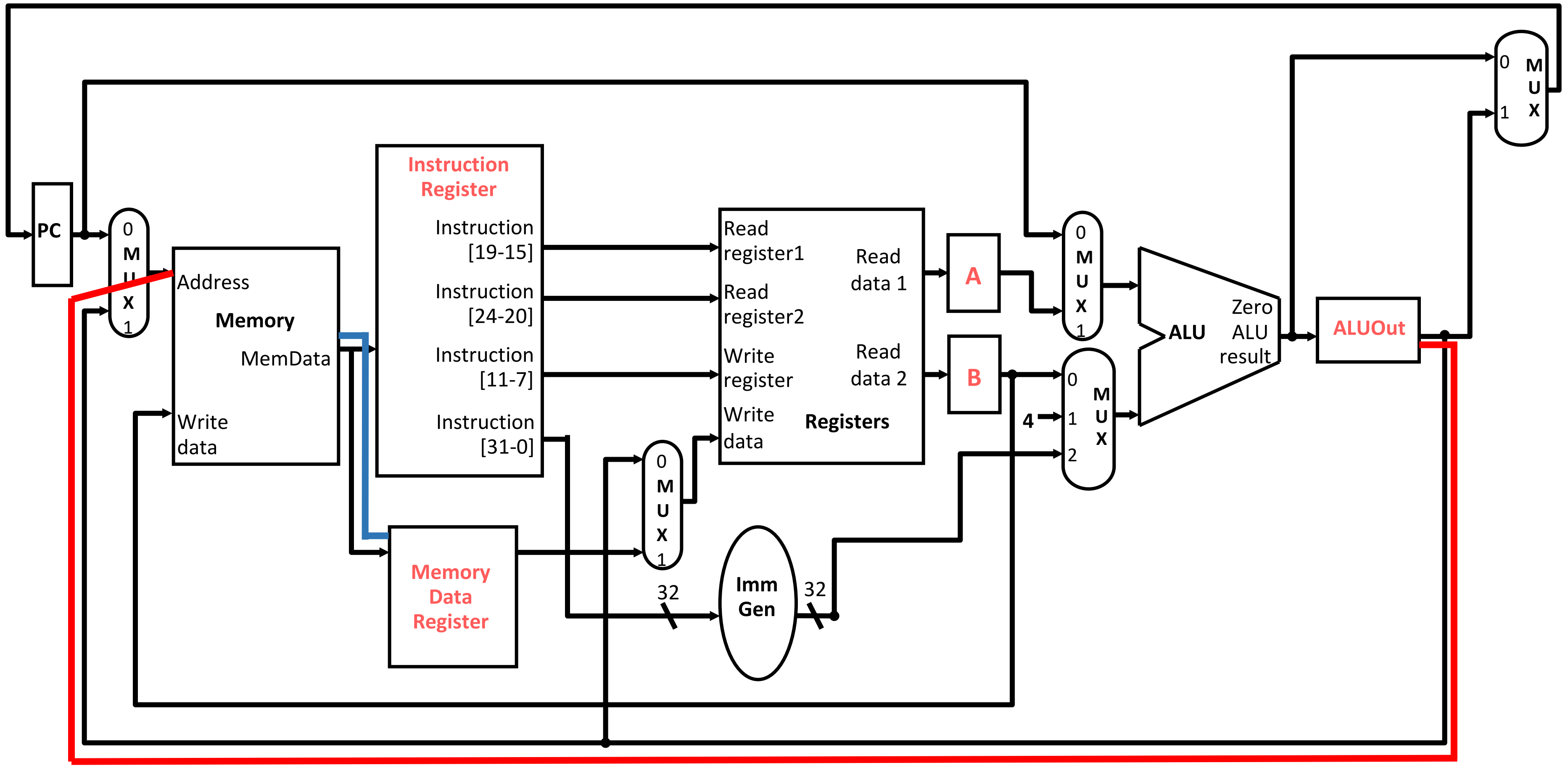
Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step4: Load Memory Access Step (Cont'd)

- **RTL Description**

  MDR = Memory[ALUOut]



