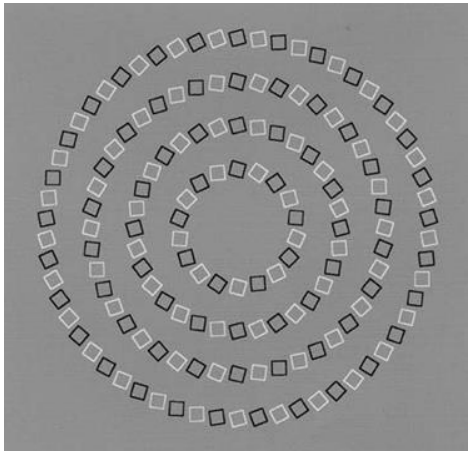
Computer Organization

Lecture 13 - Processor Implementation: Overview, Single-Cycle Design



Reading 4.1-4.4, D.1, D.2

Homework: Uclass

Roadmap for the Term: Major Topics

- Computer Systems Overview
- Technology Trends
- Performance
- Instruction Sets (and Software)
- Logic and Arithmetic
- Processor Implementation
- Memory Systems
- Input/Output

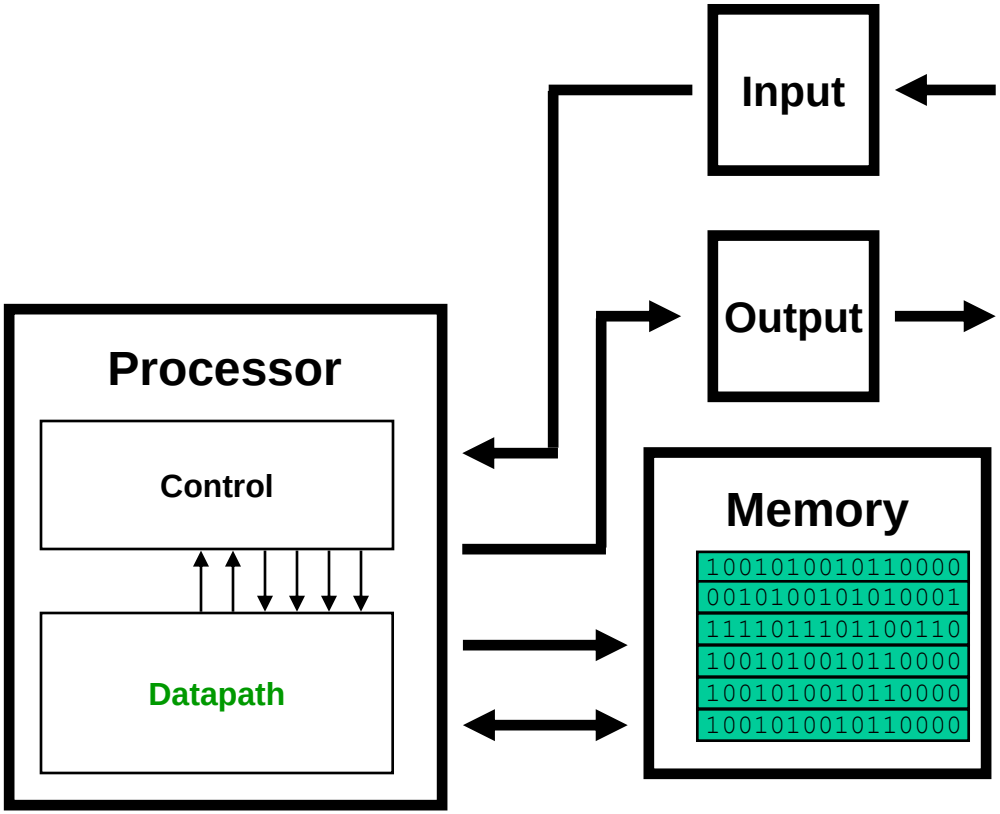
Outline - Processor Implementation

- **Overview**
  - Review of Processor Operation
  - Steps in Processor Design
  - Implementation Styles
  - The “MIPS Lite” Instruction Subset
- **Single-Cycle Implementation**
- **Pipelined Implementation**

- **본 주제의 핵심**
  - 프로세서를 설계하는 방법 혹은 과정에 대한 이해
  - ISA를 기반으로 설계과정 전개

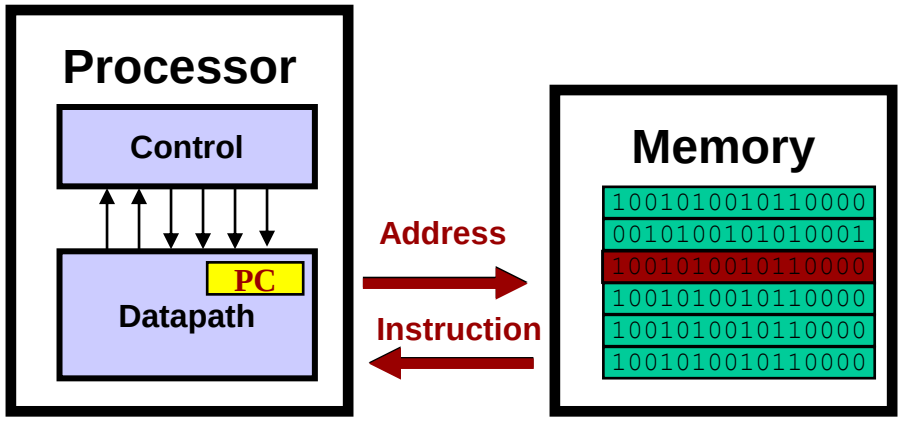
Review: The “Five Classic Components”

- Processor
  - Datapath
  - Control
- Memory
- Input
- Output



Review: Processor Operation

- Executing Programs - the “fetch/execute” cycle
  - Processor **fetches** instruction from memory
  - Processor **executes** “machine language” instruction
    - Perform calculation
    - Read/write data
  - Repeat with “next” instruction



Datapath: 명령어가 MP를 구성하는 여러 가지 H/W 요소 (Registers, ALU, Memory 등)들을 통과하면서 결과를 만들어 내기 때문에 붙여진 이름

Processor Design Goals

- Design hardware that:
  - Fetches instructions from memory
  - Executes instructions as specified by ISA
- Design considerations
  - Cost
  - Speed
  - Power

Steps in Processor Design

1. **(a) Analyze instruction set; (b) get datapath requirements**
2. **Select datapath components and establish clocking methodology**
3. **Assemble datapath that meets requirements**
4. **Determine control signal values for each instruction**
5. **Assemble control logic to generate control signals**

Processor Implementation Styles

- Single Cycle
  - Perform each instruction in 1 clock cycle
  - Disadvantage: only as fast as “slowest” instruction
- Multi-Cycle*
  - Break fetch/execute cycle into multiple steps
  - Perform 1 step in each clock cycle
- Pipelined

*once popular approach - no longer covered

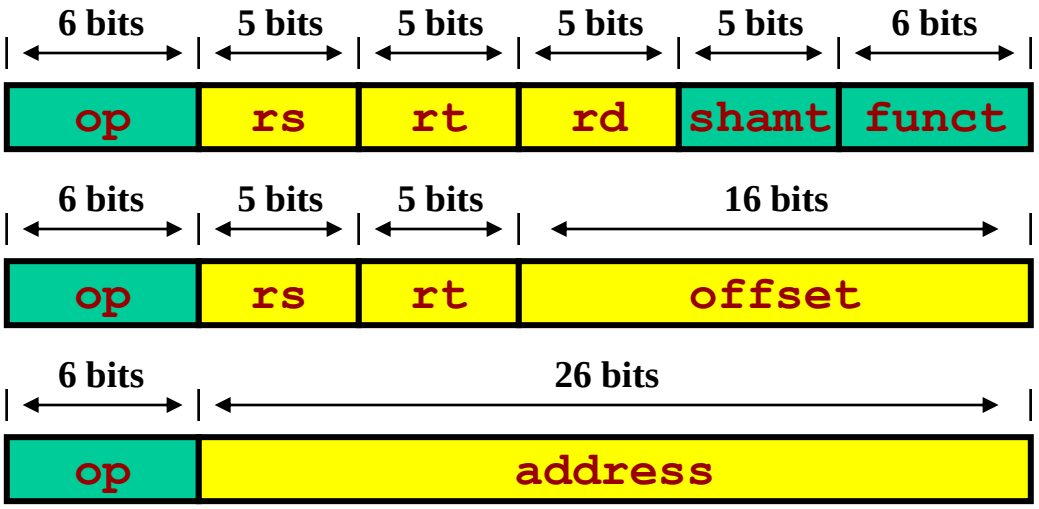
  - Execute each instruction in multiple steps
    - 4개의 step이 있으면 4개의 명령어가 MP에서 실행하는 형태
  - Perform 1 step / instruction in each clock cycle
  - Process multiple instructions in parallel - “assembly line”

“MIPS Lite” - A Pedagogical Example

- Use a MIPS to illustrate processor design
- Limit initial design to a subset of instructions:
  - Memory access: `lw`, `sw`
  - Arithmetic/Logical: `add`, `sub`, `and`, `or`, `slt`
  - Branch/Jump: `beq`, `j`

    => A machine for 9 instructions
- Add instructions as we go along (e.g., `addi`)

Review - MIPS Instruction Formats



R-Format
- Arithmetic/Logical

I-Format
- Constant operation
- Branch
- Memory access

J-Format
- Unconditional branch

- **Field definitions:**
  - **op: instruction opcode**
  - **rs, rt, rd: source (2) and destination (1) register numbers**
  - **shamt: shift amount**
  - **funct: function code (works with opcode to specify op)**
  - **offset/immediate: address offset or immediate value**
  - **address: target address for jumps**

1-(a). Analyze MIPS Instruction Subset

- **Arithmetic & Logical Instructions**

```
add $s0, $s1, $s2
sub $s0, $s1, $s2
and $s0, $s1, $s2
or $s0, $s1, $s2
```

- **Data Transfer Instructions**

```
lw $s1, offset($s0)
sw $s2, offset($s3)
```

- **Branch**

```
beq $s0, offset
j address
```

MIPS Instruction Execution

- General Procedure
  1. Fetch Instruction from memory
  2. Decode Instruction, read register values
  3. If necessary, perform an ALU operation
  4. If load or store, do memory access
  5. Write results back to register file and increment PC
- Register Transfers provide a concise description

Register Transfers for the MIPS Subset

- **Instruction Fetch**

  Instruction <- MEM[PC]

- **Instruction Execution**

Instr.	Register Transfers	
`add`	R[rd] <- R[rs] + R[rt];	PC <- PC + 4
`sub`	R[rd] <- R[rs] - R[rt];	PC <- PC + 4
`and`	R[rd] <- R[rs] & R[rt];	PC <- PC + 4
`or`	R[rd] <- R[rs] | R[rt];	PC <- PC + 4
`lw`	R[rt] <- MEM[R[rs] + s_extend(offset)];	PC<- PC + 4
`sw`	MEM[R[rs] + sign_extend(offset)] <- R[rt];	PC <- PC + 4
`beq`	if (R[rs] == R[rt]) then PC <- PC+4 + s_extend(offset<<2)	
	else PC <- PC + 4	
`j`	PC <- upper(PC)@(address << 2)	

Outline - Processor Implementation

- Overview
- Single-Cycle Implementation
  1. Analyze instruction set; get datapath requirements
  2. Select datapath components and establish clocking methodology
  3. Assemble datapath that meets requirements
  4. Determine control signal values for each instruction
  5. Assemble control logic to generate control signals
- Pipelined Implementation

1-(b). Instruction Set (datapath) Requirements

- Memory
  - Read Instructions
  - Read and Write Data
- Registers - 32
  - read (from `rs` field in instruction)
  - read (from `rt` field in instruction)
  - write (from `rd` or `rt` field in instruction)
- PC
- Sign Extender
- Add and Subtract (register values)
- Add 4 or extended immediate to PC

Review register transfers for details!

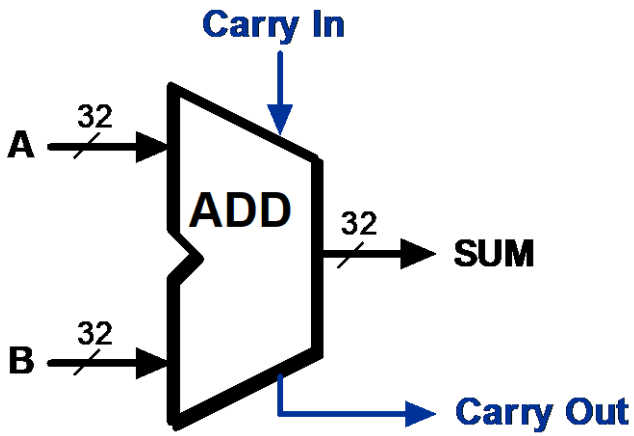
Outline - Processor Implementation

- Overview
- Single-Cycle Implementation
  1. (a) Analyze instruction set; (b) get datapath requirements
  2. (a) Select datapath components and (b) establish clocking methodology
  3. Assemble datapath that meets requirements
  4. Determine control signal values for each instruction
  5. Assemble control logic to generate control signals
- Pipelined Implementation

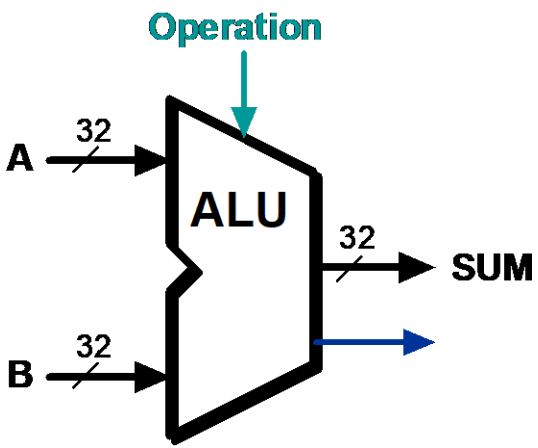
2. (a) Choose Datapath Components

- **Combinational Components**
  - **Adder**
  - **ALU**
  - **Multiplexer**
  - **Sign Extender**
- **Storage Components**
  - **Registers**
  - **Register File**
  - **Memory**

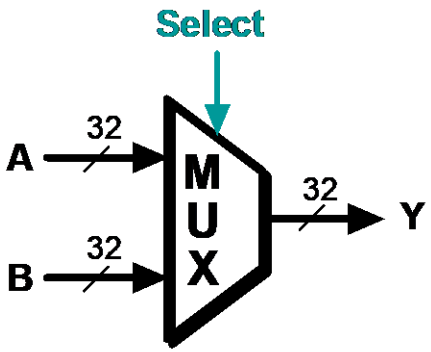
Datapath Combinational Components



Adder



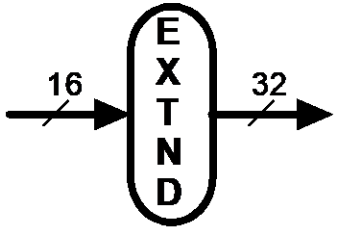
ALU



Multiplexer

NOTES:

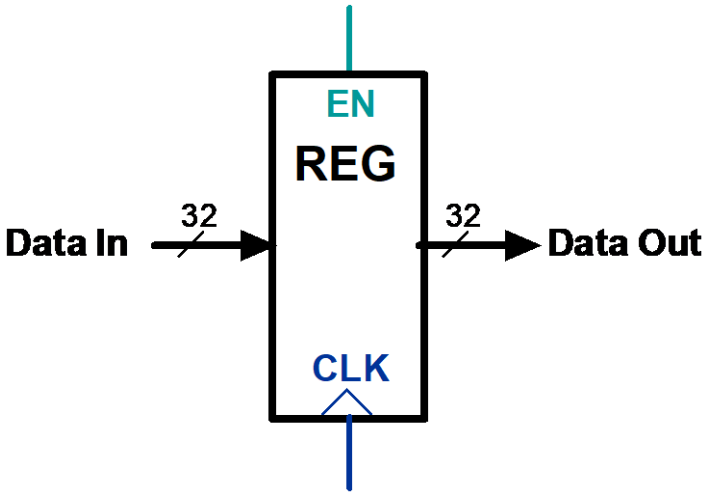
- Blue-green inputs are control lines
- Blue lines often hidden to suppress detail



Sign Extender

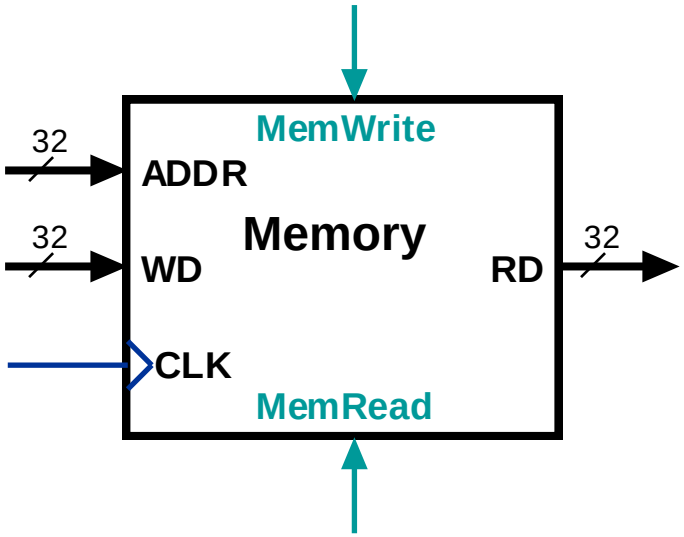
Datapath Storage - Registers

- Registers store multiple bit values
- New value loaded on clock edge when EN asserted



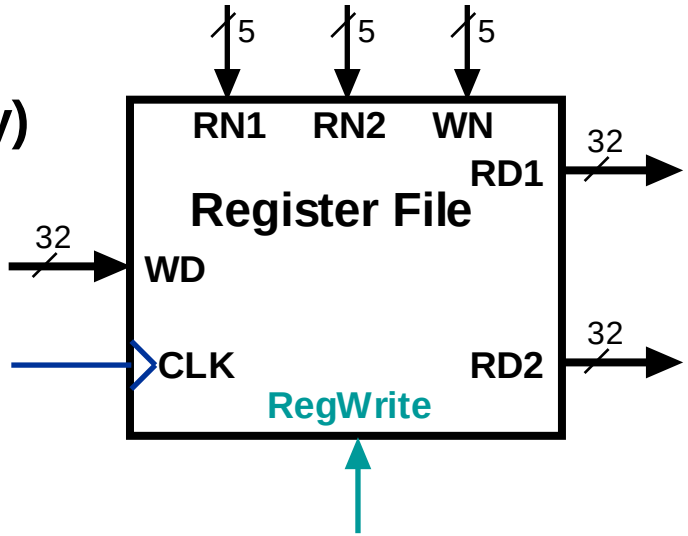
Datapath Storage: Idealized Memory

- Data Read
  - Place Address on ADDR
  - Assert MemRead
  - Data Available on RD after memory “access time”
- Data Write
  - Place address on ADDR
  - Place data input on WD
  - Assert MemWrite
  - Data written on clock edge



