Lecture 07: RISC-V Single-Cycle Implementation

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

Acknowledgements

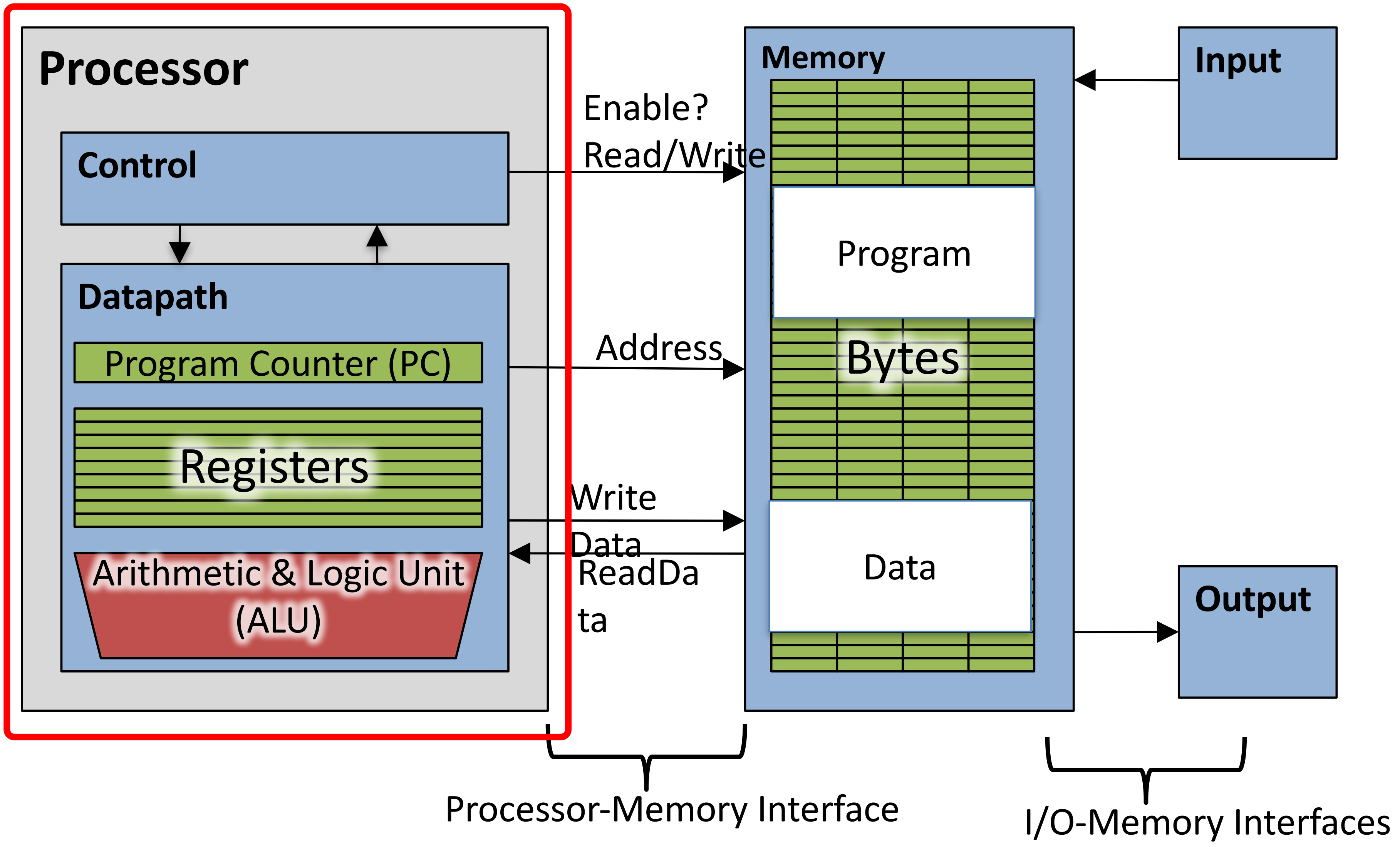
- The notes cover Appendix C of the textbook
  - Slides for general RISC ISA implementation are adapted from Lecture slides for “Computer Organization and Design, RISC-V Edition: The Hardware/Software Interface” textbook for general RISC ISA implementation
  - Slides for RISC-V single-cycle implementation are adapted from Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley

Review

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- ISA simple subset, shows most aspects
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j

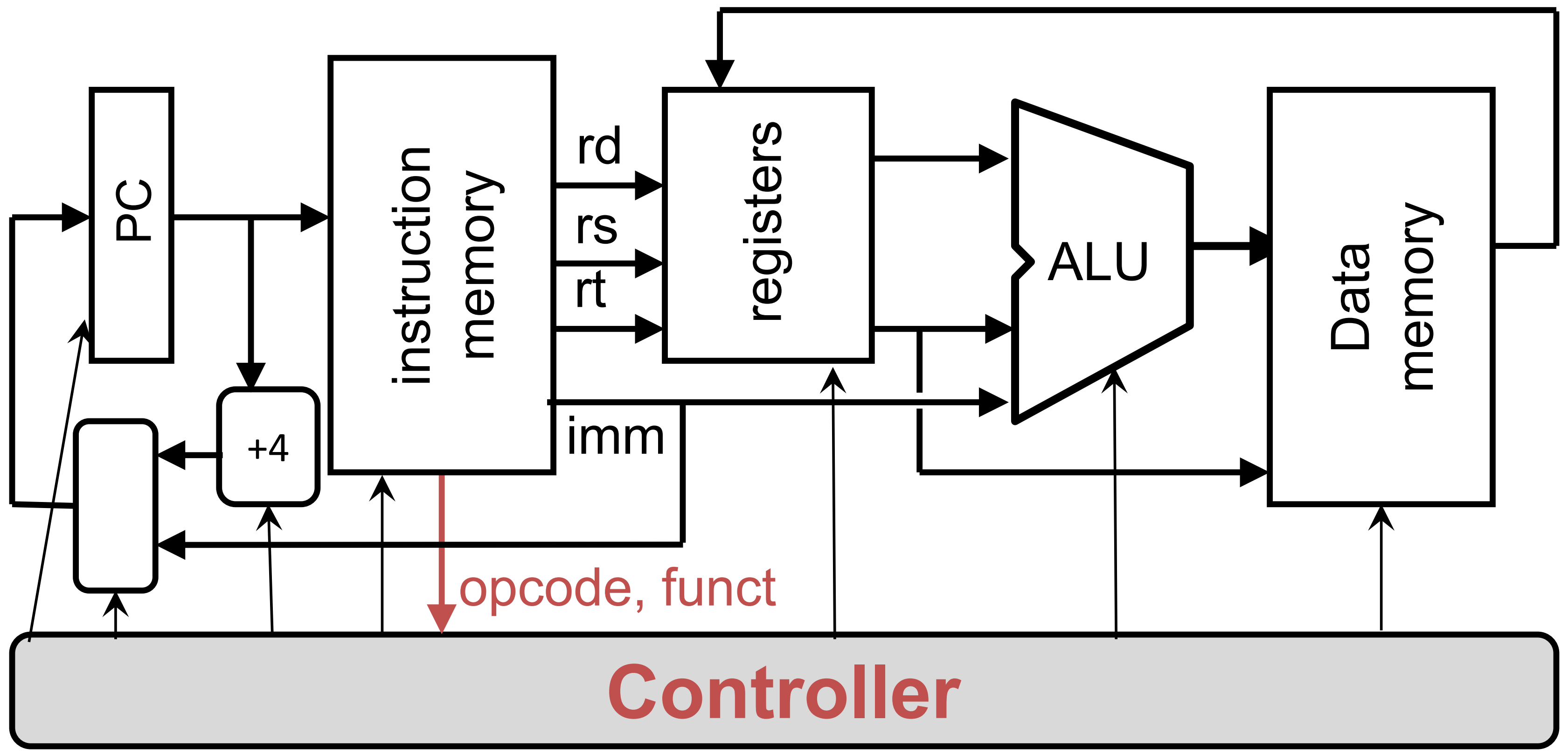
$$CPU\ \text{Time} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Components of a Computer