Step5: Load Completion Step

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

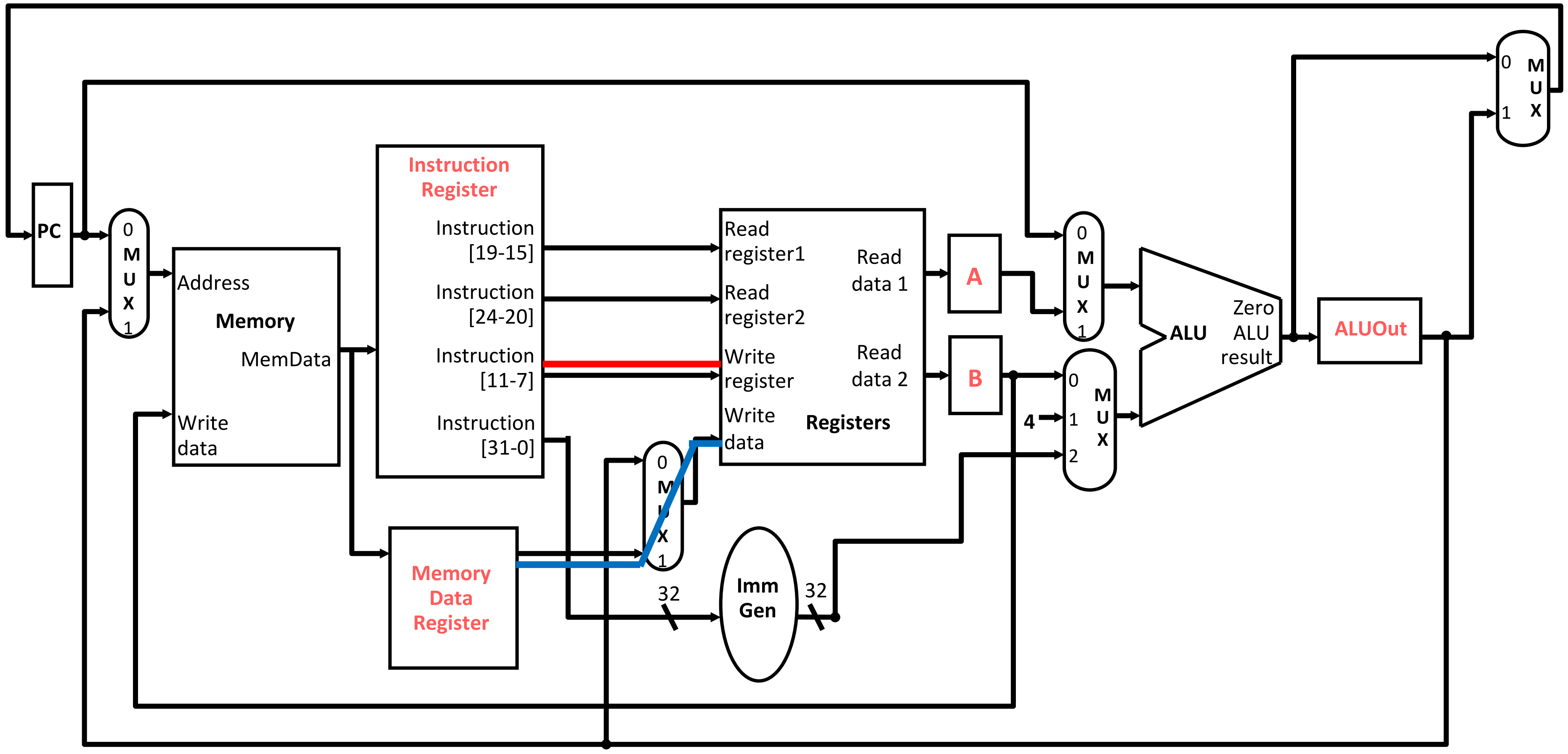
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step5: Load Completion Step (Cont'd)