Datapath Storage: Register File

- Register File - 32 registers (including `$zero`)
- Two data outputs RD1, RD2
  - Assert register number RN1/RN2
  - Read output RD1/RD2 after "access time" (propagation delay)
- One data input WD
  - Assert register number WN
  - Assert value on WD
  - Assert RegWrite
  - Value loaded on clock edge
- Implemented as a small multiport memory

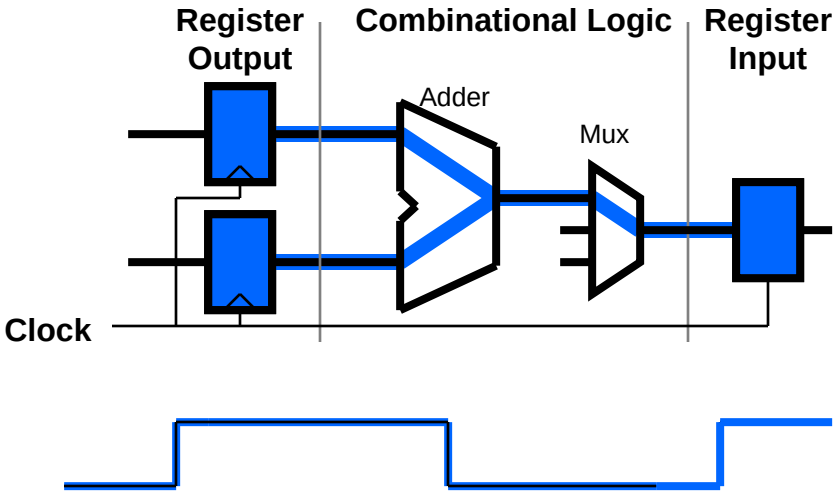


2. (b) Choose Clocking Methodology

- Clocking methodology defines
  - When signals can be read from storage elements
  - When signals can be written to storage elements
- Typical clocking methodologies
  - Single-Phase Edge Triggered
  - Single-Phase Level Triggered
  - Multiple-Phase Level Triggered
- Authors' choice: Single-Phase Edge Triggered
  - All registers updated on one edge of clock cycle
  - Simplest to work with

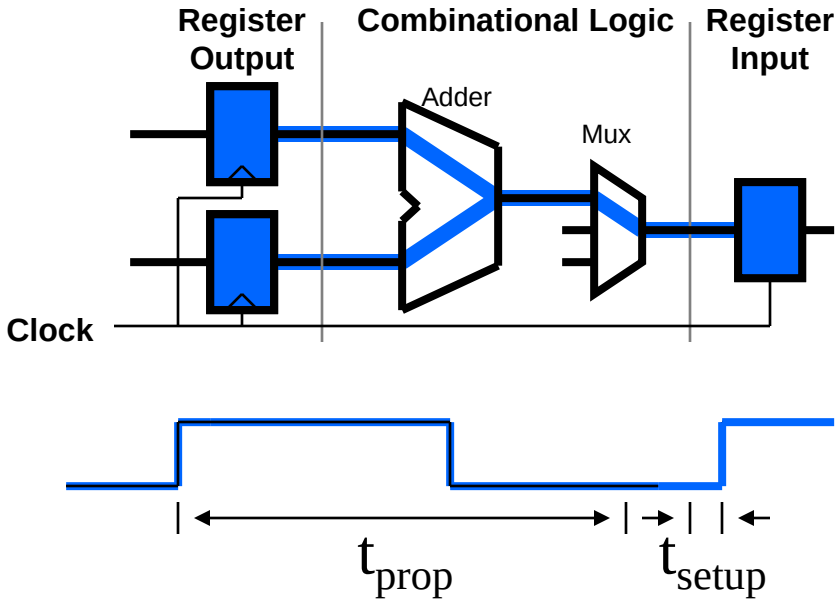
Review: Edge-Triggered Clocking

- Controls sequential circuit operation
  - Register outputs change after first clock edge
  - Combinational logic determines “next state”
  - Storage elements store new state on next clock edge



Review: Edge-Triggered Clocking

- **Propagation delay - t_{prop}**
  - Logic (including register outputs)
  - Interconnect
- **Register setup time - t_{setup}**



$$t_{clock} > t_{prop} + t_{setup}$$

$$t_{clock} = t_{prop} + t_{setup} + t_{slack}$$

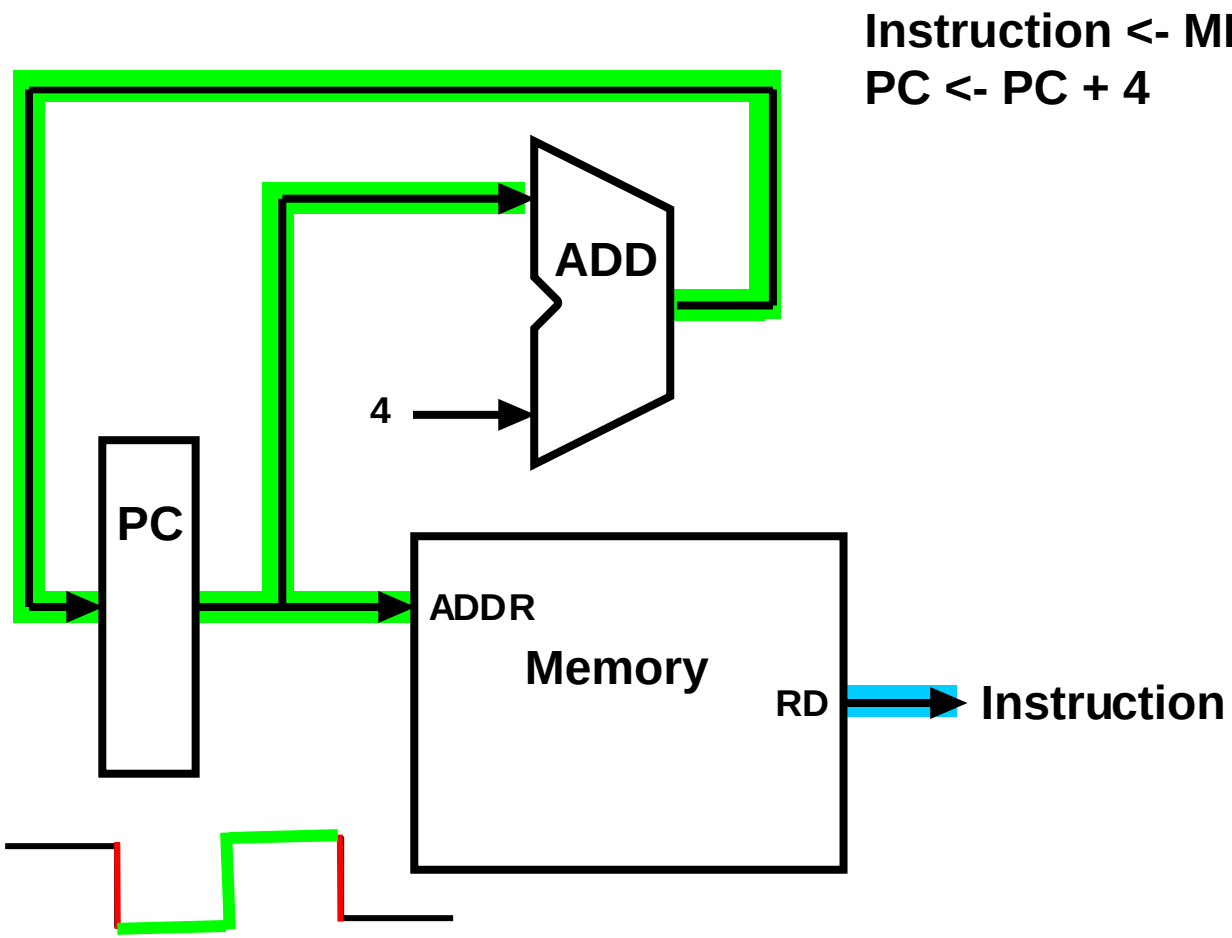
Outline - Processor Implementation

- Overview
- Single-Cycle Implementation
  1. Analyze instruction set; get datapath requirements
  2. Select datapath components and establish clocking methodology
  3. Assemble datapath that meets requirements
  4. Determine control signal values for each instruction
  5. Assemble control logic to generate control signals
- Pipelined Implementation

3. Assemble Datapath

- **Tasks that processor must implement**
  1. **Fetch Instruction from memory**
  2. **Decode Instruction, read register values**
  3. **If necessary, perform an ALU operation**
  4. **If memory address, perform load/store**
  5. **Write results back to register file and increment PC**
- **How can we do this with the datapath hardware?**

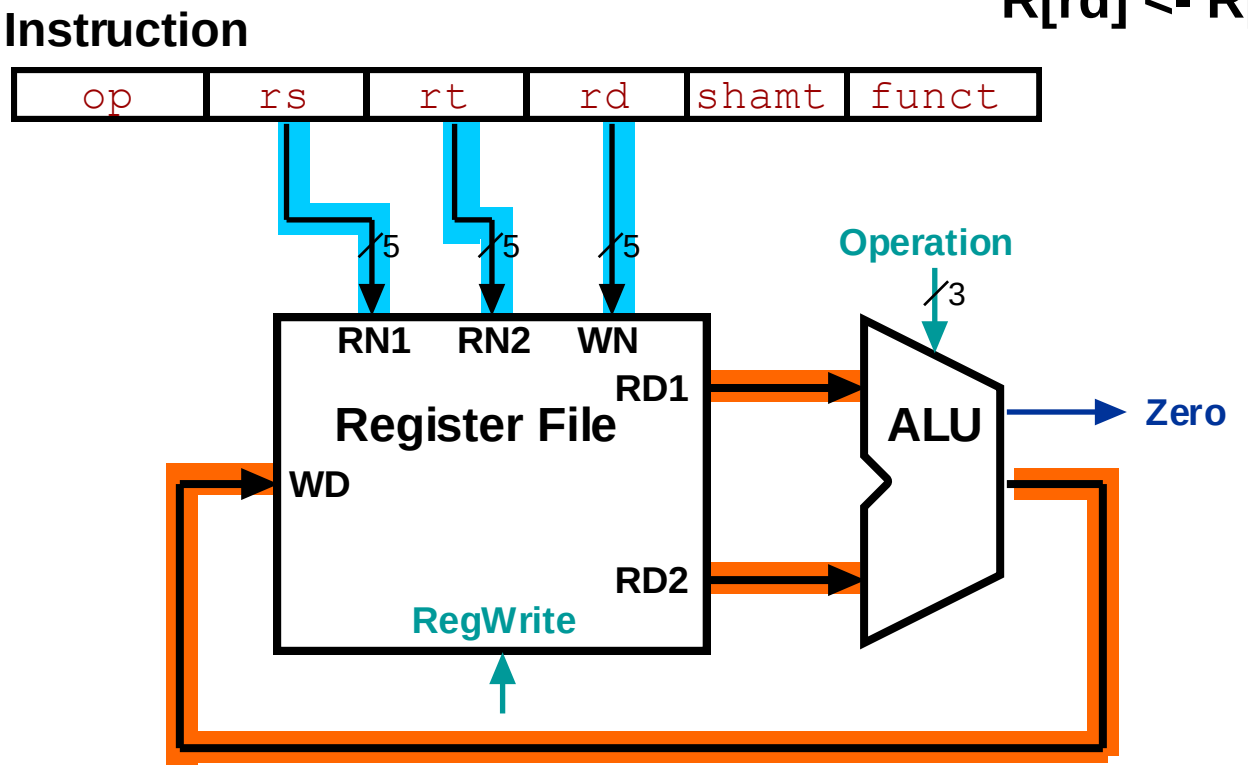
Datapath for Instruction Fetch



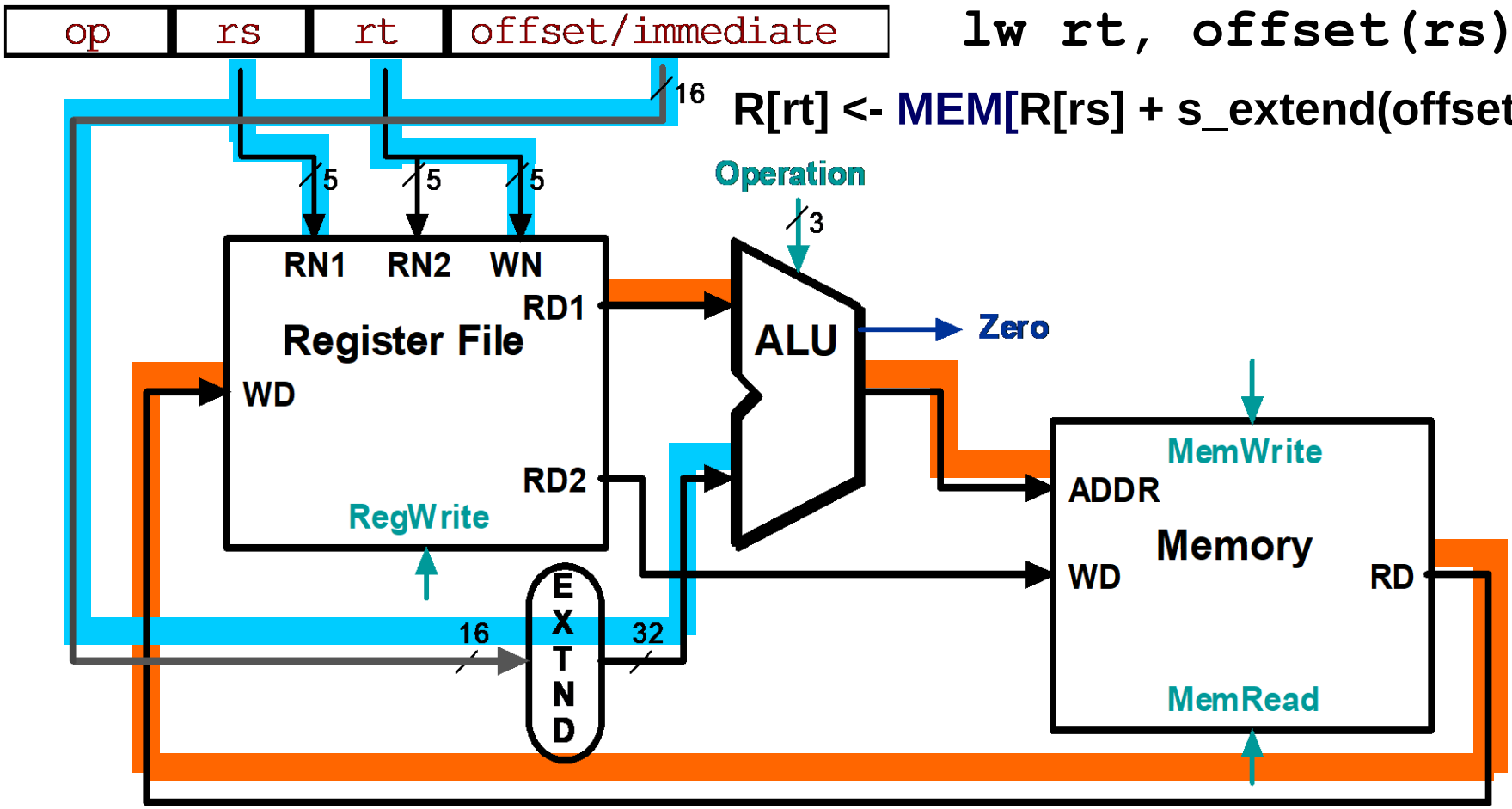
Datapath for R-Type Instructions

```
add rd, rs, rt
```

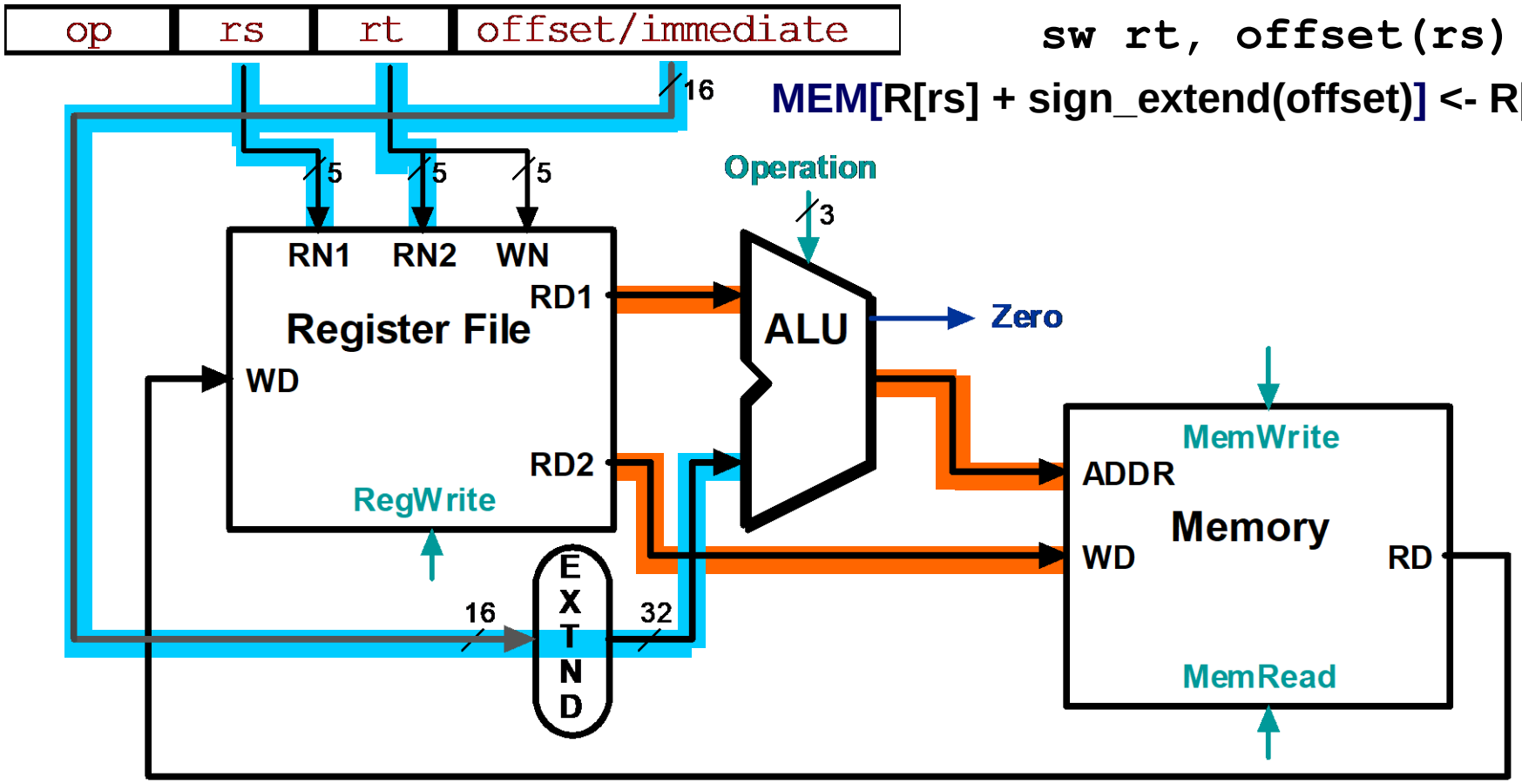
R[rd] <- R[rs] + R[rt];