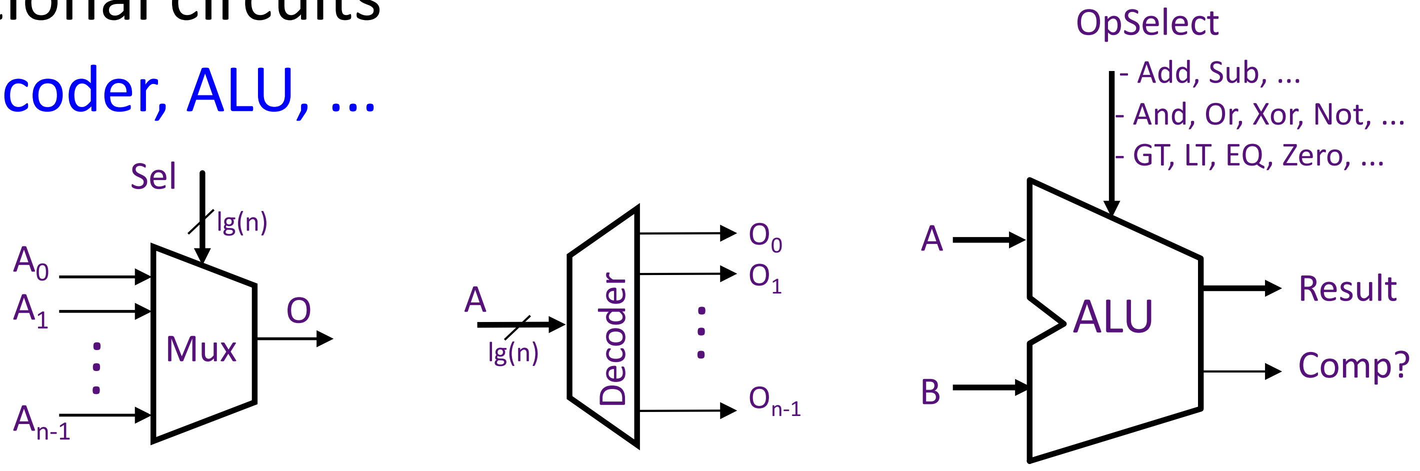
Datapath and Control

- Datapath designed to support data transfers required by instructions
- Controller causes correct transfers to happen

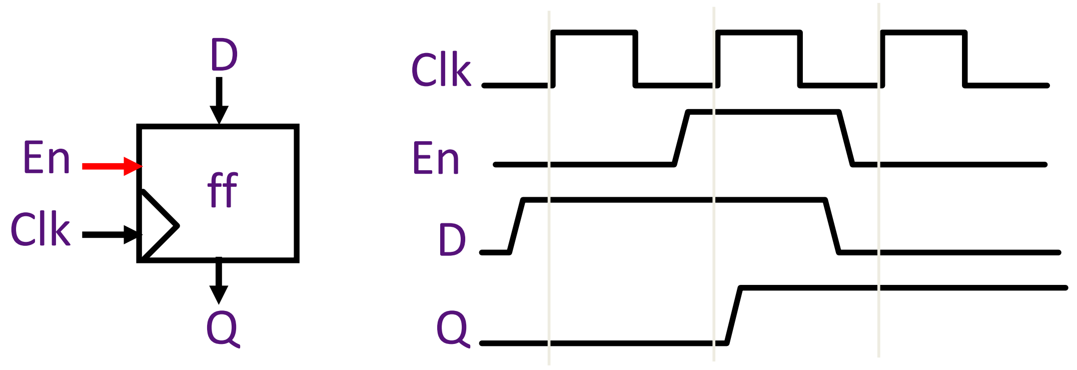


Hardware Elements of CPU

- Combinational circuits
  - Mux, Decoder, ALU, ...

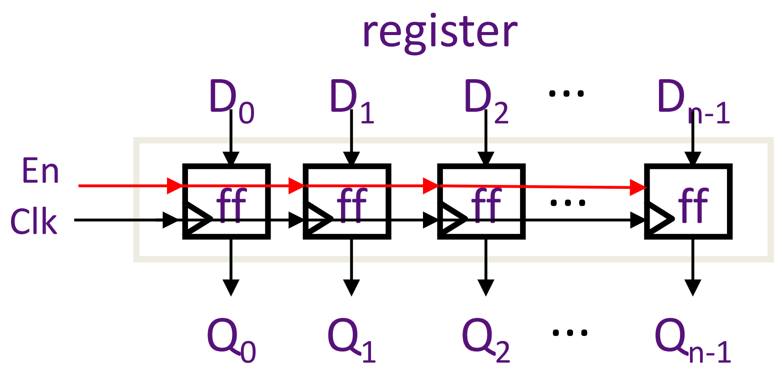


- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM

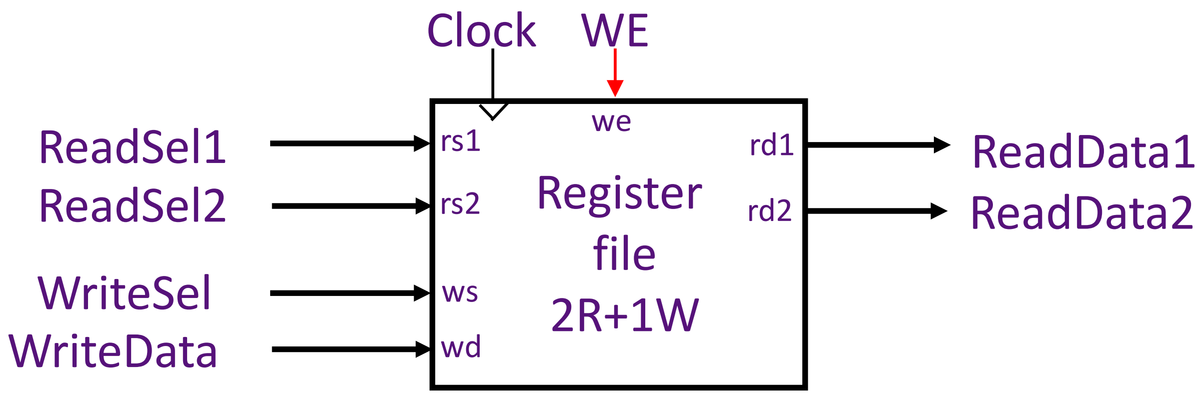


Edge-triggered: Data is sampled at the rising edge

Register Files

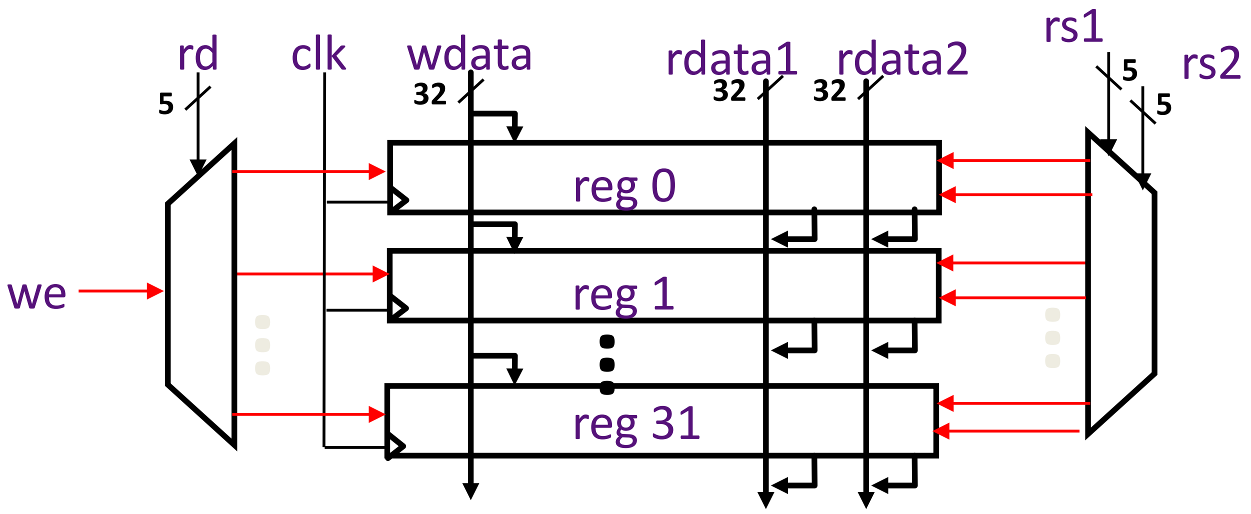


- Reads are combinational
  - Can read in any cycle and for multiple reads
  - Only 2 register source operands needed