- **RTL Description**
  Reg[IR[11-7]] = ALUOut



Step4: Store Memory Access Step

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

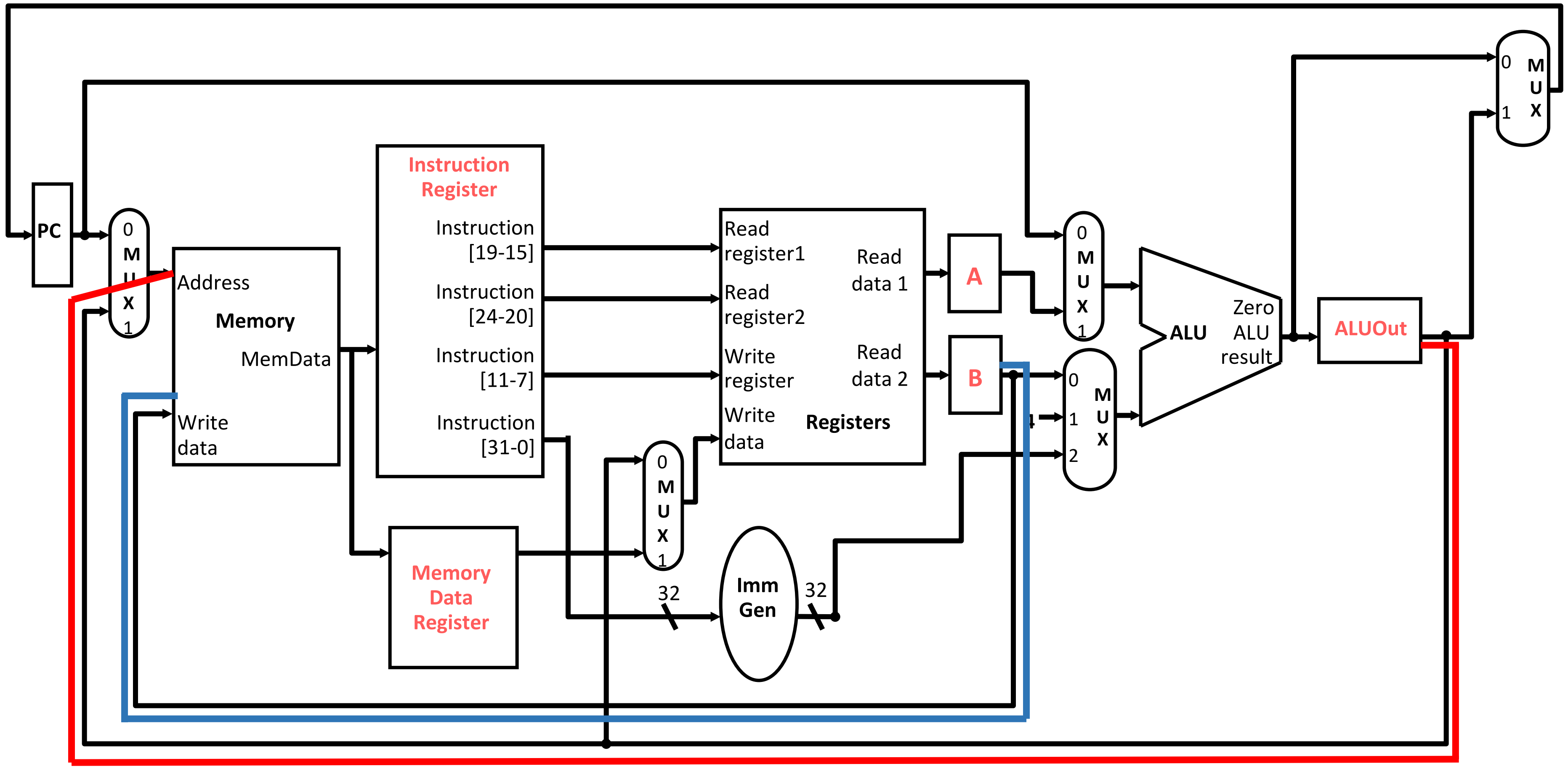
Instruction Fetch	IR = Memory[PC]
	PC = PC + 4		

Instruction Decode/
Register Fetch
	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

Step4: Store Memory Access Step (Cont'd)

- **RTL Description**
  Memory[ALUOut] = B



Summary of Multicycle Steps

Step Name	Action for R-type Instructions	Action for Memory Instructions	Action for Branches

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

Instruction Decode/ Register Fetch	A = Reg[IR[19-15]]
	B = Reg[IR[24-20]]
	ALUOut = PC + Branch Label (i.e., sign-extend (imm12 << 1))		

Execution, Address Computation, Branch Completion	ALUOut = A op B	ALUOut = A + sign-extend(imm12)	If (A==B) then PC = ALUOut
Memory Access, R-Type Completion	Reg[IR[11-7]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B	
Load Completion		Load: Reg[IR[11-7]] = MDR	

- **Load is the most complicate instruction**

Control Signals

- **Control signals no longer determined solely from decoded instruction**
  - Finite state machine used for sequencing

CPI of the Multicycle Implementation

- **Number of clock cycles**
  - Loads: 5
  - Stores: 4
  - R-format instructions: 4
  - Branches (& Jumps): 3
- **Instruction mix**
  - 22% Loads, 11% Stores, 49% R-format instructions, 16% Branches, and 2% Jumps
- **CPI = 0.22 x 5 + 0.11 x 4 + 0.49 x 4 + 0.16 x 3 + 0.02 x 3 = 4.04**
  - CPI of multicycle would be higher than that of single cycle
  - But, clock cycle time of multicycle is much shorter that that of single cycle

Pros and Cons of Multicycle Design

- **Advantages**
  - Shorter cycle time
  - Simple instructions executed in short period of time
    - Variable cycles per instruction no longer restricts to worst case
  - Functional units can be used more than once/instruction
    - Less hardware required to implement processor

- **Disadvantages**
  - Requires additional registers to store between stages
  - More timing paths to design, analyze, and tune

Summary

- **Processor design requires refinement of datapath and control**

- **Disadvantages of the single cycle implementation**
  - Long cycle time, too long for all instructions except for the slowest
  - Inefficient hardware utilization with unnecessarily duplicated resources

- **Multicycle Implementation**
  - Partition execution into small steps of comparable duration