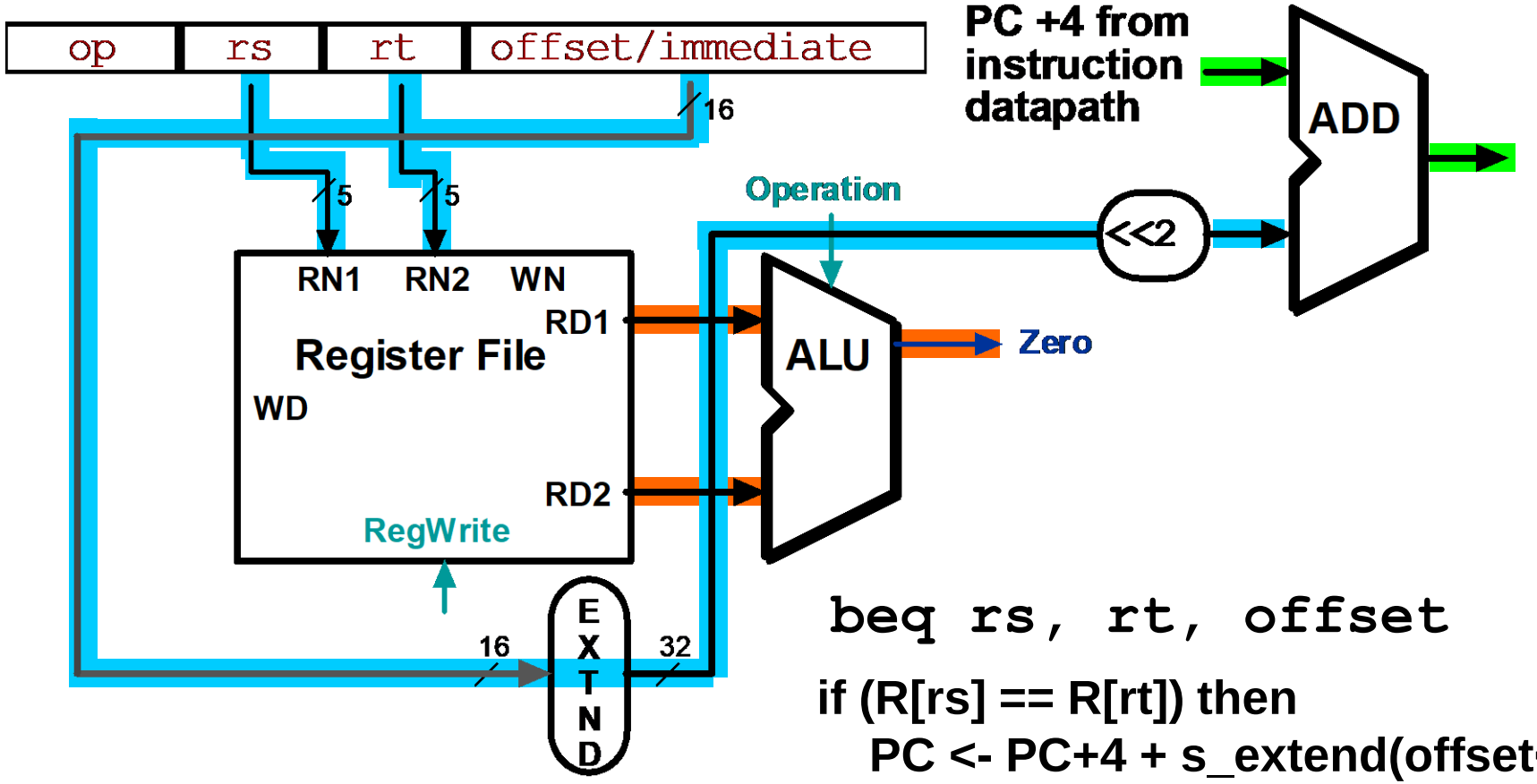
Datapath for Load/Store Instructions



Datapath for Load/Store Instructions



Datapath for Branch Instructions

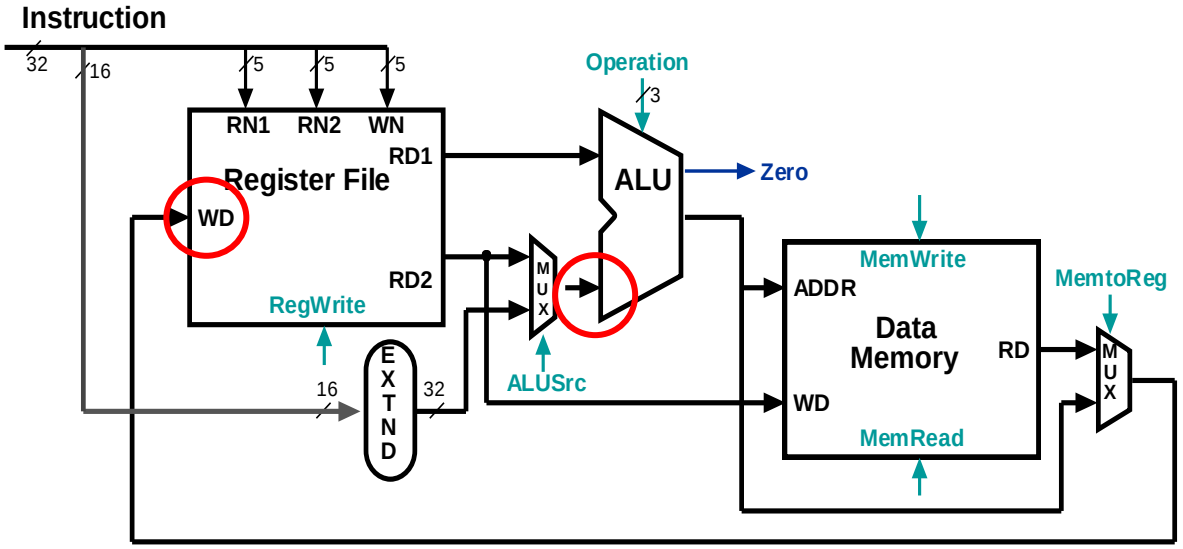


Putting it all together…

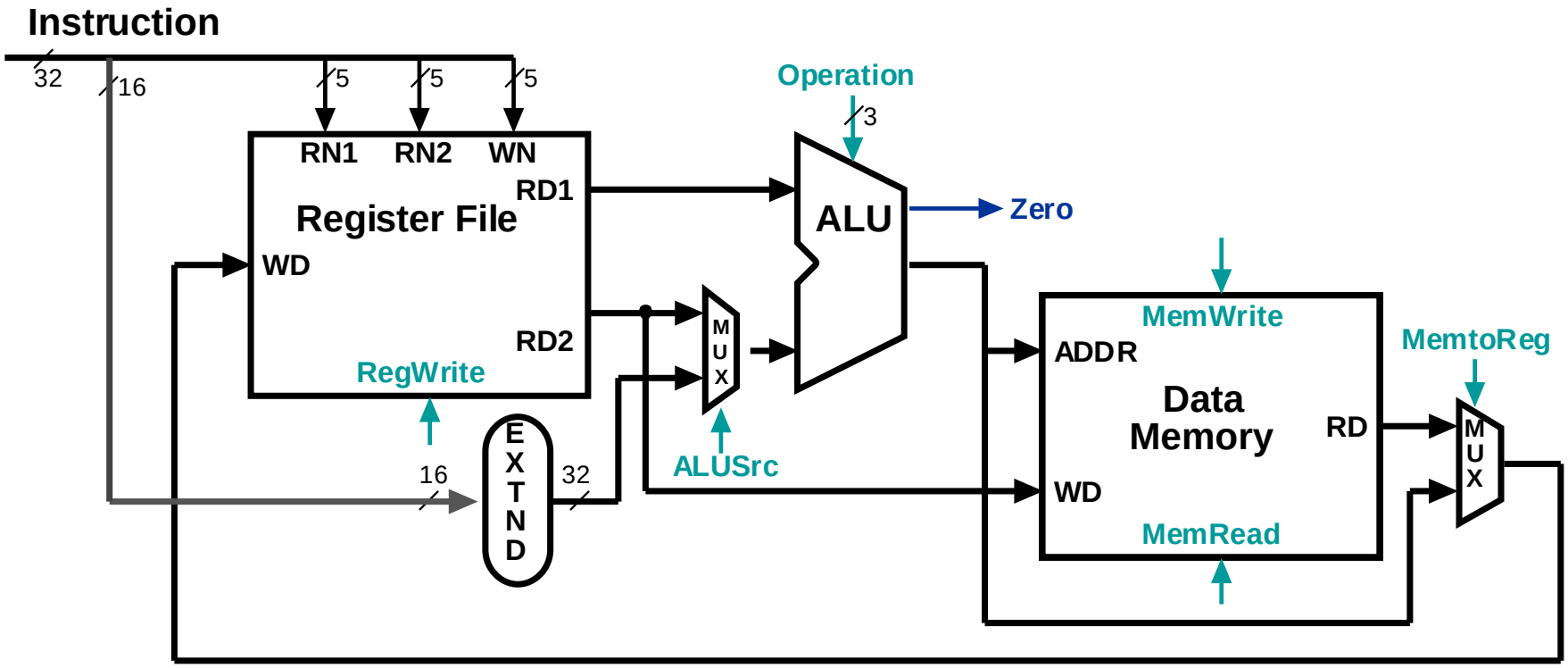
- Goal: merge datapaths for each function
  - Instruction Fetch
  - R-Type Instructions
  - Load/Store Instructions
  - Branch instructions
- Add multiplexers to steer data as needed

Example: combine R-Type and Load/Store Datapaths

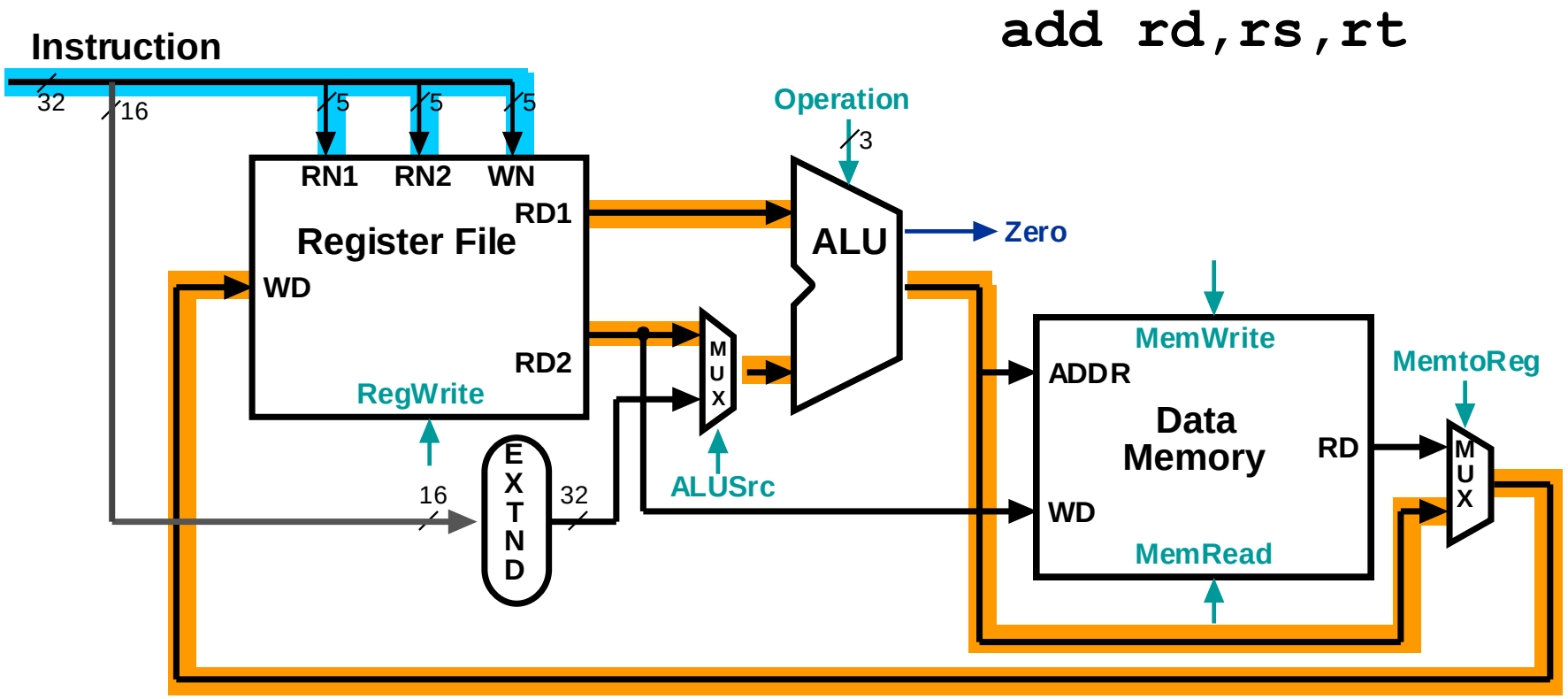
- Select an ALU input from either
  - Register File output RD2 (for R-Type)
  - Sign-extender output (for LW/SW)
- Select Register File input WD from either
  - ALU output (for R-Type)
  - Memory output RD (for LW)



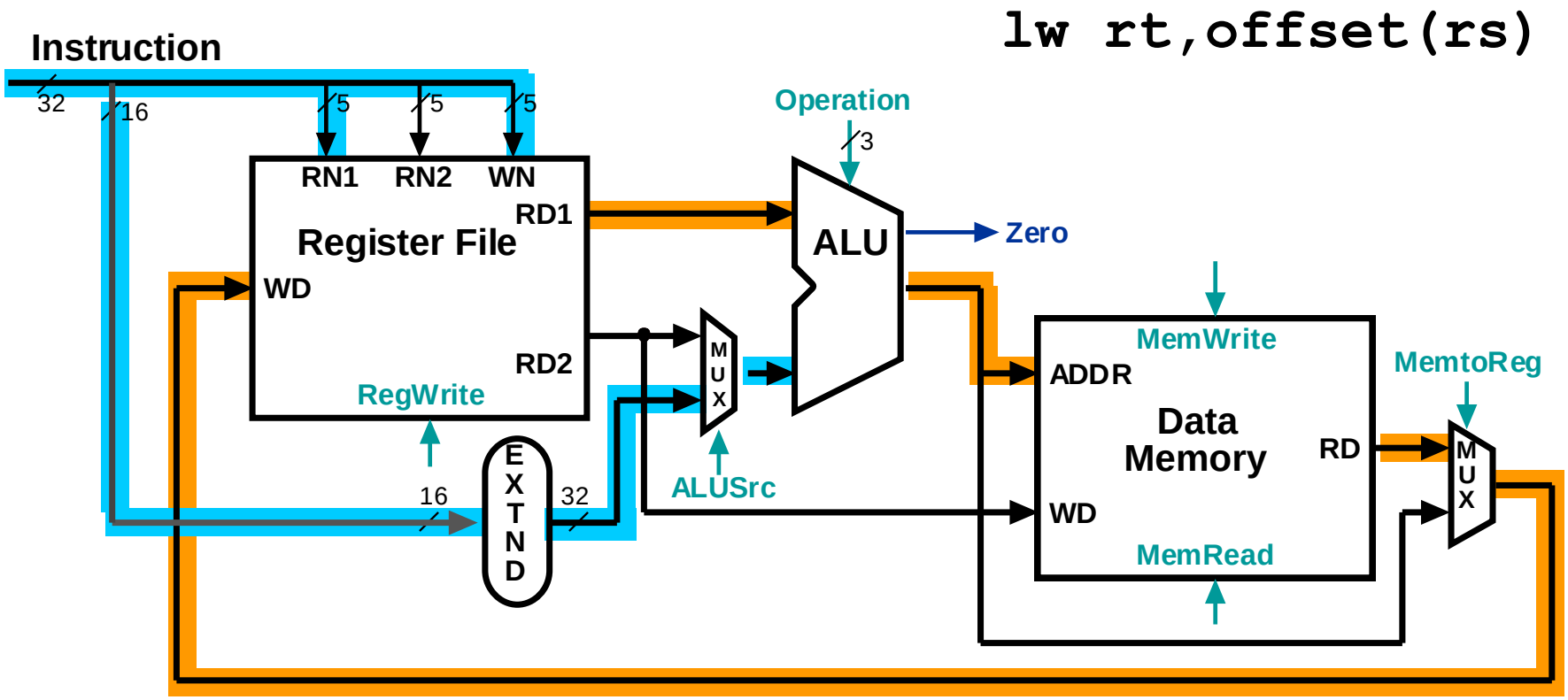
Combined Datapath: R-Type and Load/Store Instructions



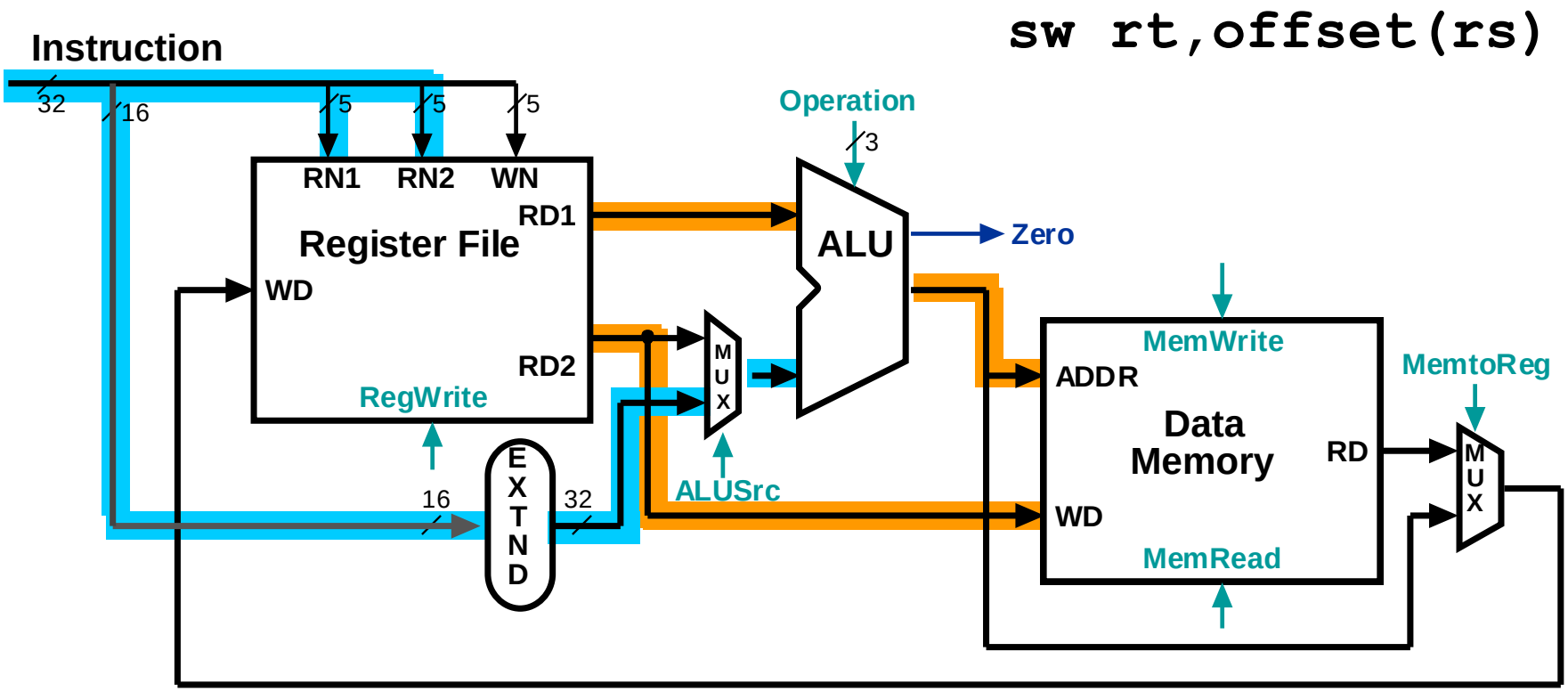
Combined Datapath: Executing an R-Type Instruction