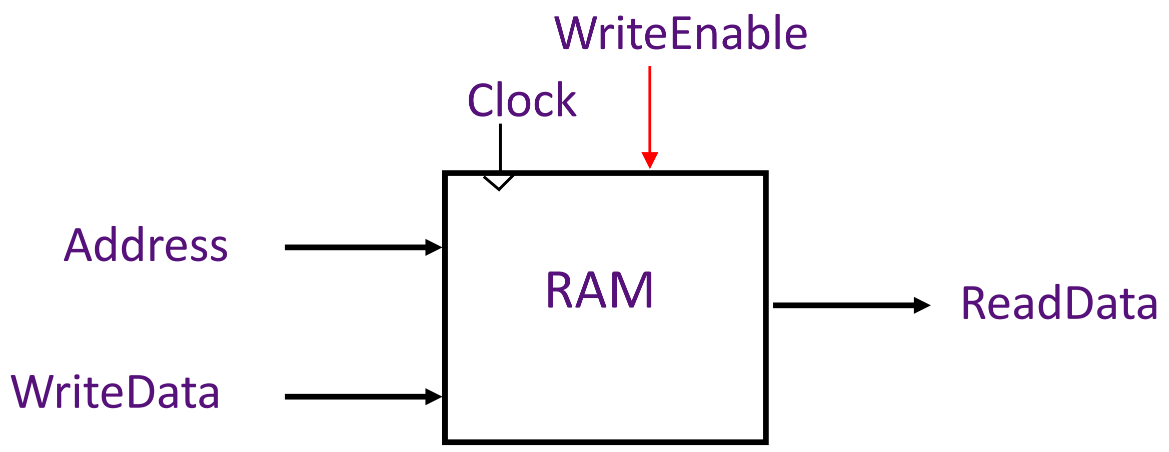


Register File Implementation

- RISC-V integer instructions have at most 2 register source operands



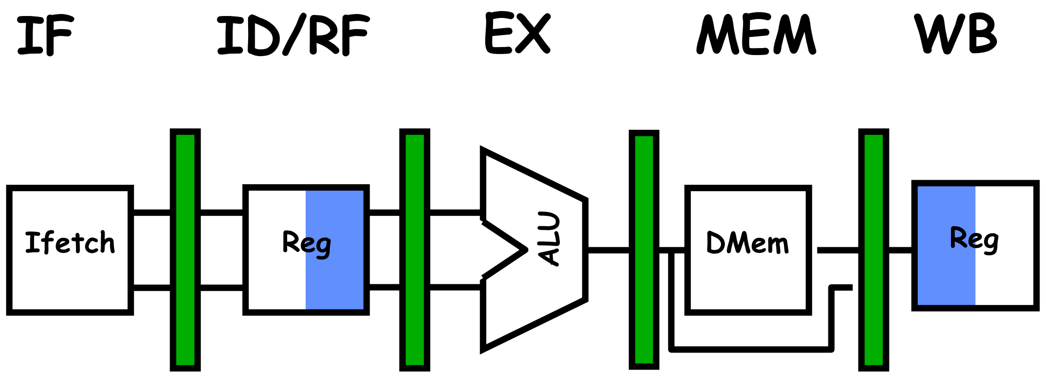
A Simple Memory Model



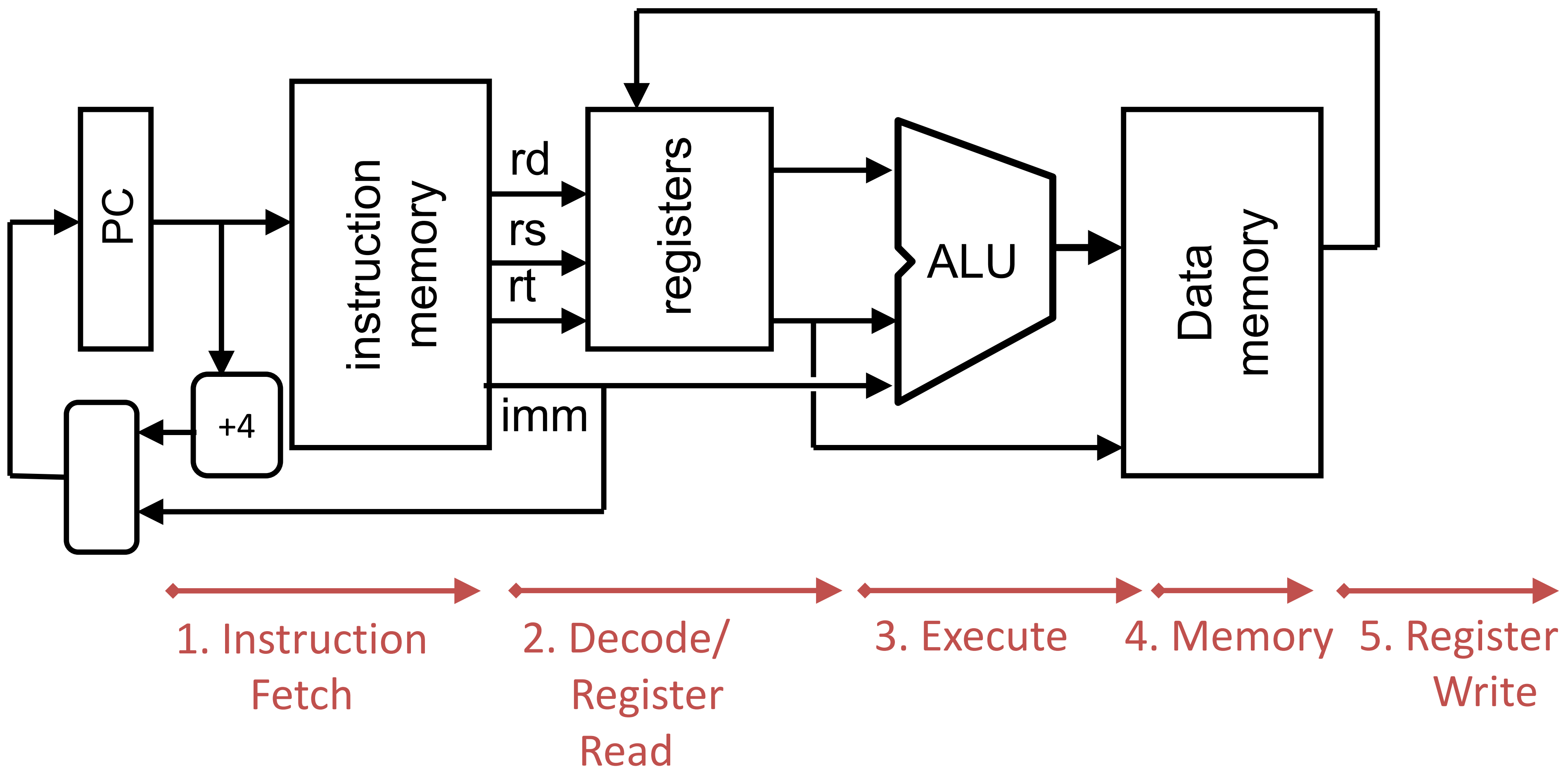
- Reads and writes are always completed in one cycle
- Read can be done any time (i.e. combinational)
- Write is performed at the rising clock edge
  - if it is enabled

Five Stages of Instruction Execution

- Stage 1: Instruction Fetch
- Stage 2: Instruction Decode
- Stage 3: ALU (Arithmetic-Logic Unit)
- Stage 4: Memory Access
- Stage 5: Register Write

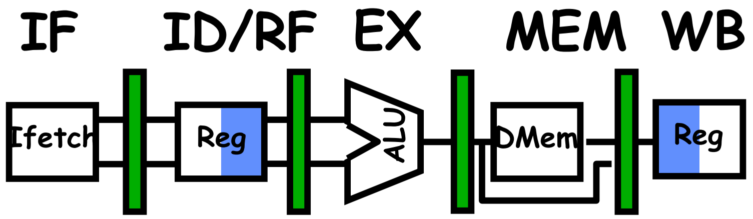


Stages of Execution and Datapath



Stages of Execution (1/5)

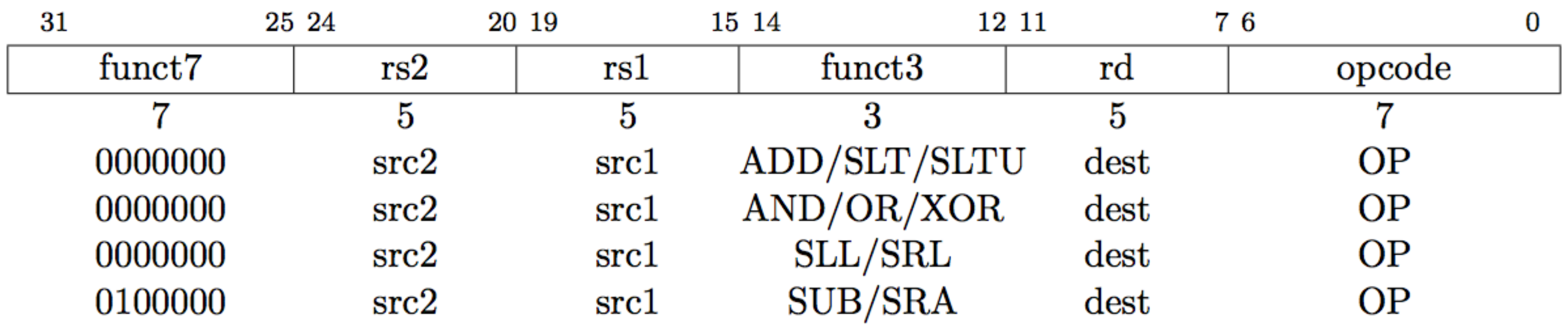
- A wide variety of instructions: so what general steps do they have in common?
  - Focus:
    - Memory reference: lw, sw
    - Arithmetic/logical: add, sub, and, or, slt
    - Control transfer: beq, j