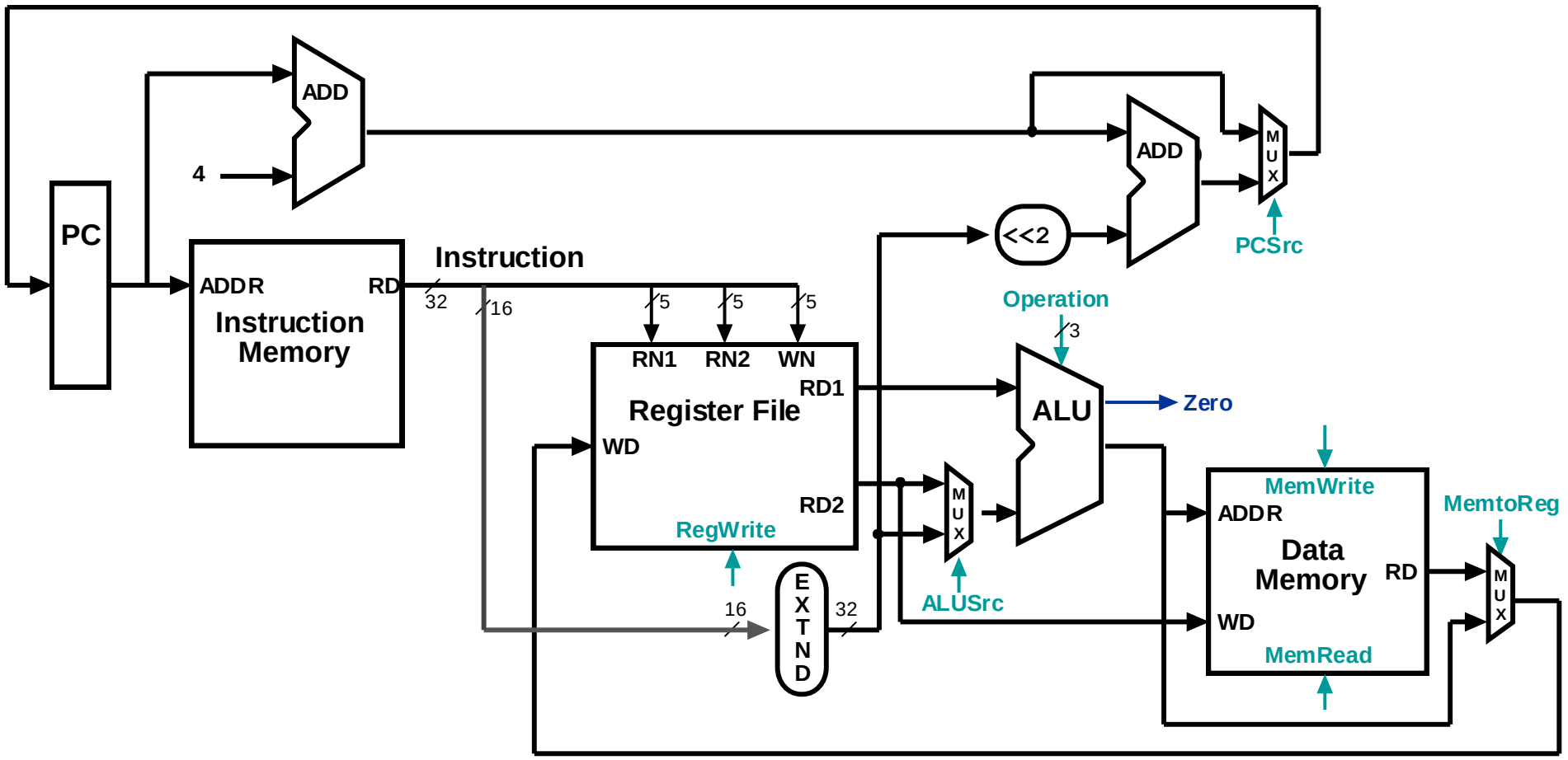
Combined Datapath: Executing a load instruction



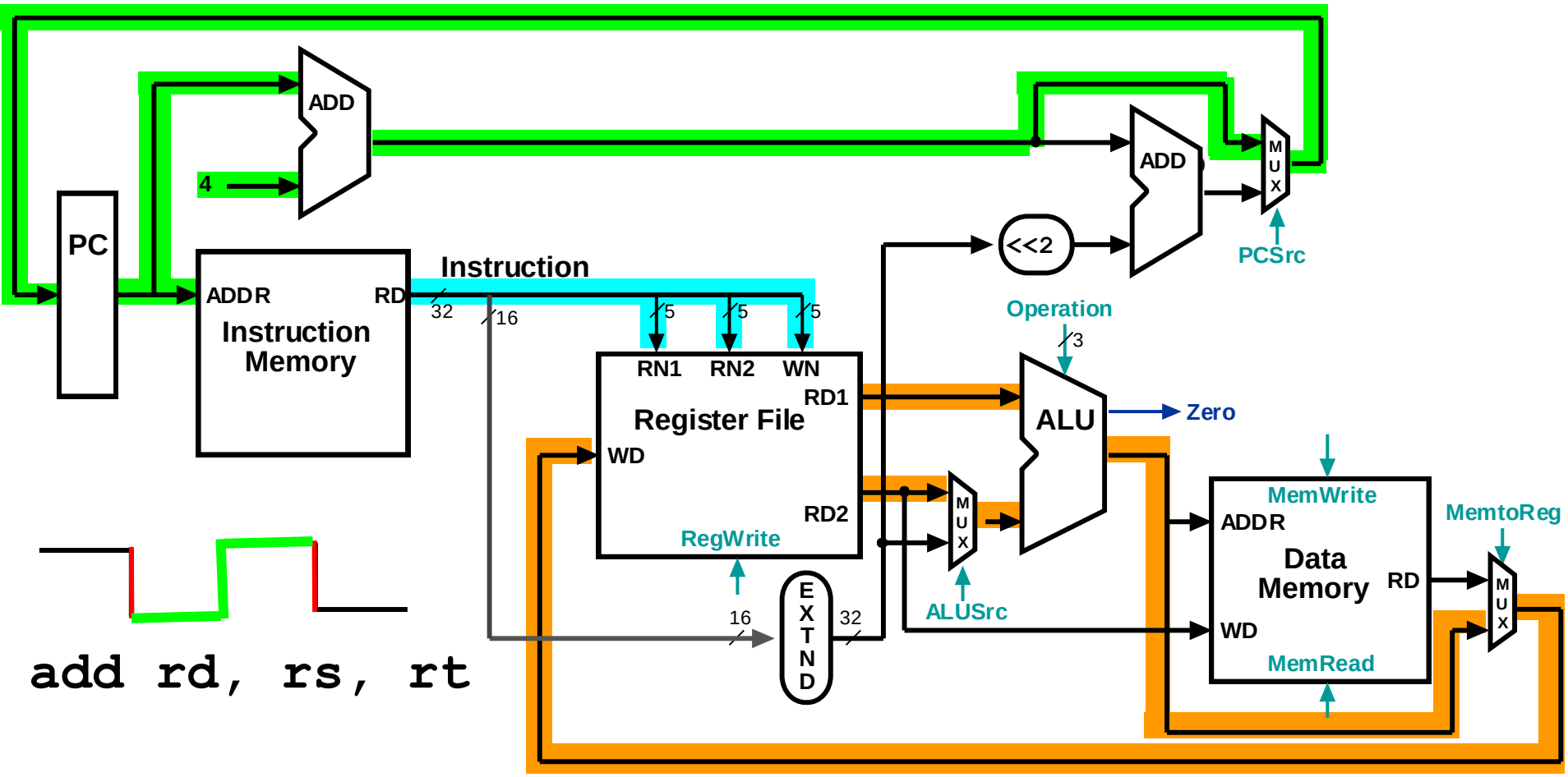
Combined Datapath: Executing a store instruction



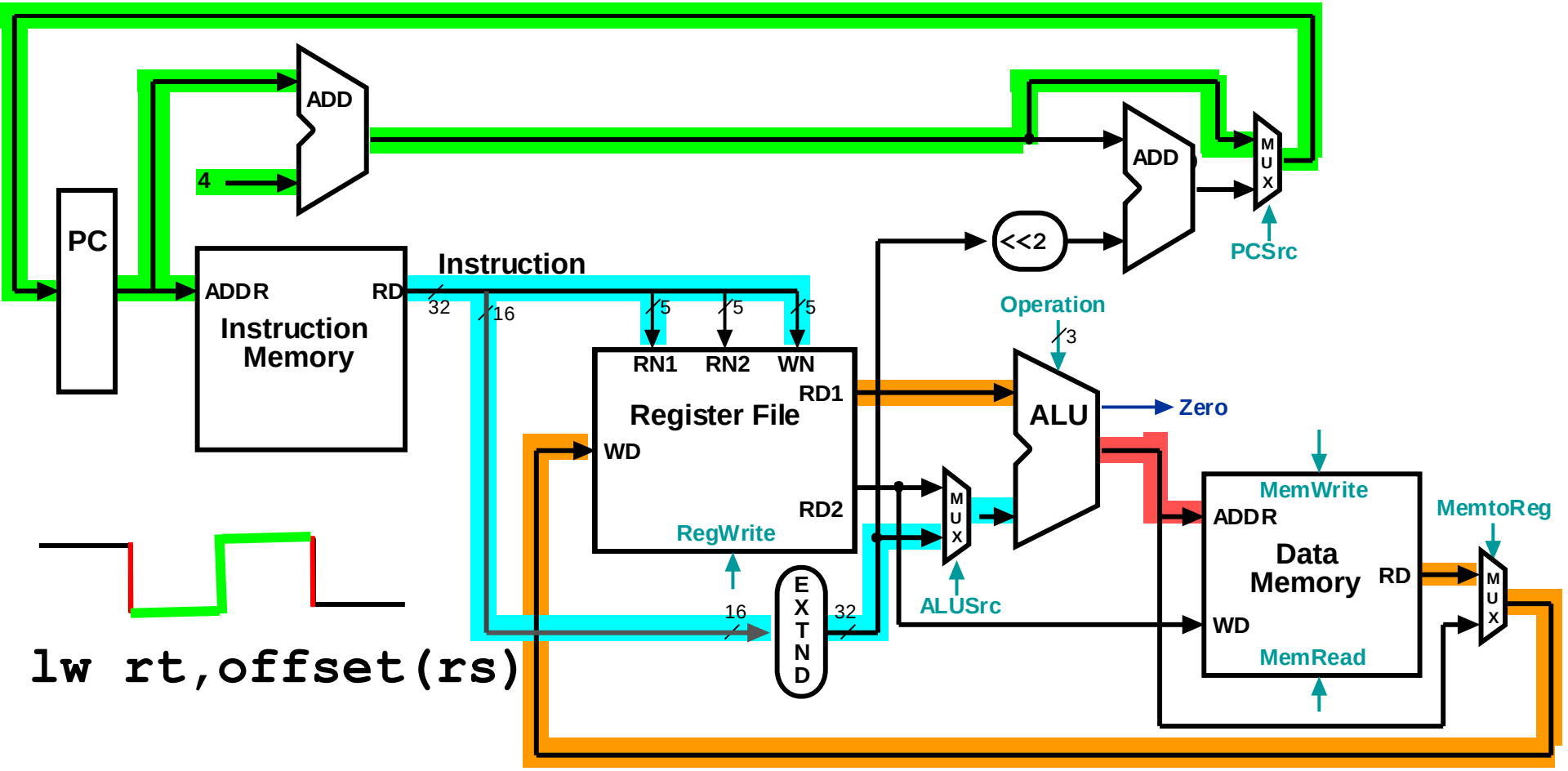
Complete Single-Cycle Datapath



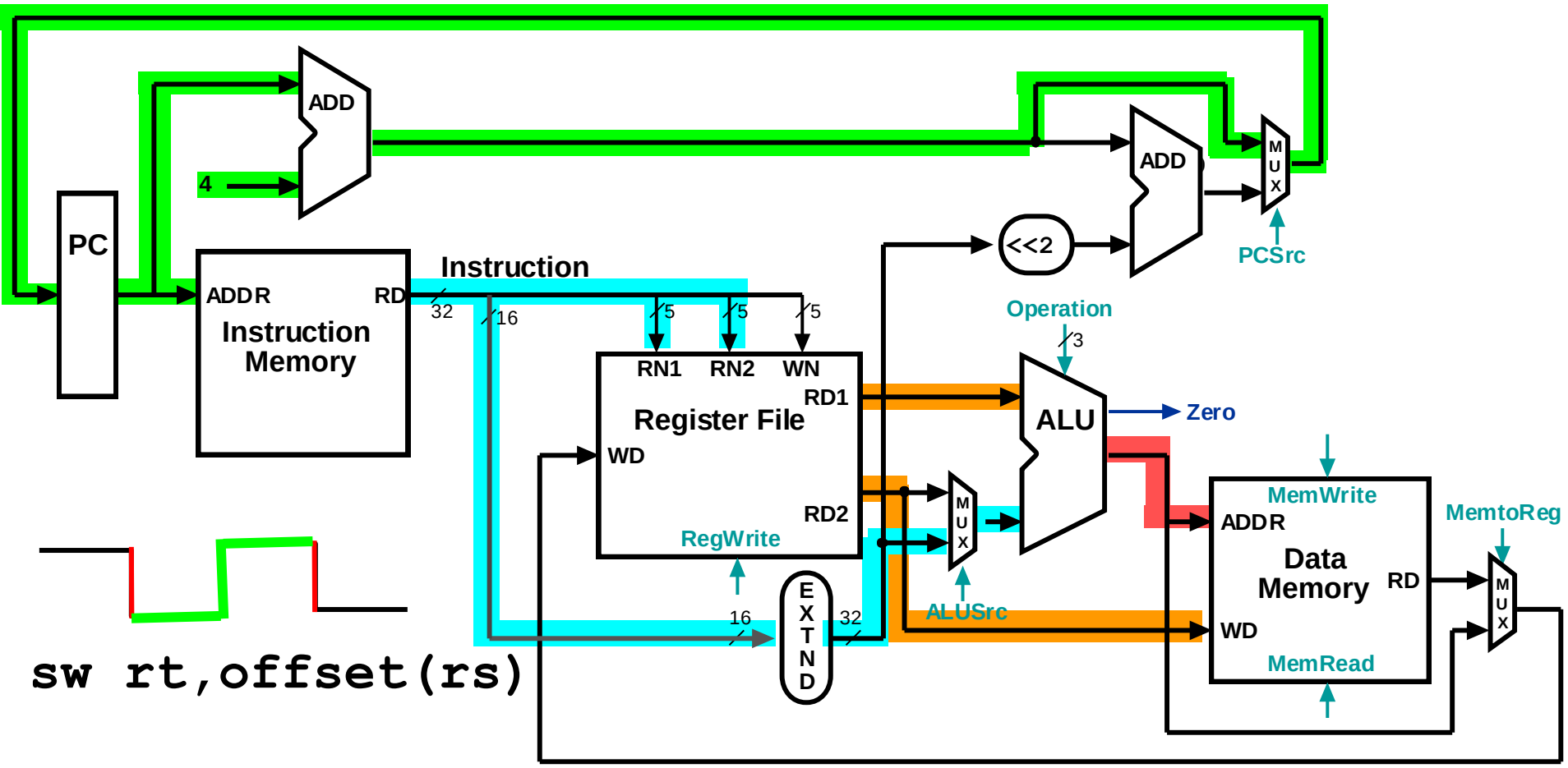
Complete Datapath Executing add



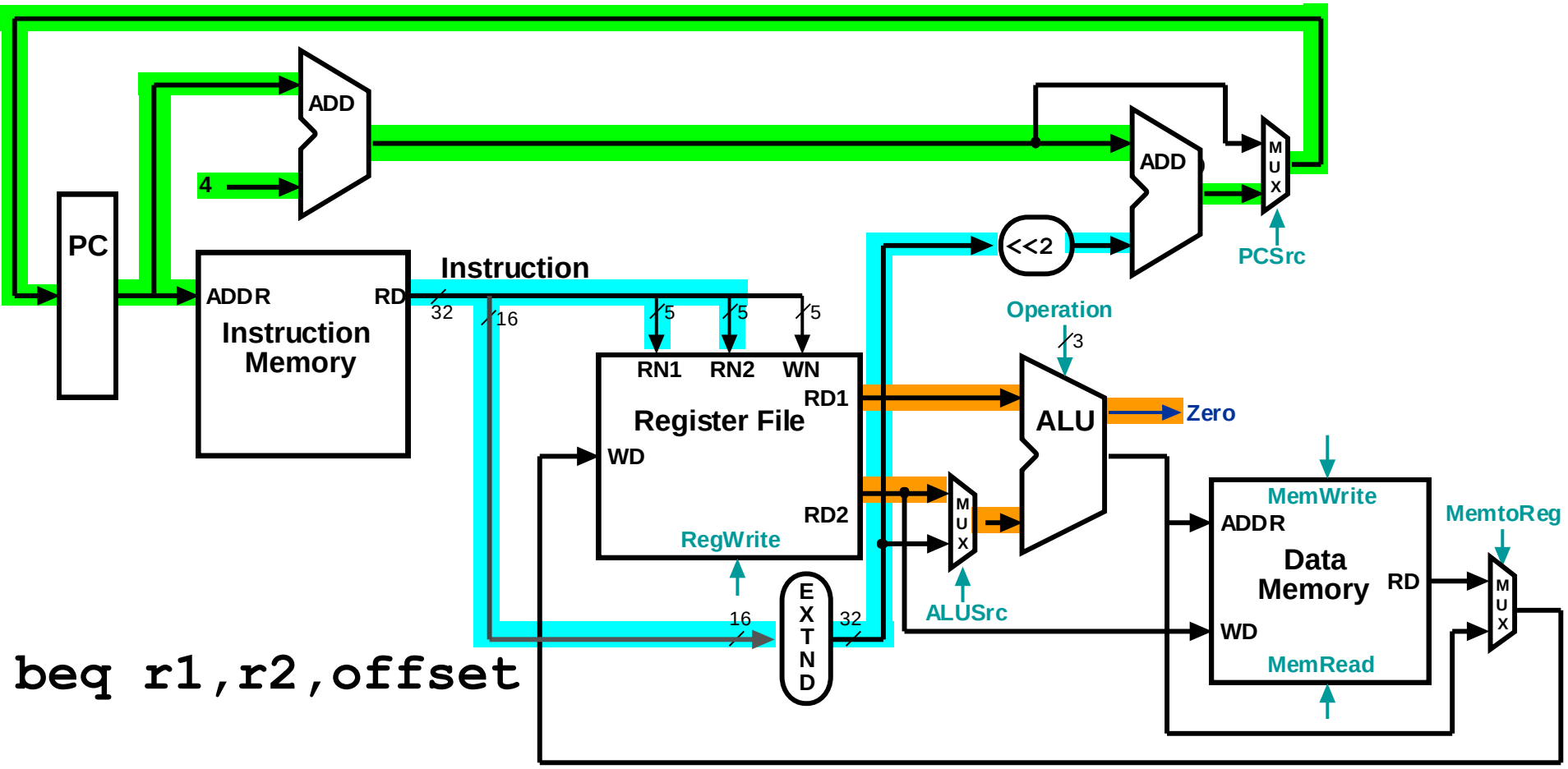
Complete Datapath Executing load



Complete Datapath Executing store

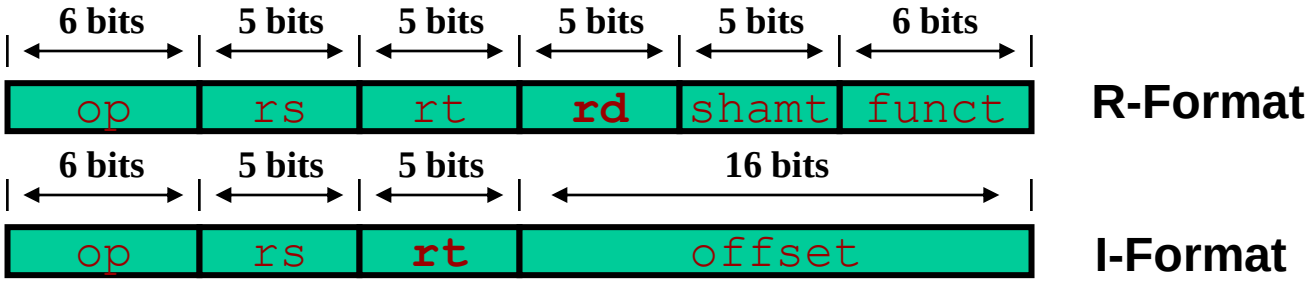


Complete Datapath Executing branch

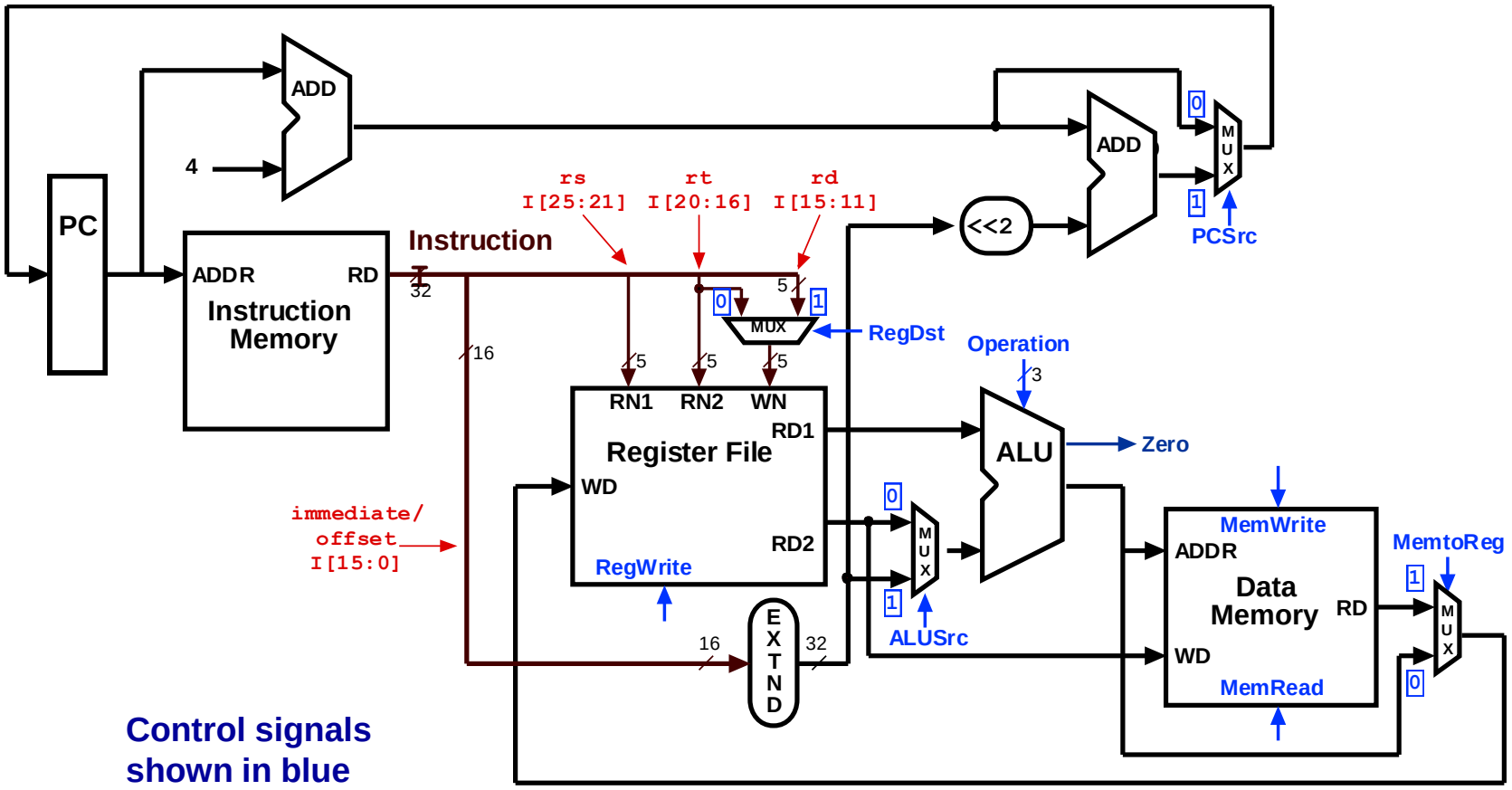


Refining the Complete Datapath

- Depending on the instruction, register file input WN is fed by different fields of the instruction
  - R-Type Instructions: rd field (bits 15:11)
  - Load Instructin: rt field (bits 21:16)
- Result: need an additional multiplexer on WN input



Complete Single-Cycle Datapath

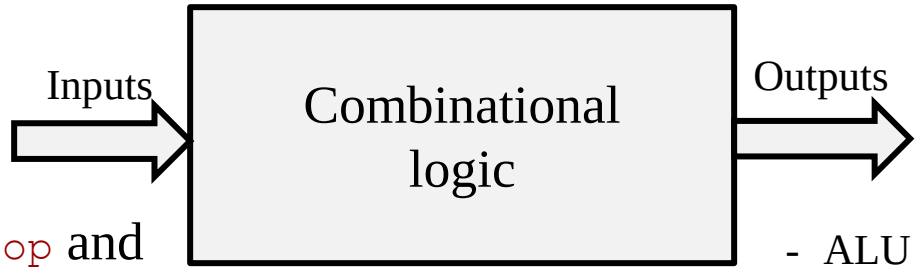


Outline - Processor Implementation

- Overview
- Single-Cycle Implementation
  1. Analyze instruction set; get datapath requirements
  2. Select datapath components and establish clocking methodology
  3. Assemble datapath that meets requirements
  4. Determine control signal values for each instruction
  5. Assemble control logic to generate control signals
- Pipelined Implementation

Control Unit Design

- **Desired function:**
  - **Given an instruction word….**
  - **Generate control signals needed to execute instruction**
- **Implemented as a <u>combinational</u> logic function:**



- Instruction word : `op` and `funct` fields
- ALU result output : `zero`

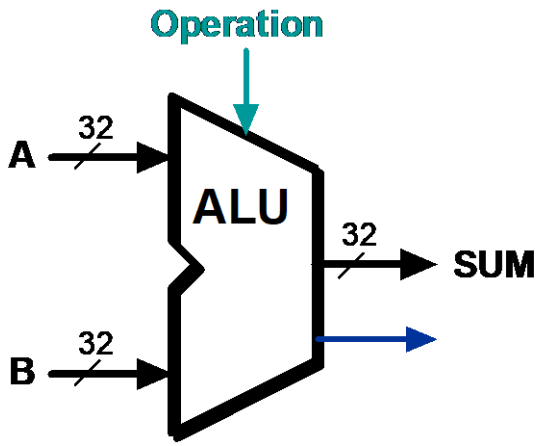
- ALU control signals
- Multiplexer control signals
- Register File & memory control signal

Determining Control Points

- For each instruction type, determine proper value for each control point (control signal)
  - 0
  - 1
  - X (don't care - either 1 or 0)
- Ultimately … use these values to build a truth table

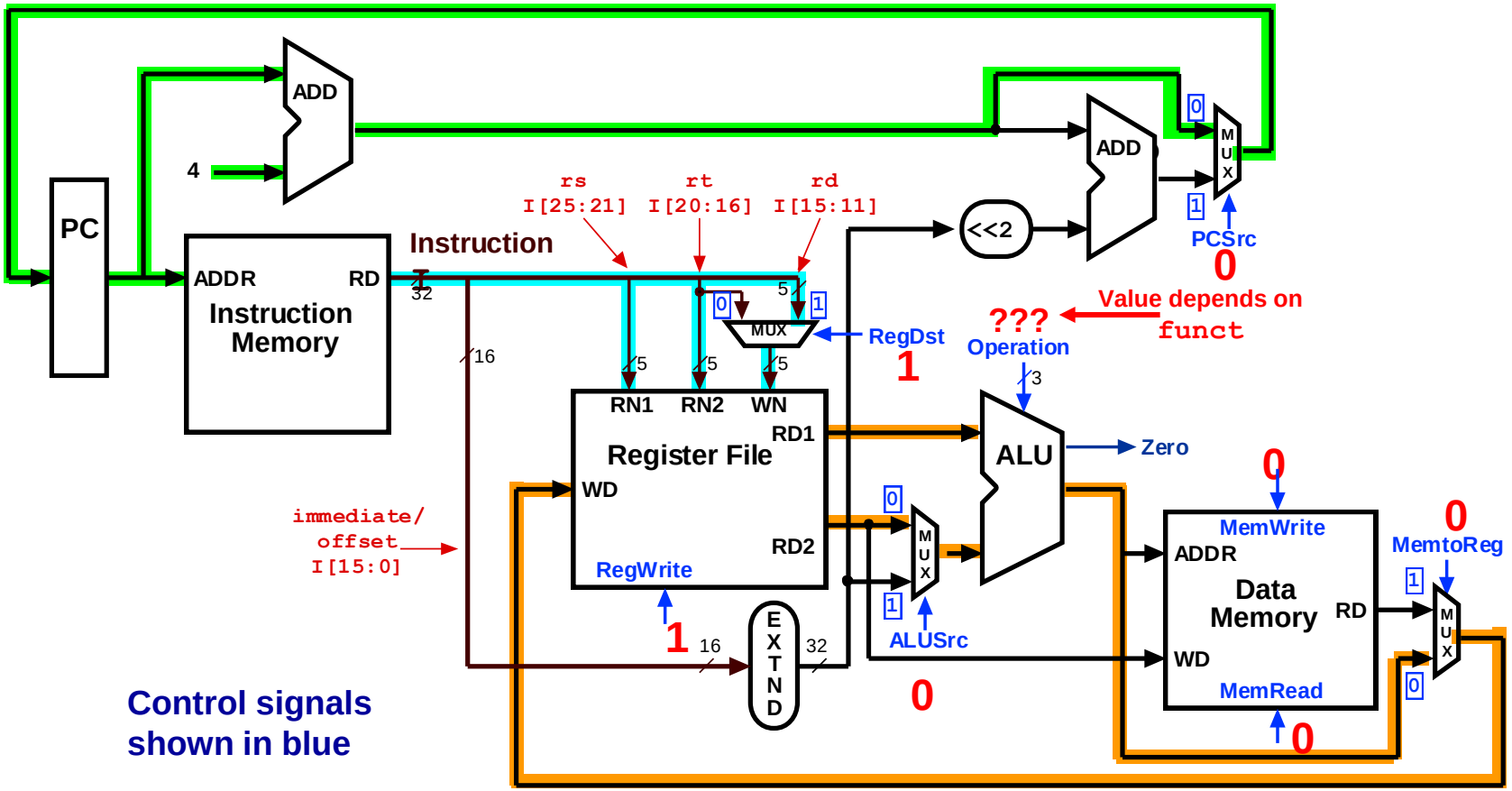
Review: ALU Control Signals

- Functions: Ch. 4 - p. 316 (4th edition)