- Stage 1: Instruction Fetch
  - The 32-bit instruction word must first be fetched from memory
    - The cache-memory hierarchy
  - Increment PC
    - PC = PC + 4, to point to the next instruction
      - byte addressing so + 4

Stages of Execution (2/5)

- Stage 2: Instruction Decode: gather data from the fields (decode all necessary instruction data)
  1. Read the opcode to determine instruction type and field lengths
  2. Read in data from all necessary registers
     - For add/store/conditional branch, read two registers
     - For addi/load, read one register
     - For jal, no reads necessary



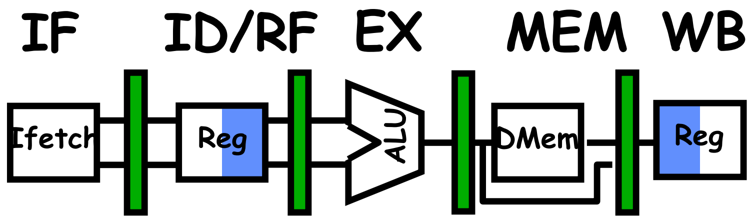
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Stages of Execution (3/5)

- Stage 3: ALU (Arithmetic-Logic Unit): the real work of most instructions is done here
  - AL operations:
    - arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)
  - Loads and stores: addition is done
    - lw $R4, 40($R1)
    - Add to calcuate the the address for accessing memory
      - [$R4] + 40
  - Conditional branch: comparison is done in this stage (one solution)
    - BEQ $R3, $R4, 128
      - [$R3] - [$R4]

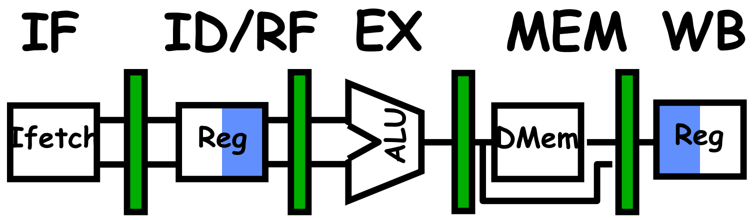
Stages of Execution (4/5)

- Stage 4: Memory Access: only for load and store
  - The other instructions remain idle during this stage or skip it all together
  - Since load/store have a unique step, we need this extra stage to account for them
  - As a result of the cache system, this stage is expected to be fast
    - 1 cycle ideally

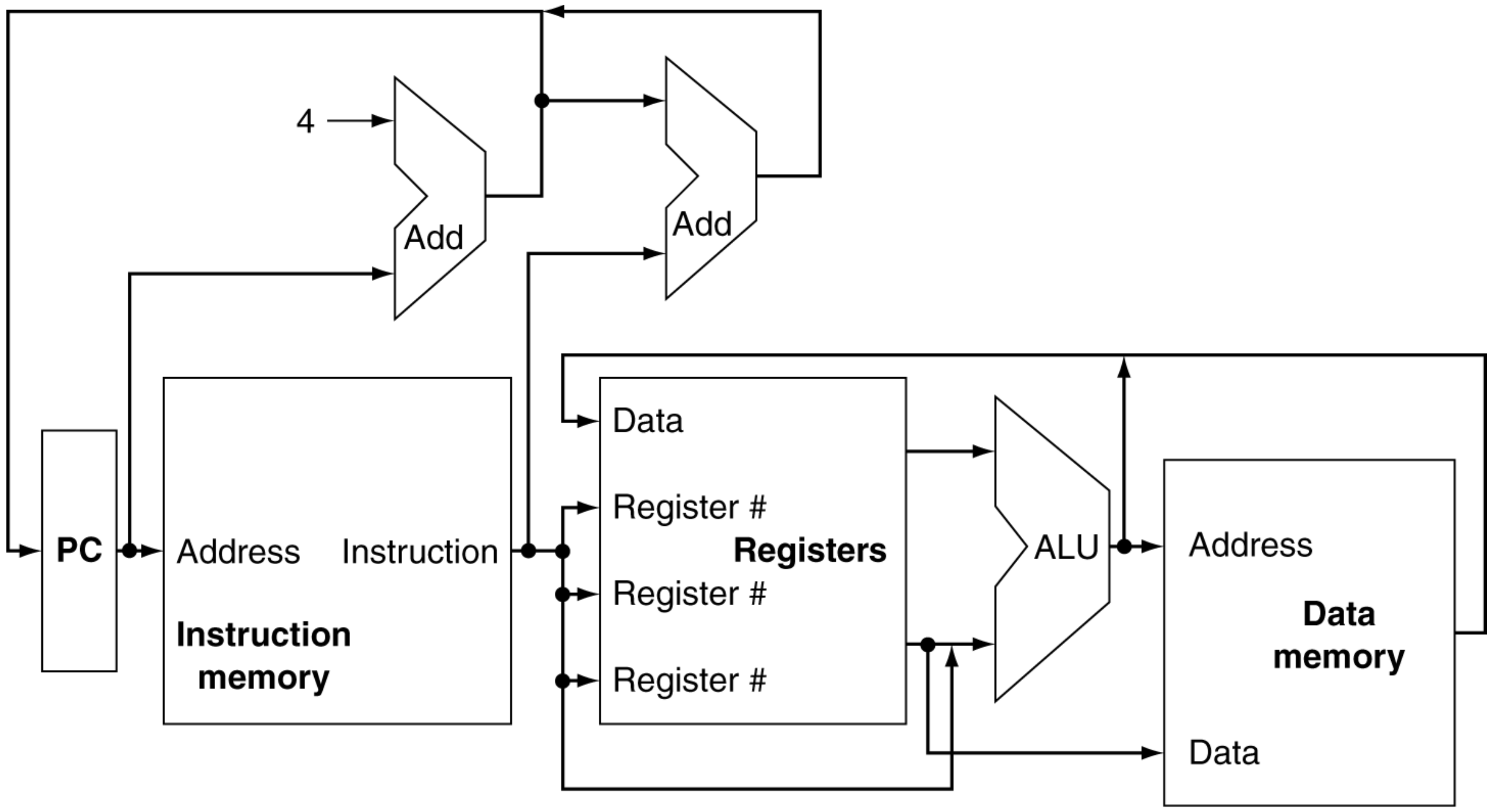


Stages of Execution (5/5)