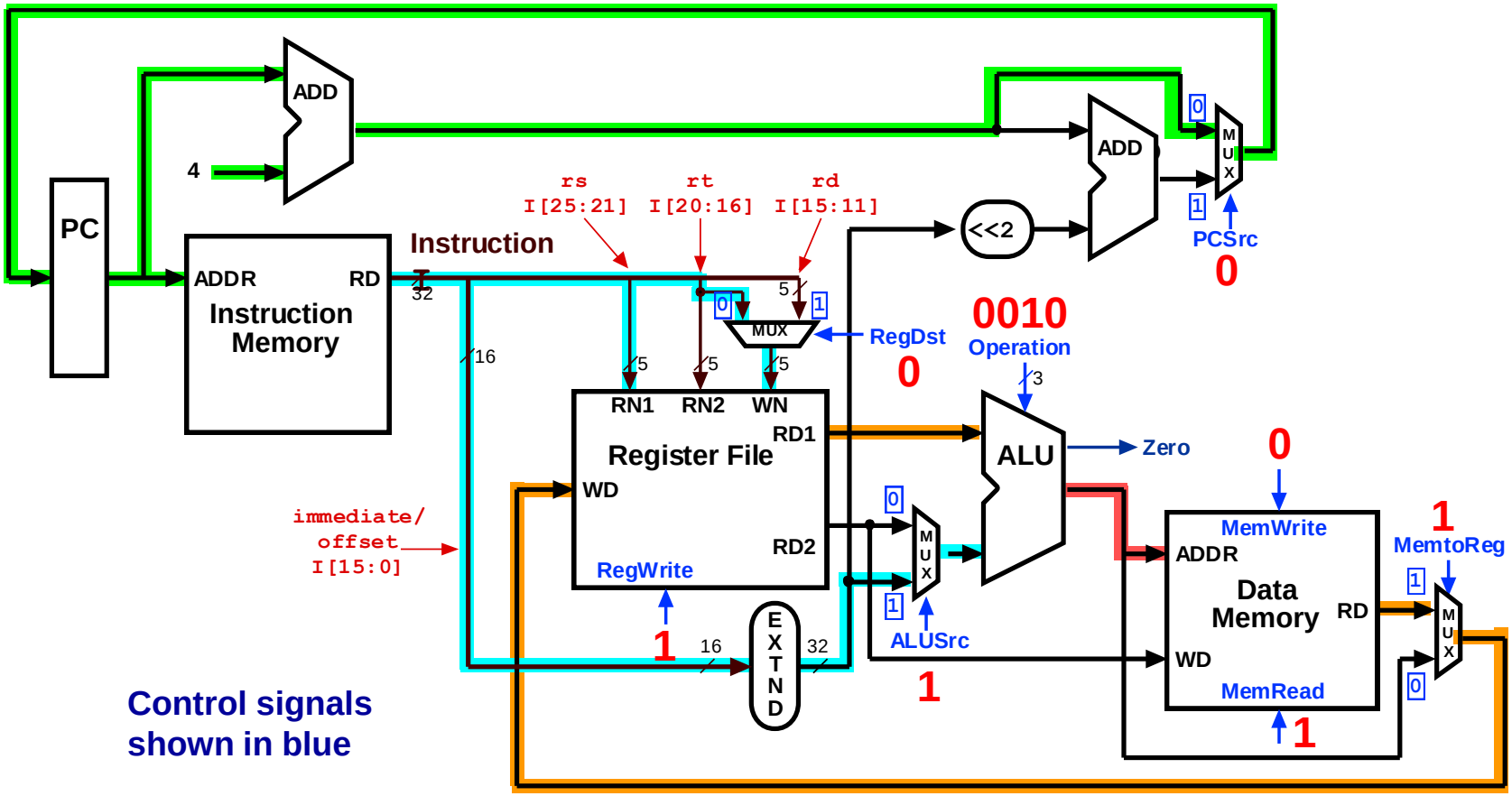


ALU control input	Function
0000	AND
0001	OR
0010	add
0110	sub
0111	slt
1100	NOR

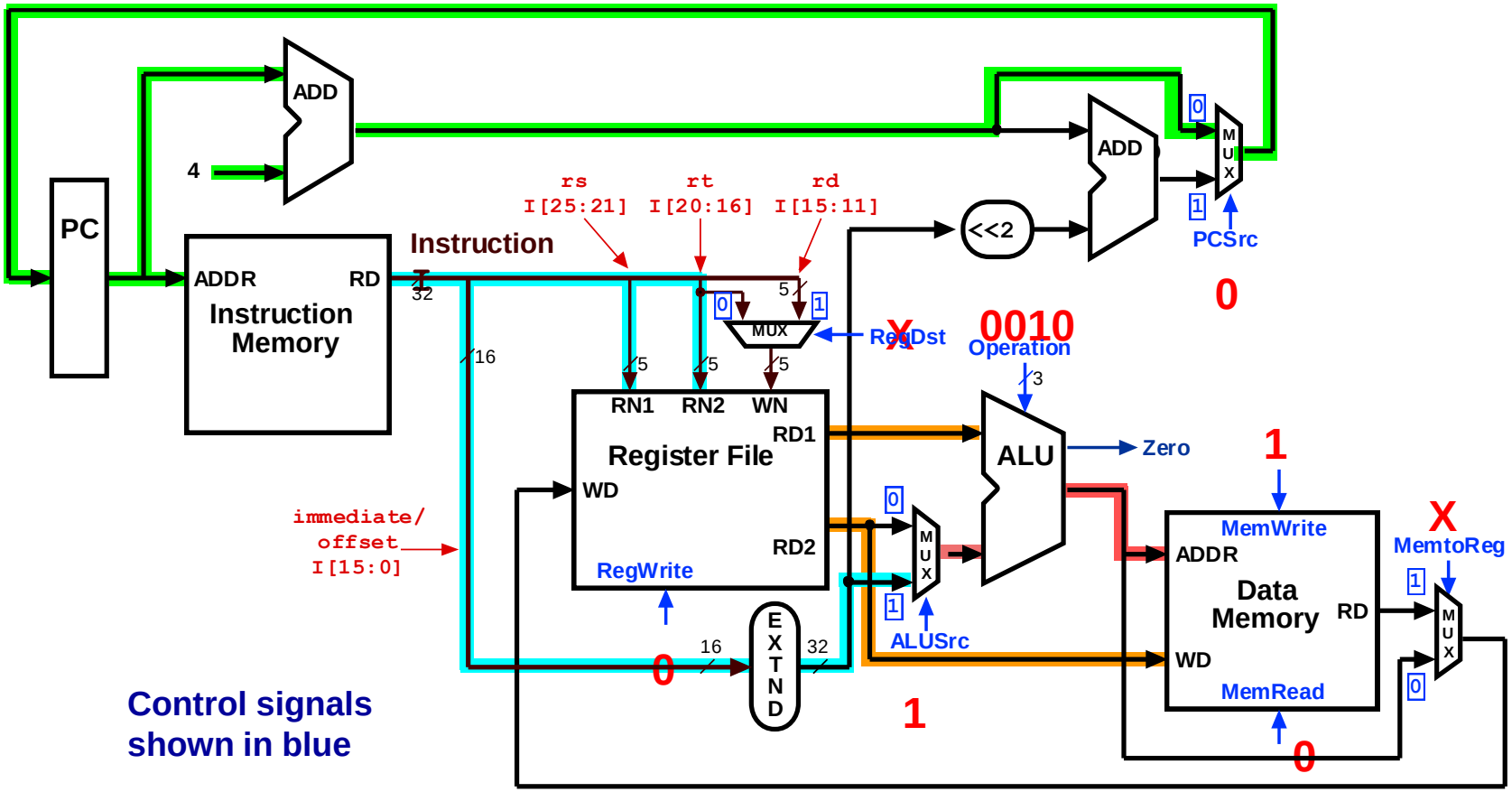
Control Signals - R-Type Instruction



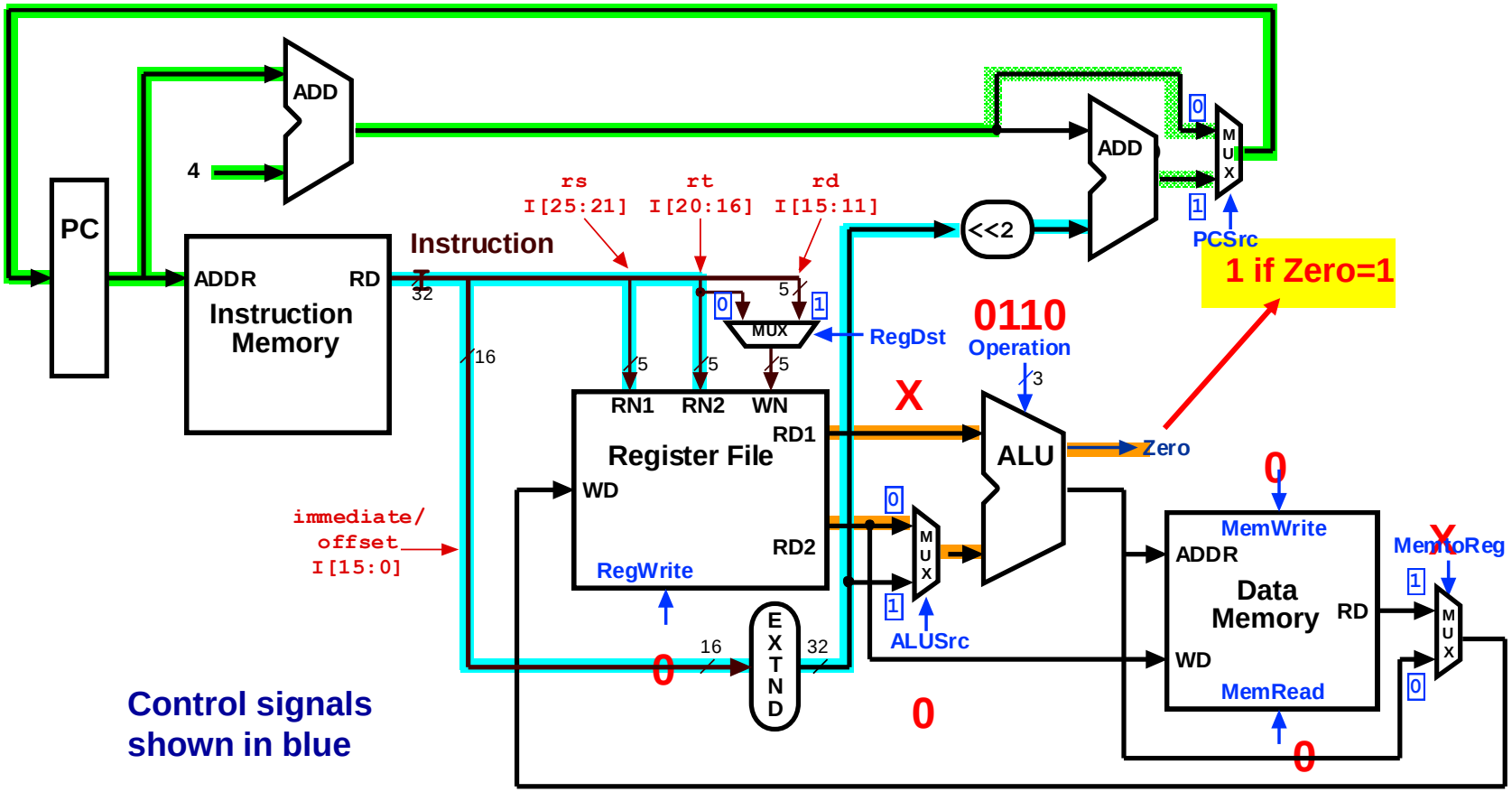
Control Signals - lw Instruction



Control Signals - sw Instruction



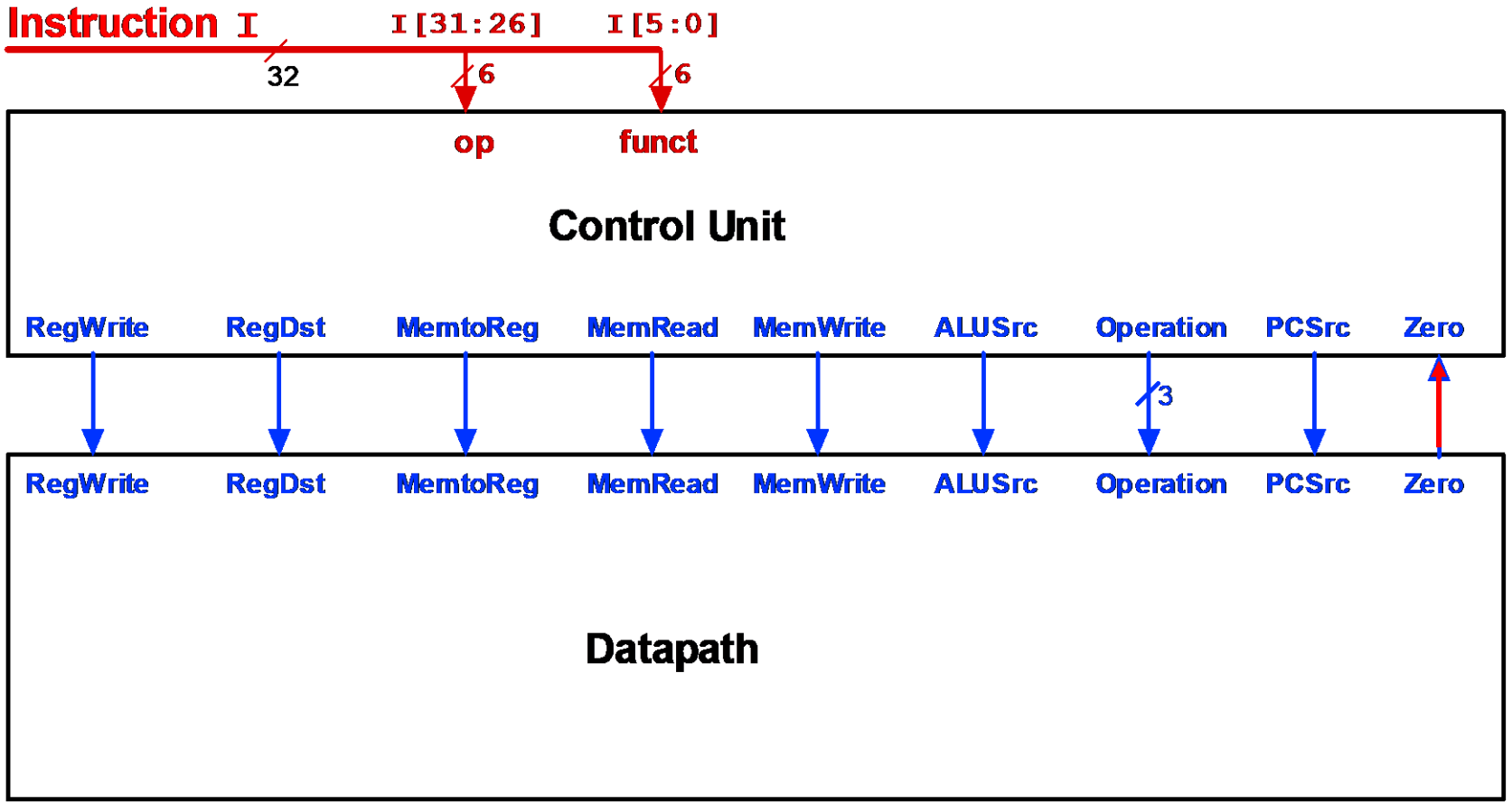
Control Signals - beq Instruction



Outline - Processor Implementation

- Overview
- Single-Cycle Implementation
  1. Analyze instruction set; get datapath requirements
  2. Select datapath components and establish clocking methodology
  3. Assemble datapath that meets requirements
  4. Determine control signal values for each instruction
  5. Assemble control logic to generate control signals
- Pipelined Implementation

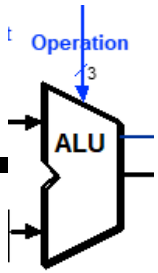
Control Unit Structure



More notes about Control Unit Structure

- Control unit as shown: one huge logic block
- Idea: decompose into smaller logic blocks
  - Smaller blocks can be faster
  - Smaller blocks are easier to work with
- Observation (rephrased):
  - The *function* field is only used to control the ALU operation
  - The *instruction types* can determine the ALU operation
    - transfer type (+)
    - branch type(-)
    - r-format type (depends on the function code)
  - Idea: separate logic for ALU control

Modified Control Unit Structure



Instruction I

I[31:26]

I[5:0]

32

6

6

6

op

funct

Control Unit

aluop

2

aluop

RegWrite RegDst MemtoReg MemRead MemWrite ALUSrc PCSrc Zero

Operation

3

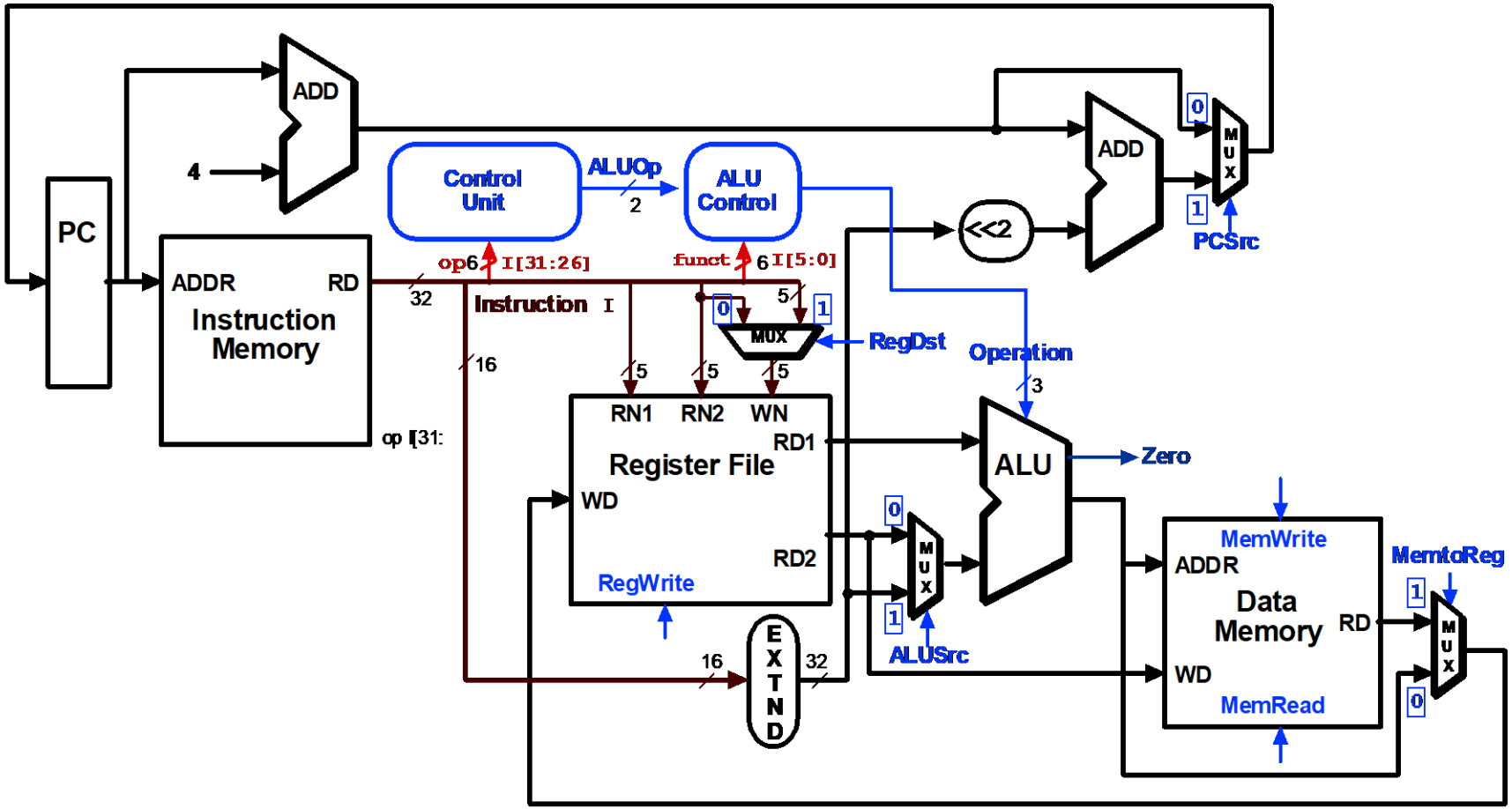
RegWrite RegDst MemtoReg MemRead MemWrite ALUSrc PCSrc Zero

Operation

Datapath

This is called “derived control” or “Local decoding”

Datapath with Modified Control Unit

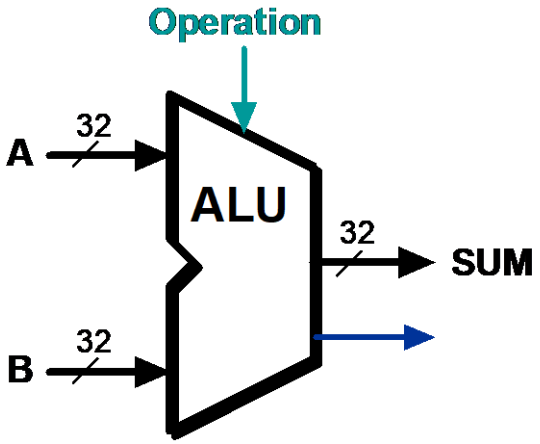


Review from Ch. 4: ALU Function

- **Functions: Ch. 4 - p. 316**

R-Type의 경우

ALU control input	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1000	NOR



ALU Usage in Processor Design

- Usage depends on instruction type
  - Instruction type (specified by opcode)
  - `funct` field (r-type instructions only)
- Encode instruction type in ALUOp signal

For i-format, ALUOp determines ALU ctrl

Instr. type	Operation	funct	Desired Action	ALU Ctl.	ALUOp
data transfer	lw	XXXXXX	add	0010	00
data transfer	sw	XXXXXX	add	0010	00
branch	beq	XXXXXX	subtract	0110	01
r-type	add	100000	add	0010	10
r-type	sub	100010	subtract	0110	10
r-type	and	100100	and	0000	10
r-type	or	100101	or	0001	10
r-type	slt	101010	set on less than	0111	10

For r-format, function code determines ALU ctrl

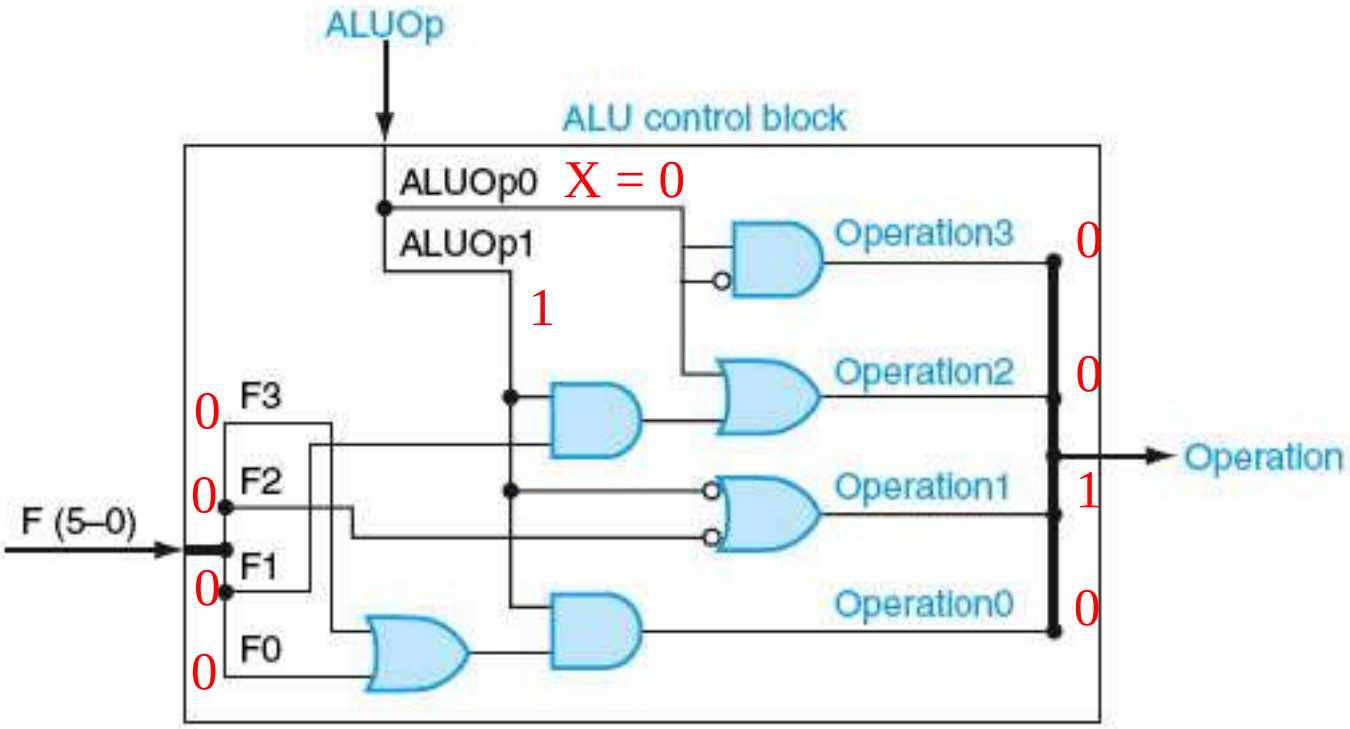
ALU Control - Truth Table (Fig. D.2.1)

- Use don't care values to minimize length
  - Ignore F5, F4 (they are always "10")
  - Assume ALUOp never equals "11"

	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation	
transfer	0	0	X	X	X	X	X	X	0010	
branch	X	1	X	X	X	X	X	X	0110	
Function code	1	X	X	X	0	0	0	0	0010	add
	1	X	X	X	0	0	1	0	0110	sub
	1	X	X	X	0	1	0	0	0000	AND
	1	X	X	X	0	1	0	1	0001	OR
	1	X	X	X	1	0	1	0	0111	slt

ALU Control - Implementation

- **Figure D.2.3, page D-6**



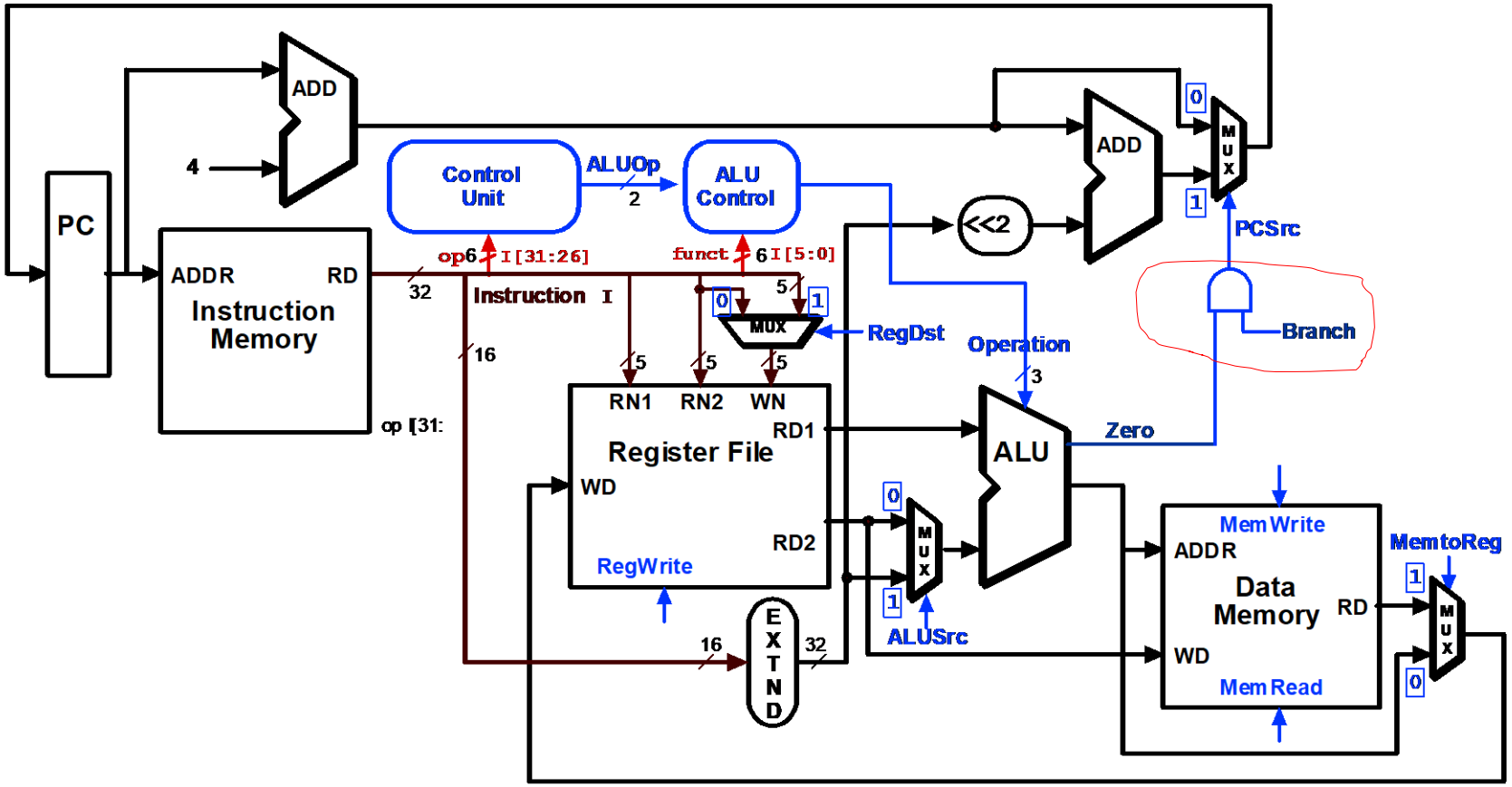
Example

1	X	X	X	0	0	0	0	0010

One More Modification - for Branch

- **BEQ instruction depends on Zero output of ALU**
- **No other instruction uses Zero output (yet)**
- **Local decoding**
  - **Implement with new "Branch" control signal**
  - **Add AND gate to generate PCSelect**

Processor Design - Branch Modification



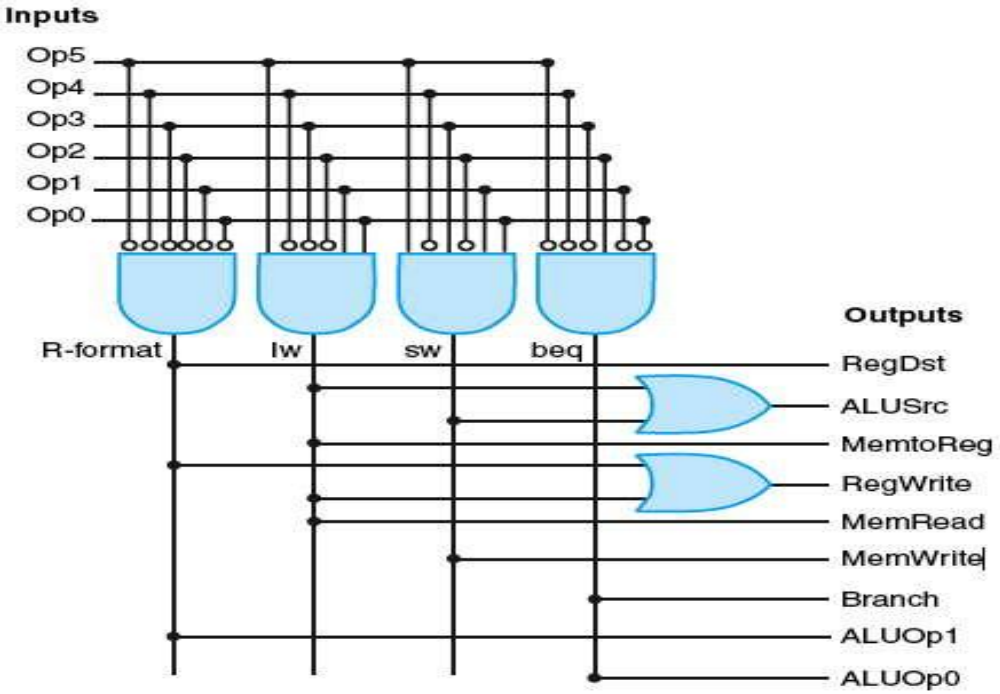
Control Unit Implementation

- **Review: Opcodes for key instructions**
- **Control Unit Truth Table: Fill in the blanks (or see Fig. 4-18, p. 323)**
- **Implementation: Decoder + 2 Gates (Fig. D.2.4)**

Input							Output								
OP	Op5	Op4	Op3	Op2	Op1	Op0	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
RT	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
sw	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0
beq	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1

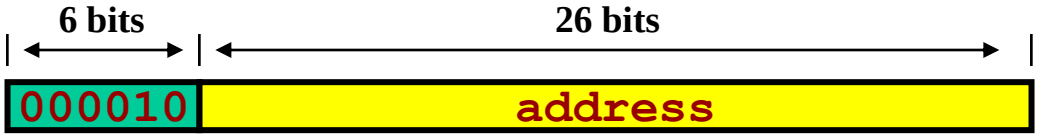
Control Unit Implementation (Fig. D.2.5)

Input							Output								
OP	Op5	Op4	Op3	Op2	Op1	Op0	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
RT	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0
sw	1	0	1	0	1	1	X	1	X	0	0	1	0	0	0
beq	0	0	0	1	0	0	X	0	X	0	0	0	1	0	1



Final Extension: Implementing `j` (jump)

- **Instruction Format**



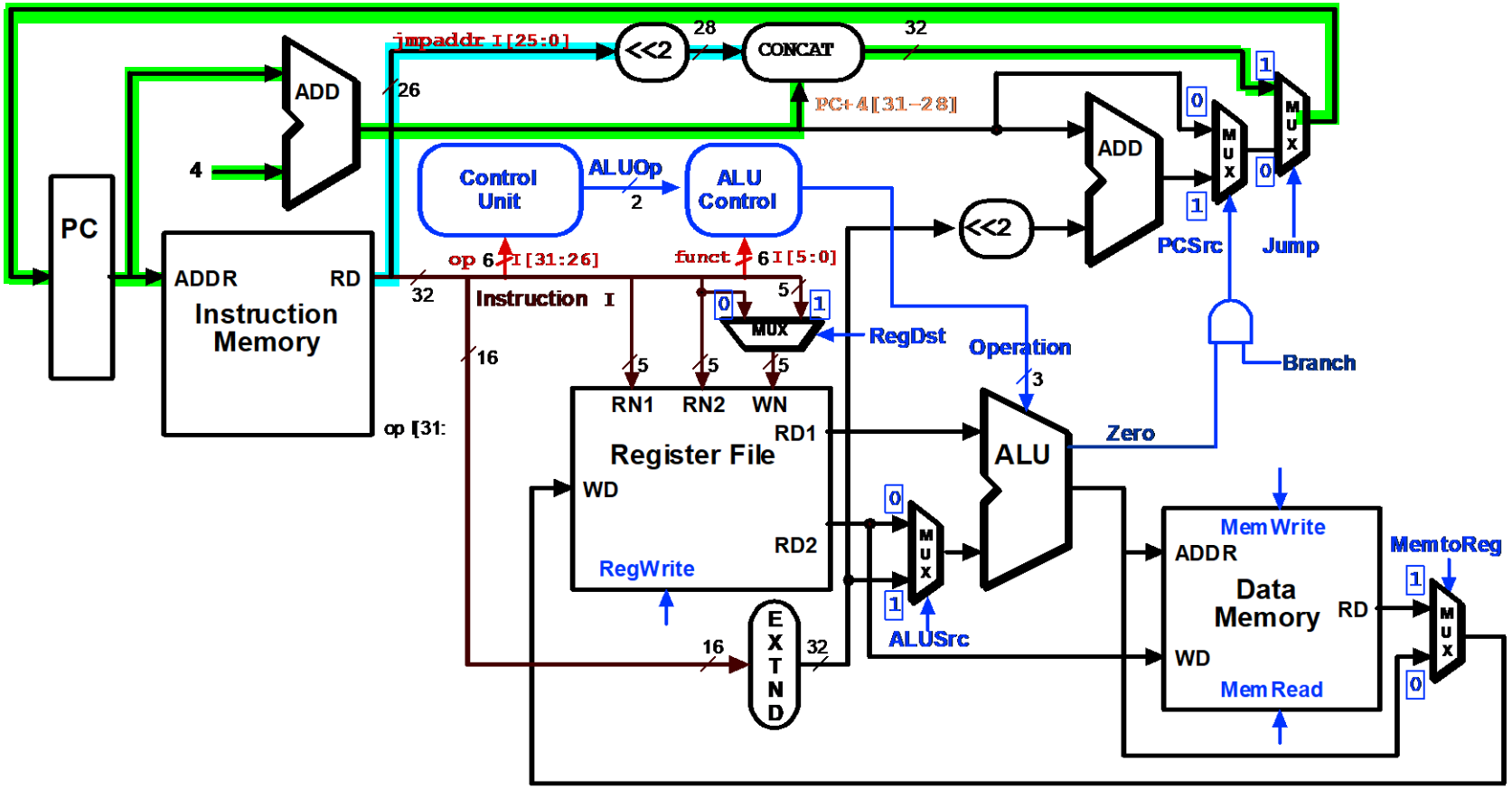
J-Format

- **Register Transfer:**

```
PC <- (PC + 4)[31:28] @ ( I[25:0] << 2 )
```

- **Remember, it's <u>unconditional</u>**

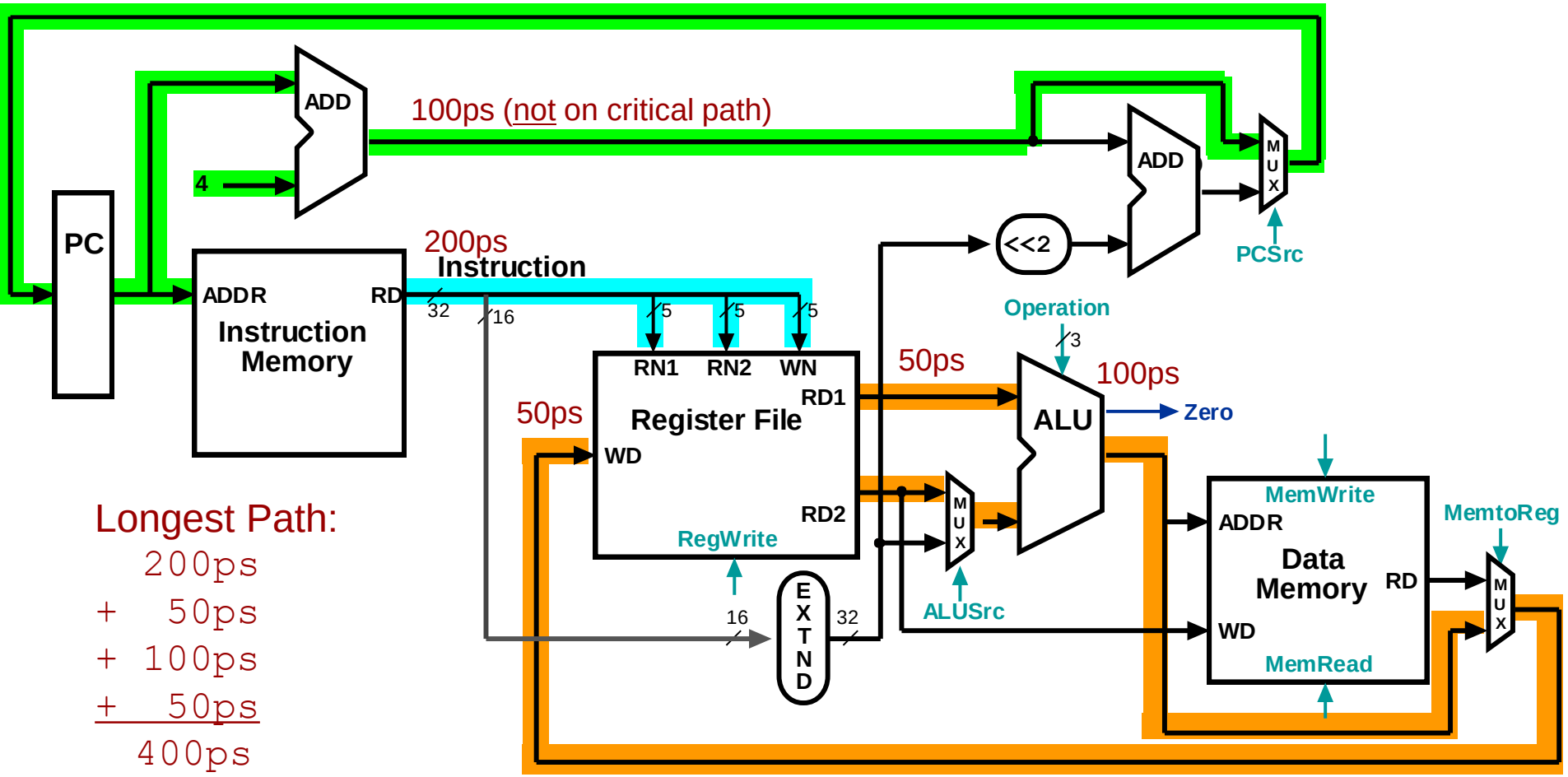
Final Extension: Implementing jump



Single-Cycle Processor Performance

- Example: suppose we have the following delays
  - Memory read/write 200ps
  - ALU and adders 100ps
  - Register File read/write 50ps
  - Control - negligible 0ps
  - Mux - negligible 0ps
- What is the critical path for each instruction?