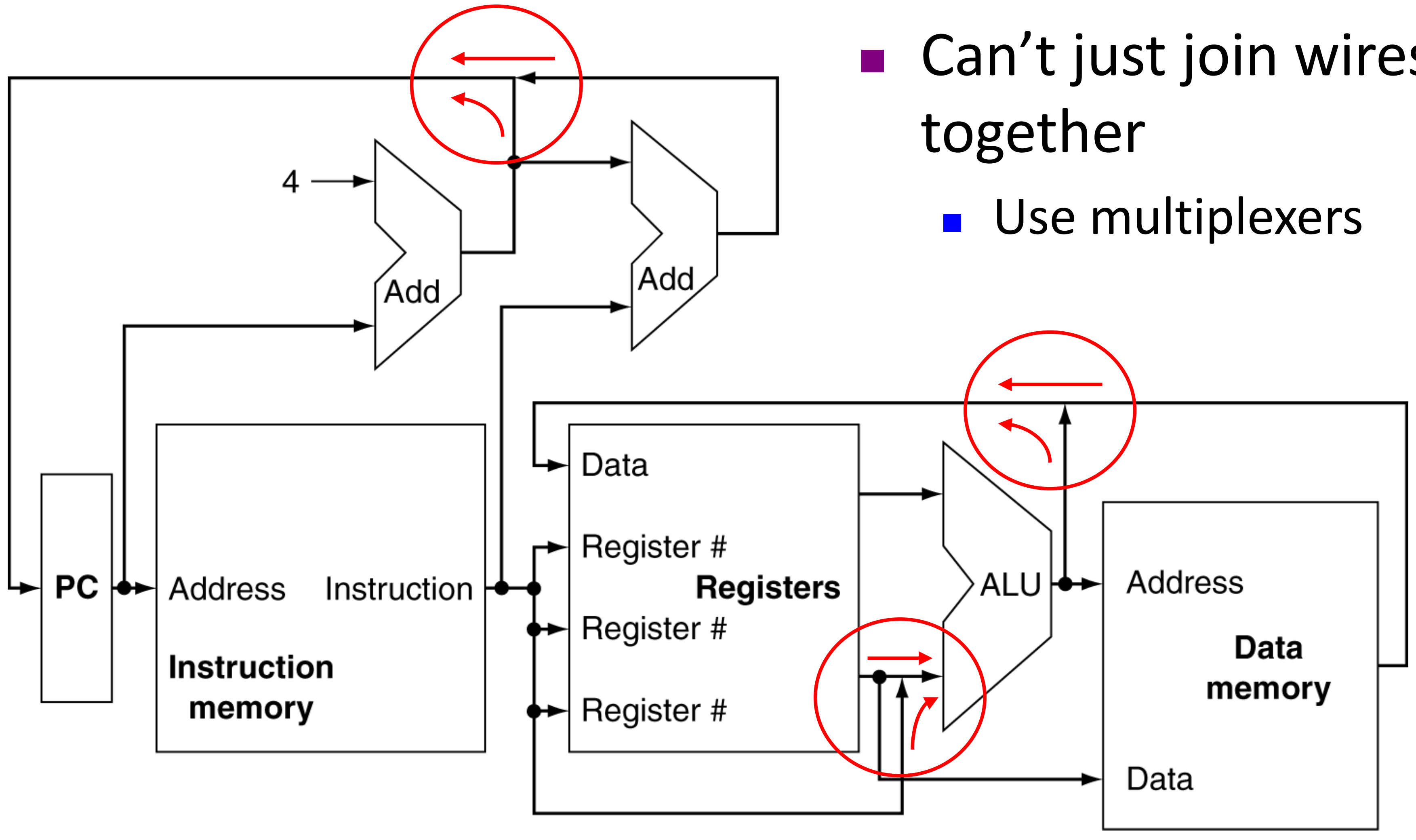
- Stage 5: Register Write
  - Most instructions write the result of some computation into a register
    - Arithmetic, logical, shifts, loads, slt
  - For stores, branches, jumps:
    - Don’t write anything into a register at the end
    - They remain idle during this fifth stage or skip it all together