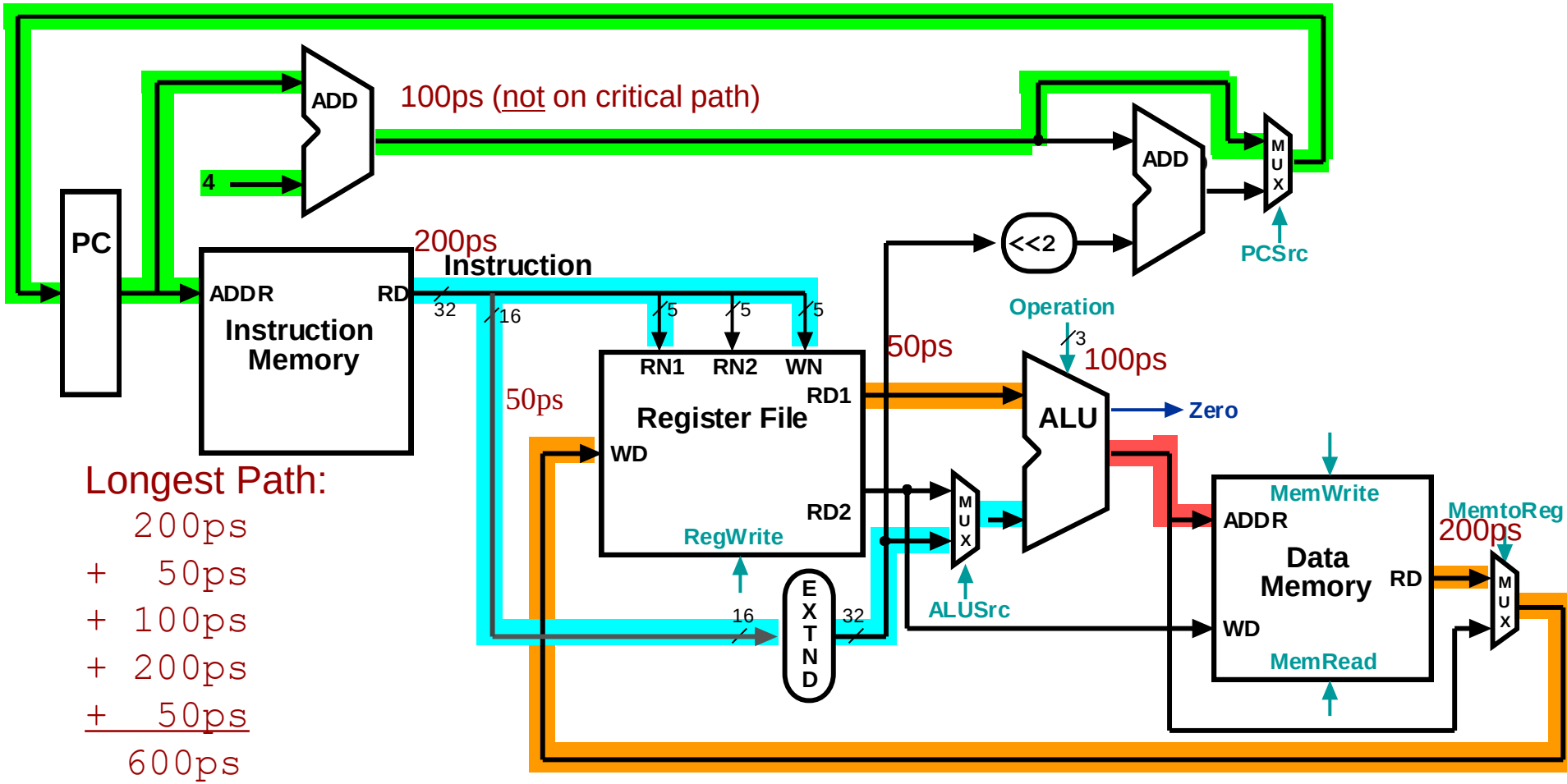
Critical Path for R-Type Instructions



Longest Path:

```
  200ps
+  50ps
+ 100ps
+  50ps
  400ps
```

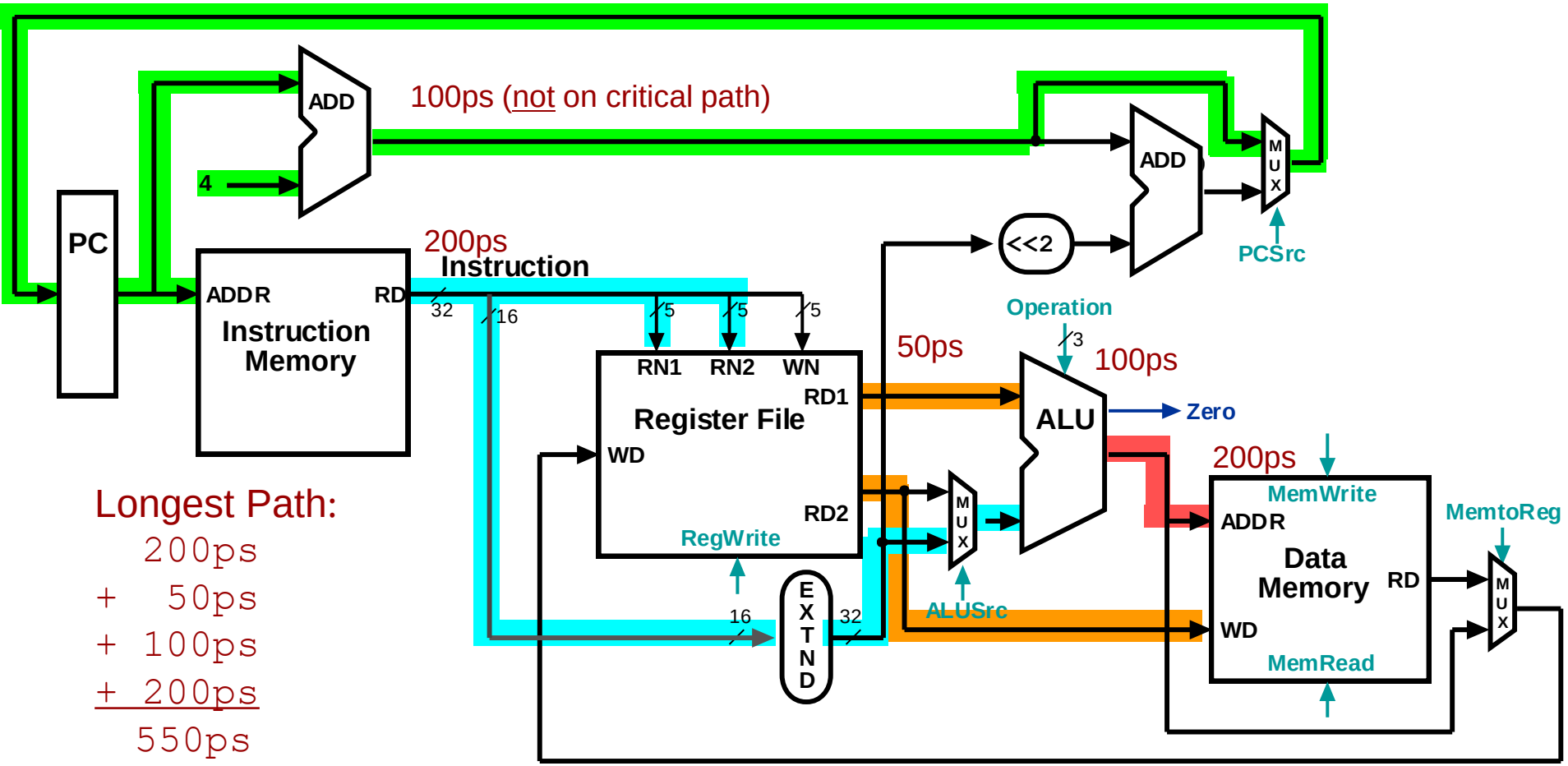
Critical Path for `lw` Instruction



Longest Path:

```
  200ps
+  50ps
+ 100ps
+ 200ps
+  50ps
  600ps
```

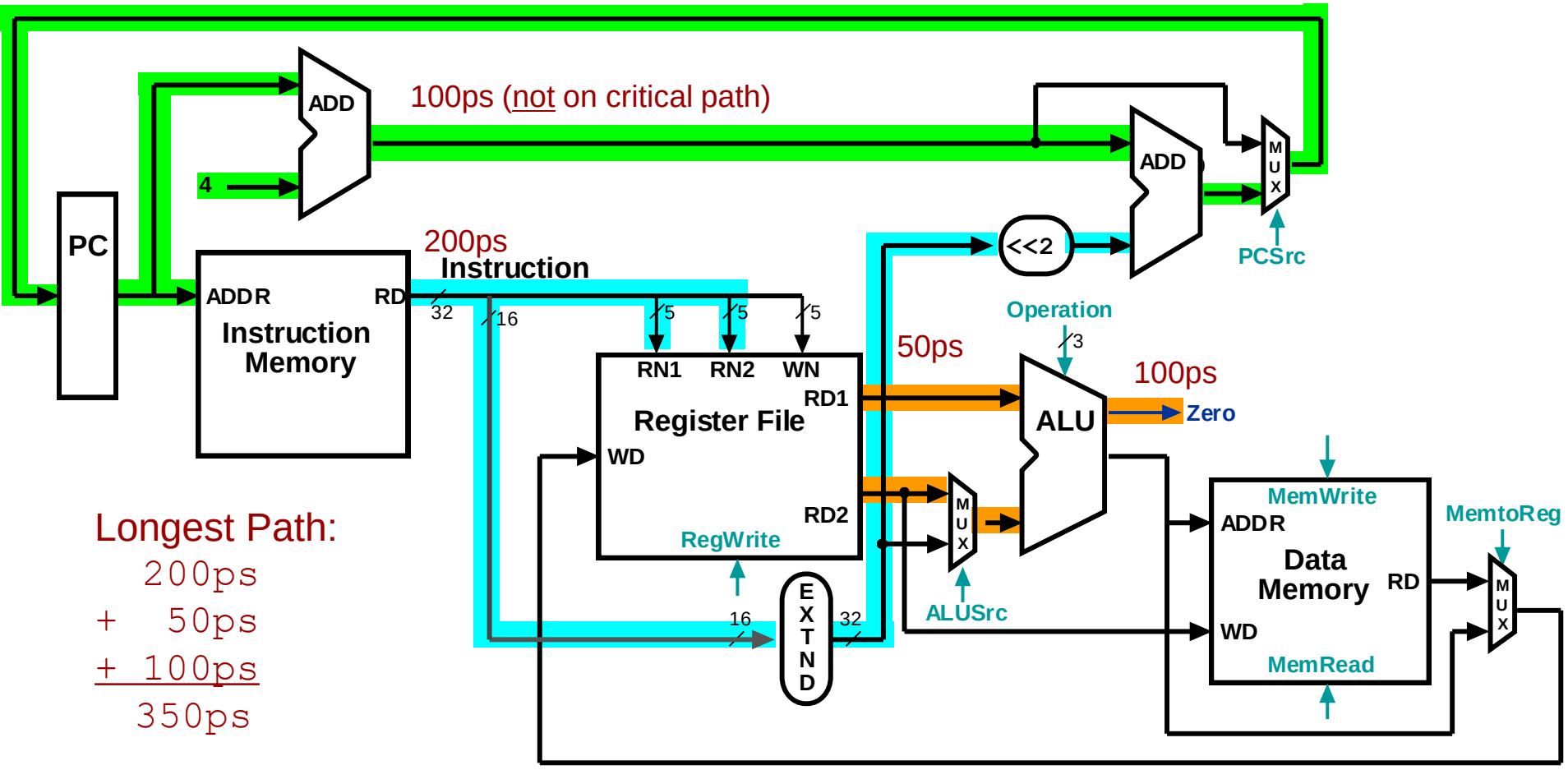
Critical Path for `sw` Instruction



Longest Path:

200ps
+ 50ps
+ 100ps
+ 200ps
550ps

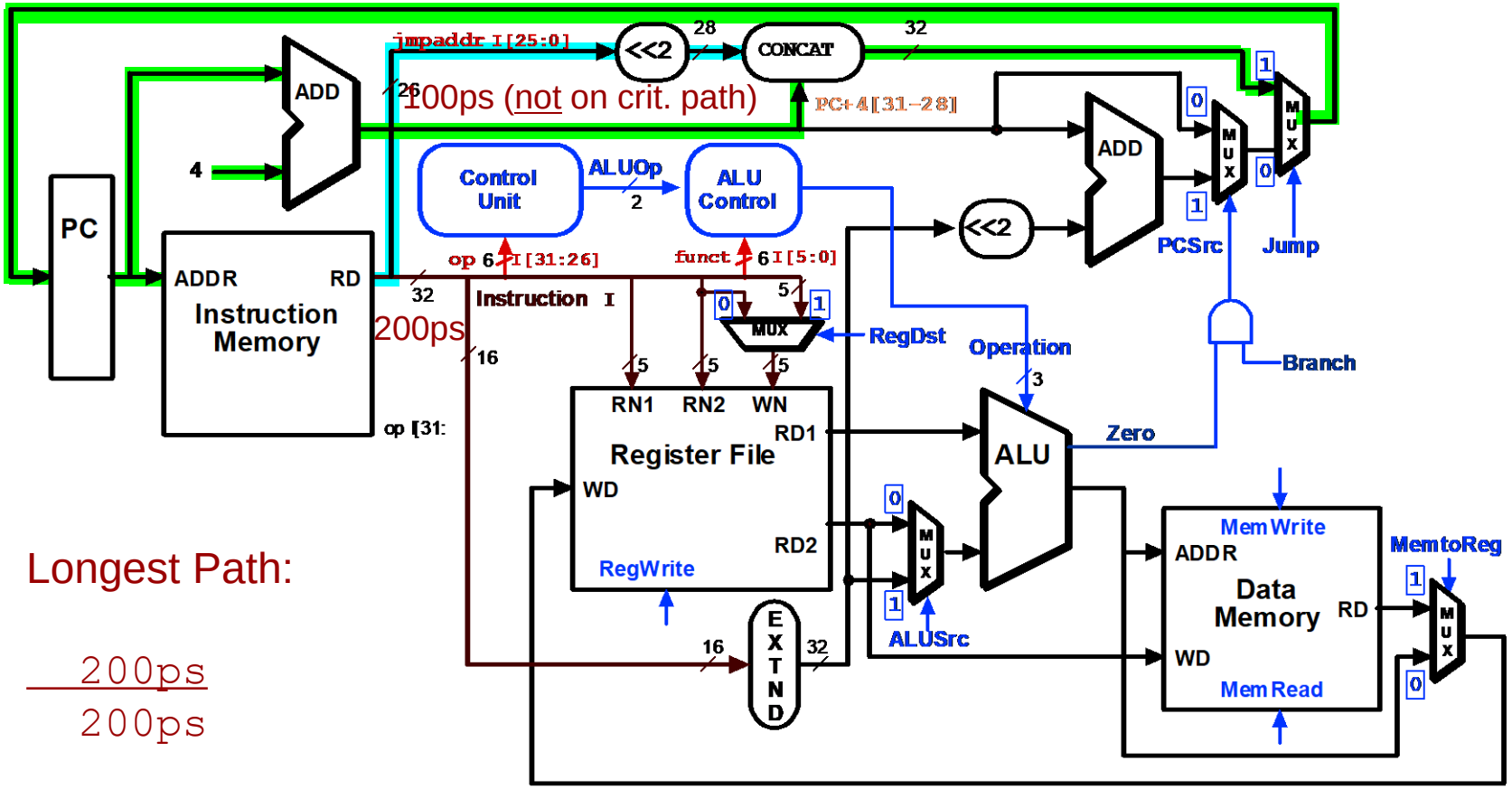
Critical Path for `beq` Instruction



Longest Path:

200ps
+ 50ps
+ 100ps
350ps

Critical Path for `j` (jump) Instruction



Longest Path:

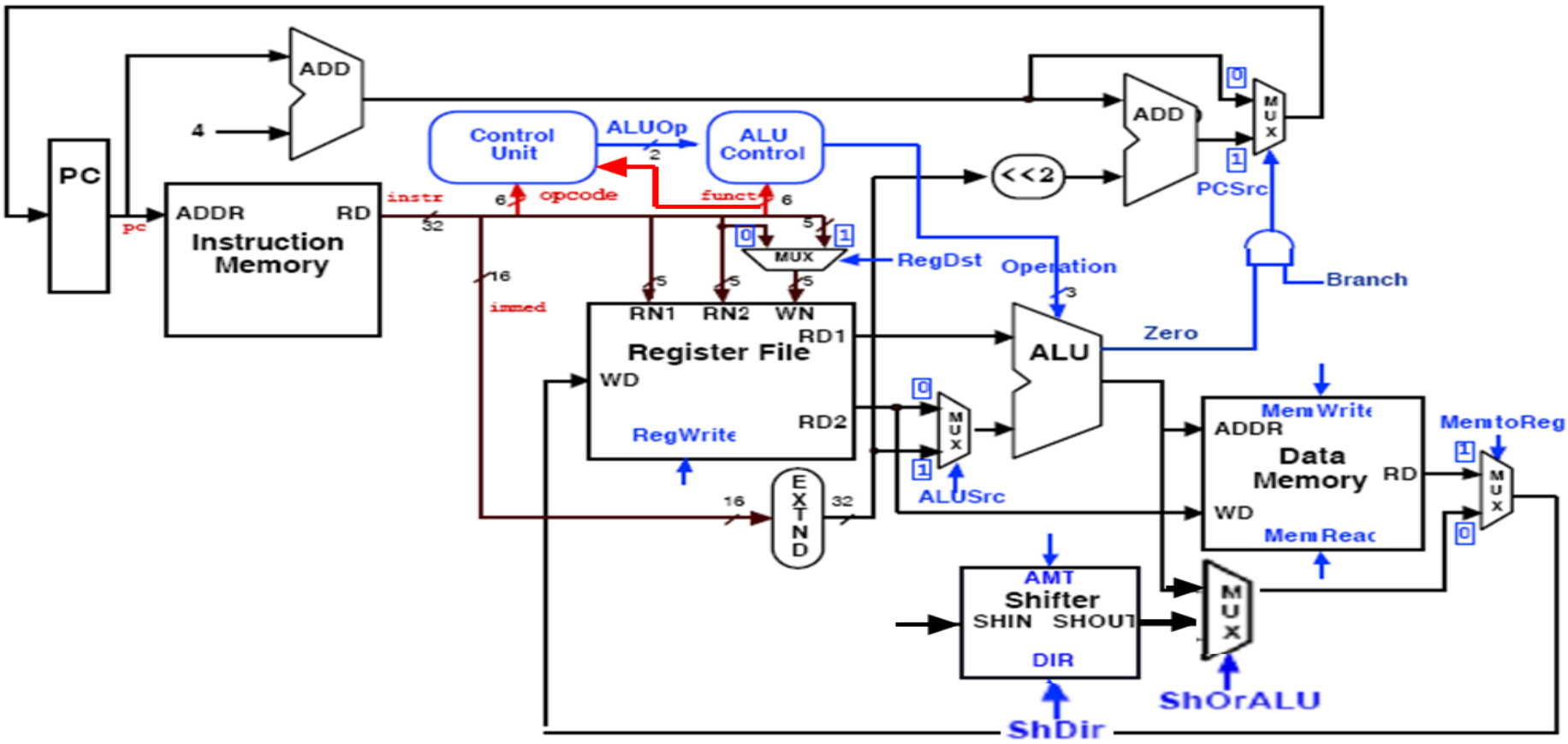
200ps

200ps

Alternatives to Single-Cycle

- Multicycle Processor Implementation
  - Shorter clock cycle
  - Multiple clock cycles per instruction
  - Some instructions take more cycles than others
  - Less hardware required
- Pipelined Implementation
  - Overlap execution of instructions
  - Try to get short cycle times and low CPI
  - More hardware required … but also more performance!

- srl rd, rt, shamt 명령어를 완성하는 문제



Shift => 3ns, Mem Fetch => 2ns, Reg. Read/Write => 1ns, ALU OP => 2ns)
SRL 명령어의 critical path 지연시간은?
LW 명령어의 critical path 지연시간은?

Outline - Processor Implementation

- Overview
- Single-Cycle Implementation
- Single-Cycle Verilog Model