CPU Components and Major Datapath



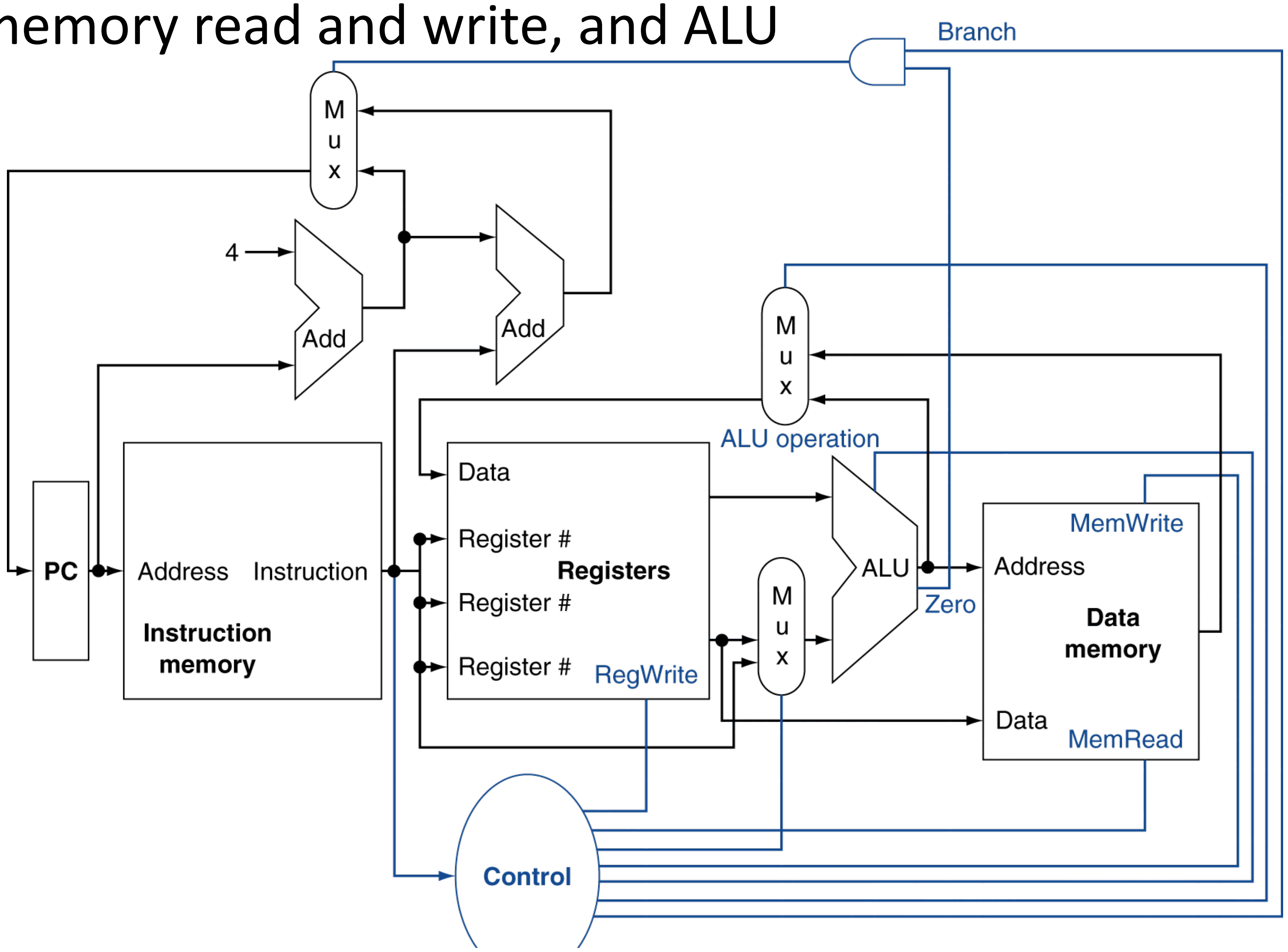
Multiplexers for Selecting Data



- Can’t just join wires together
  - Use multiplexers

Control Signals

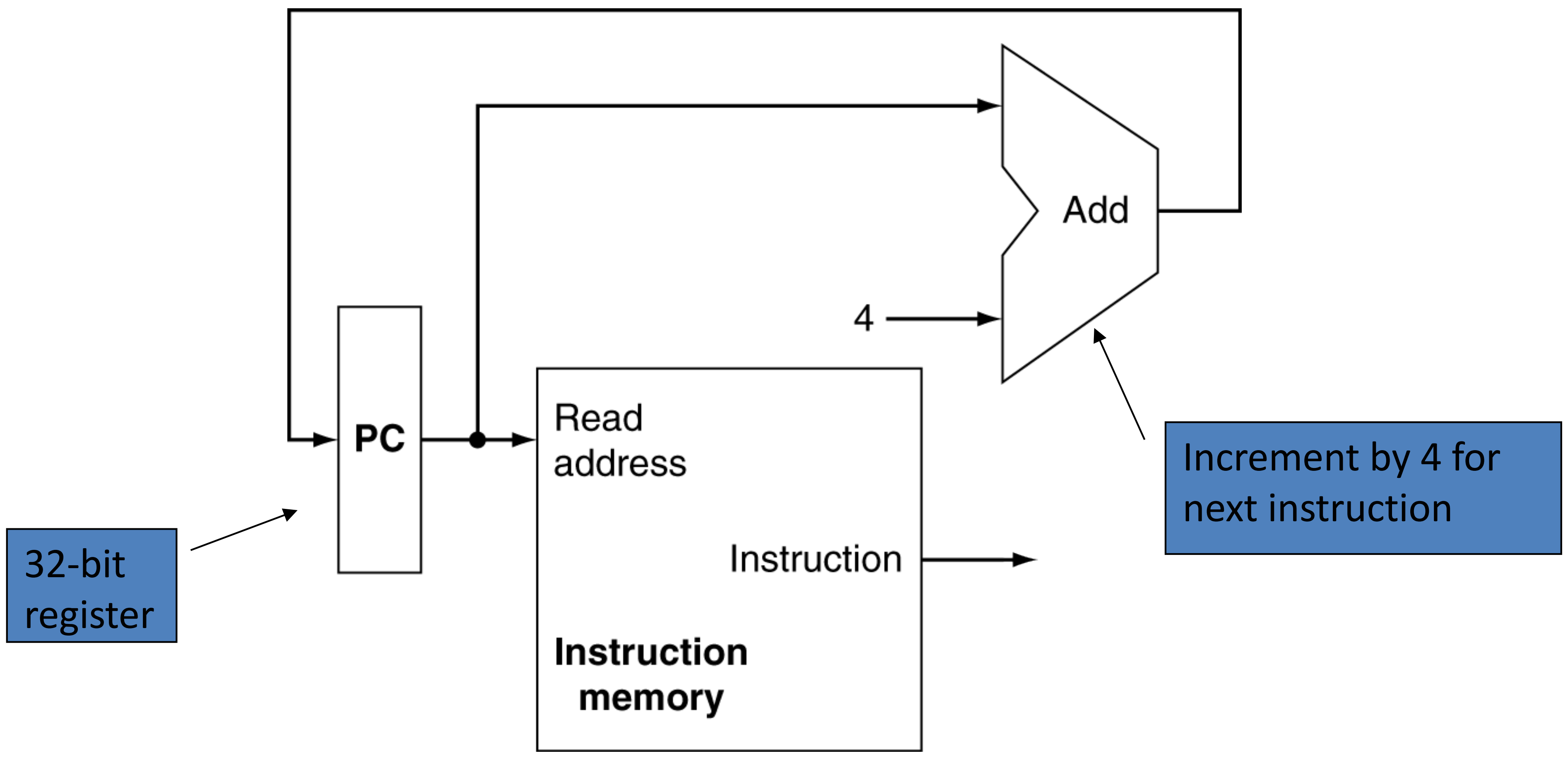
- For selecting input in Mux, and enabling register write, memory read and write, and ALU



Building Datapath

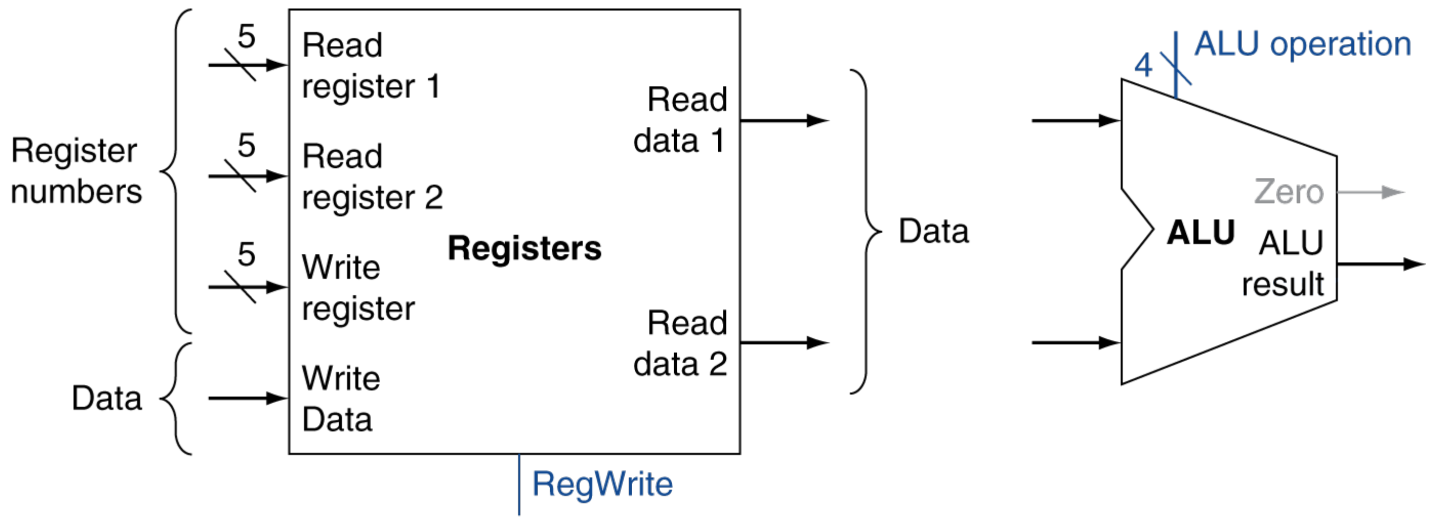
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux’s, memories, …
- To build a RISCV datapath incrementally
  - Refining the overview design

Datapath for Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

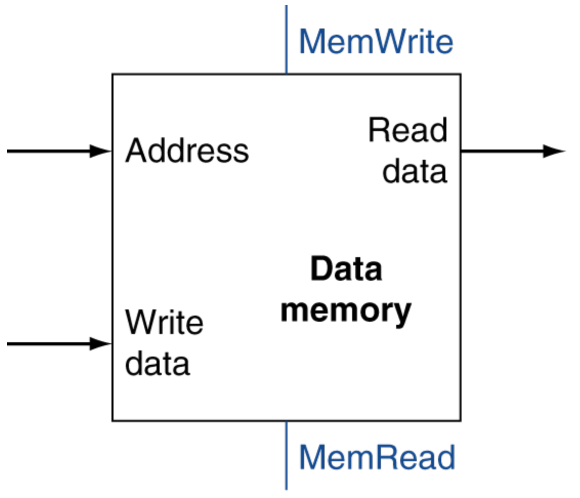


a. Registers

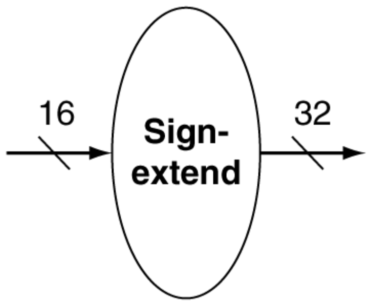
b. ALU

Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

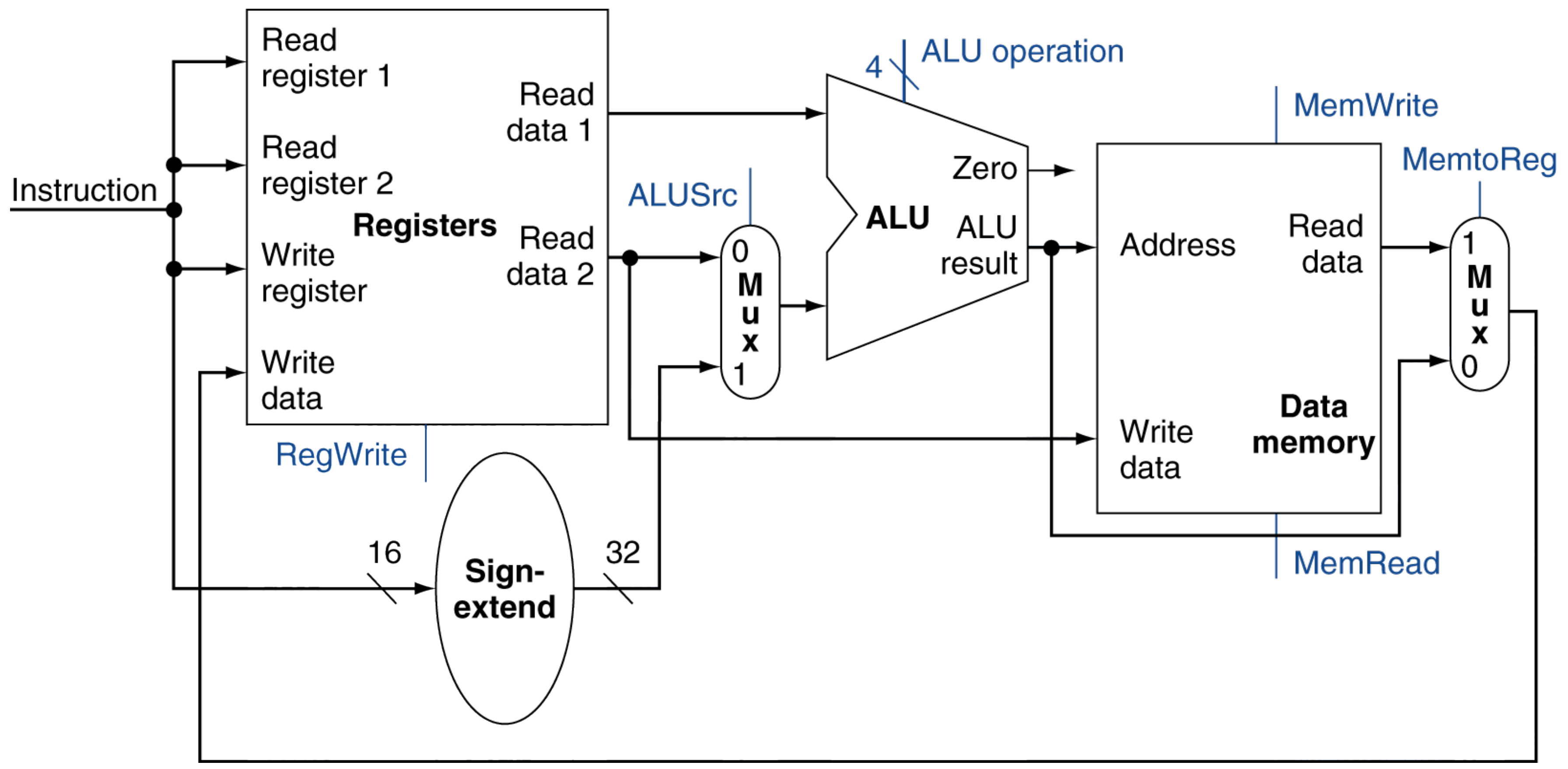


a. Data memory unit



b. Sign extension unit

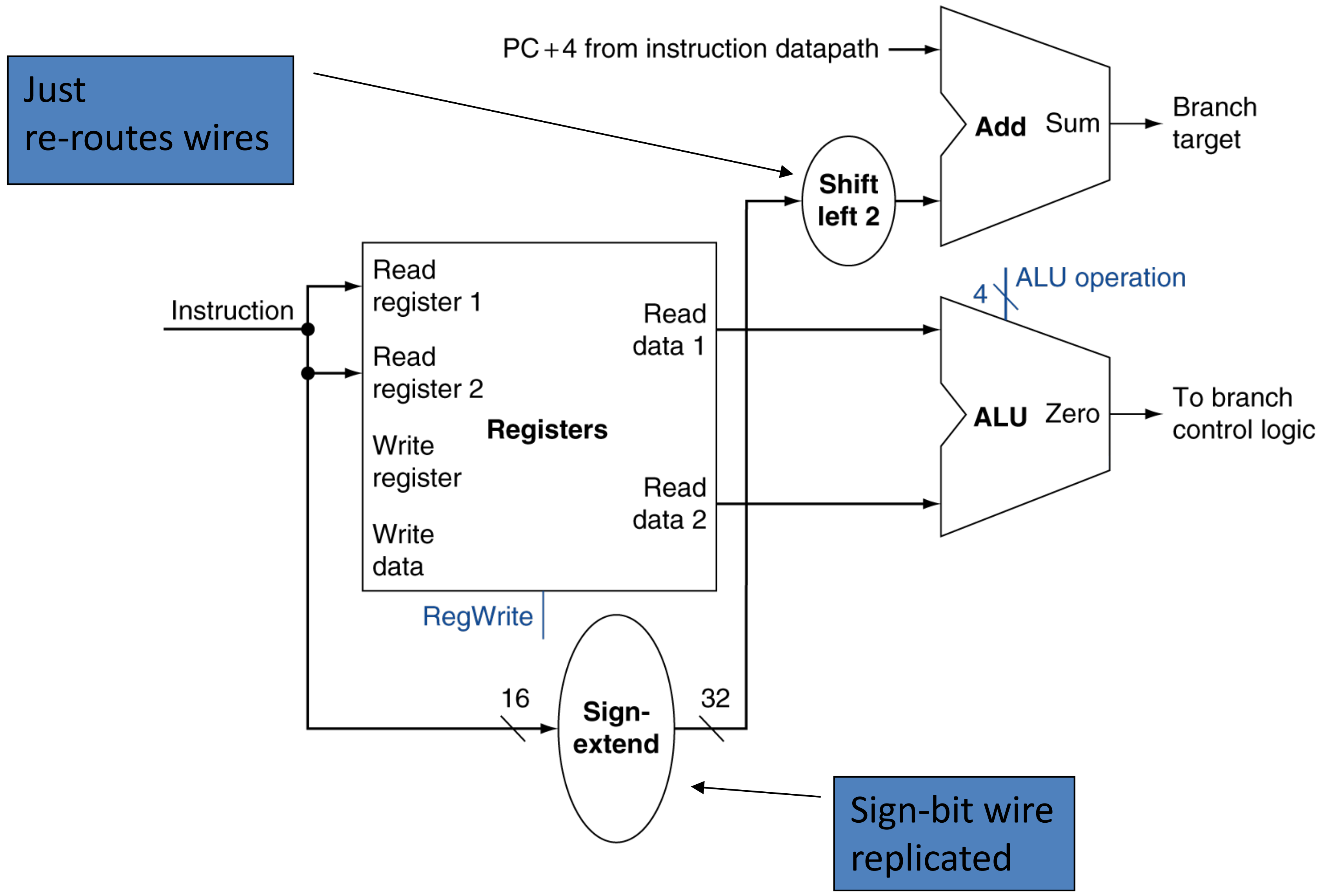
Datapath for R|I-Type, Load and Store



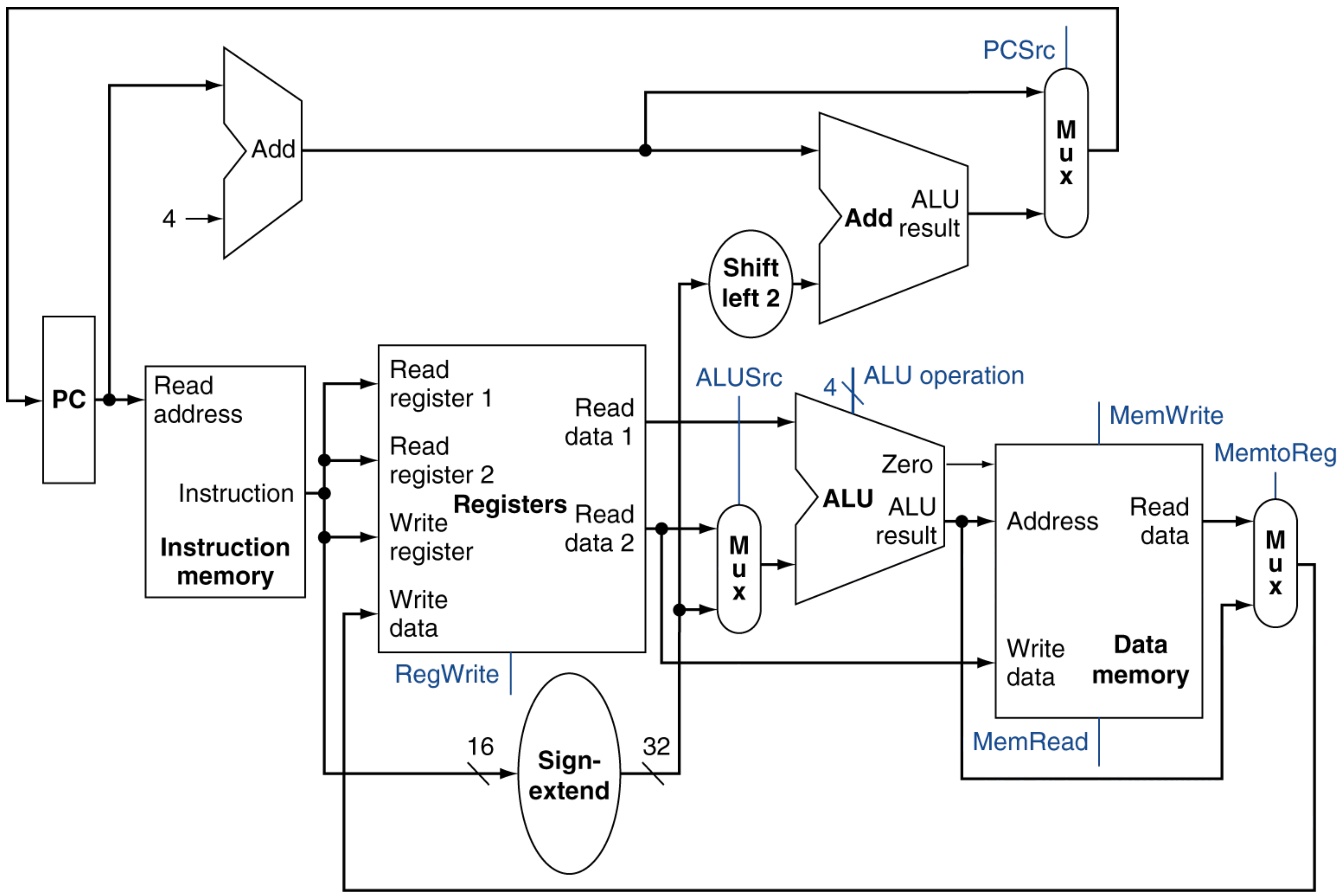
Branch Instructions

- Read register operands
- **Compare operands: To Take or Not**
  - Use ALU, subtract and check Zero output
- **Calculate target address: Where branch to**
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Datapath for Branch Instructions

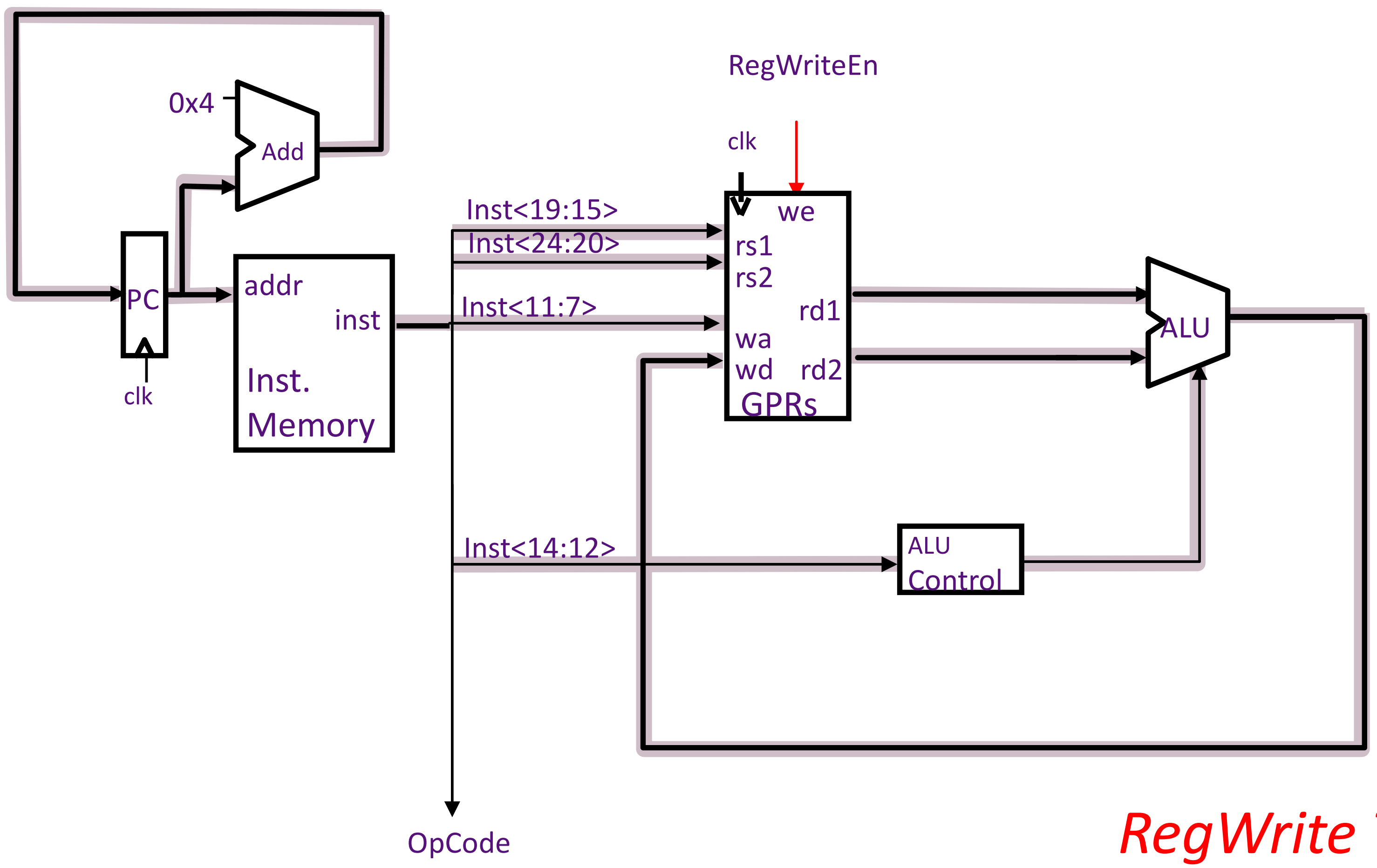


Full Datapath



In More Details using RISC-V

Datapath: Reg-Reg ALU Instructions

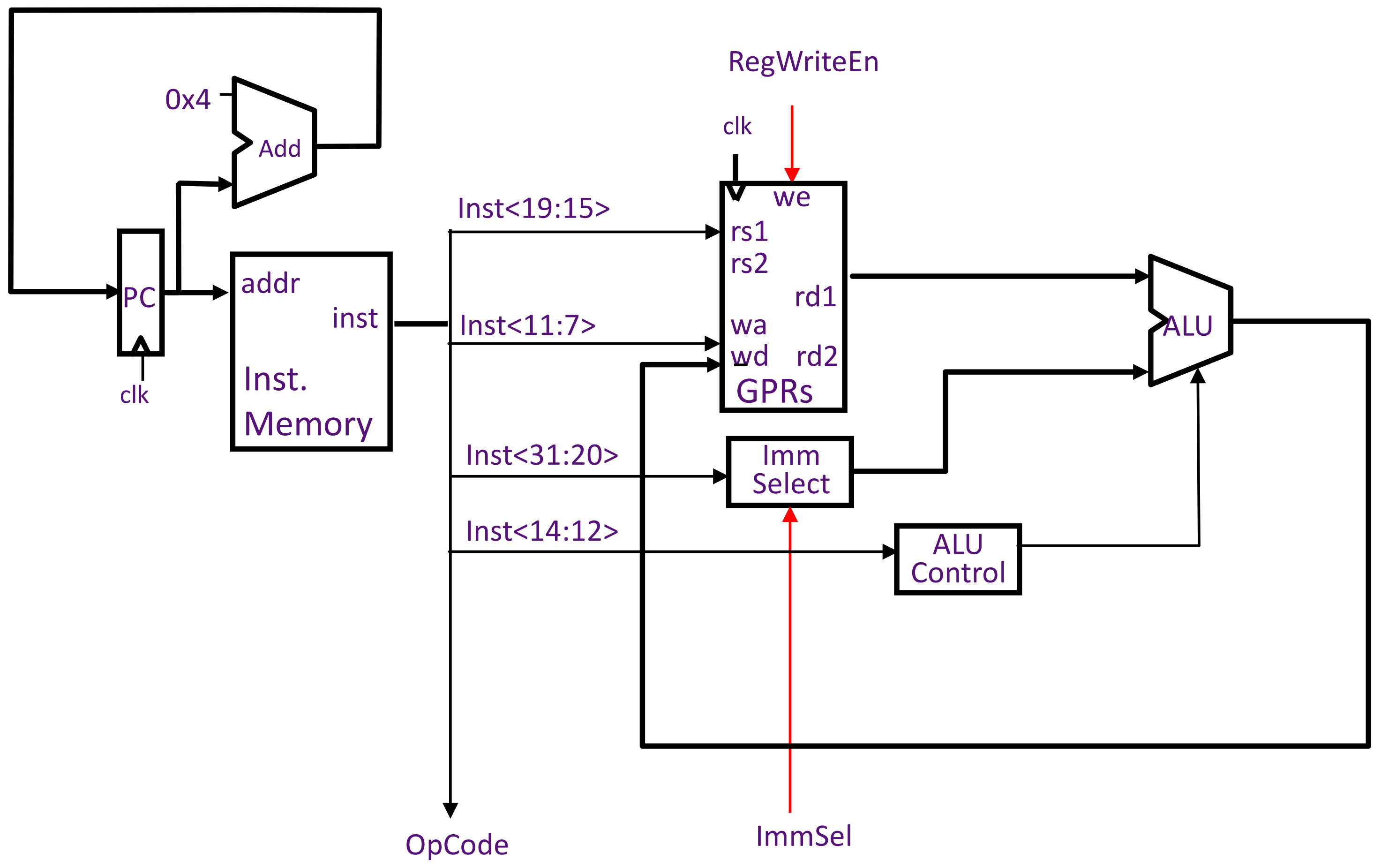


RegWrite Timing?

7	5	5	3	5	7
func7	rs2	rs1	func3	rd	opcode
31 25	24 20	19 15	14 12	11 7	6 0

rd ← (rs1) func (rs2)

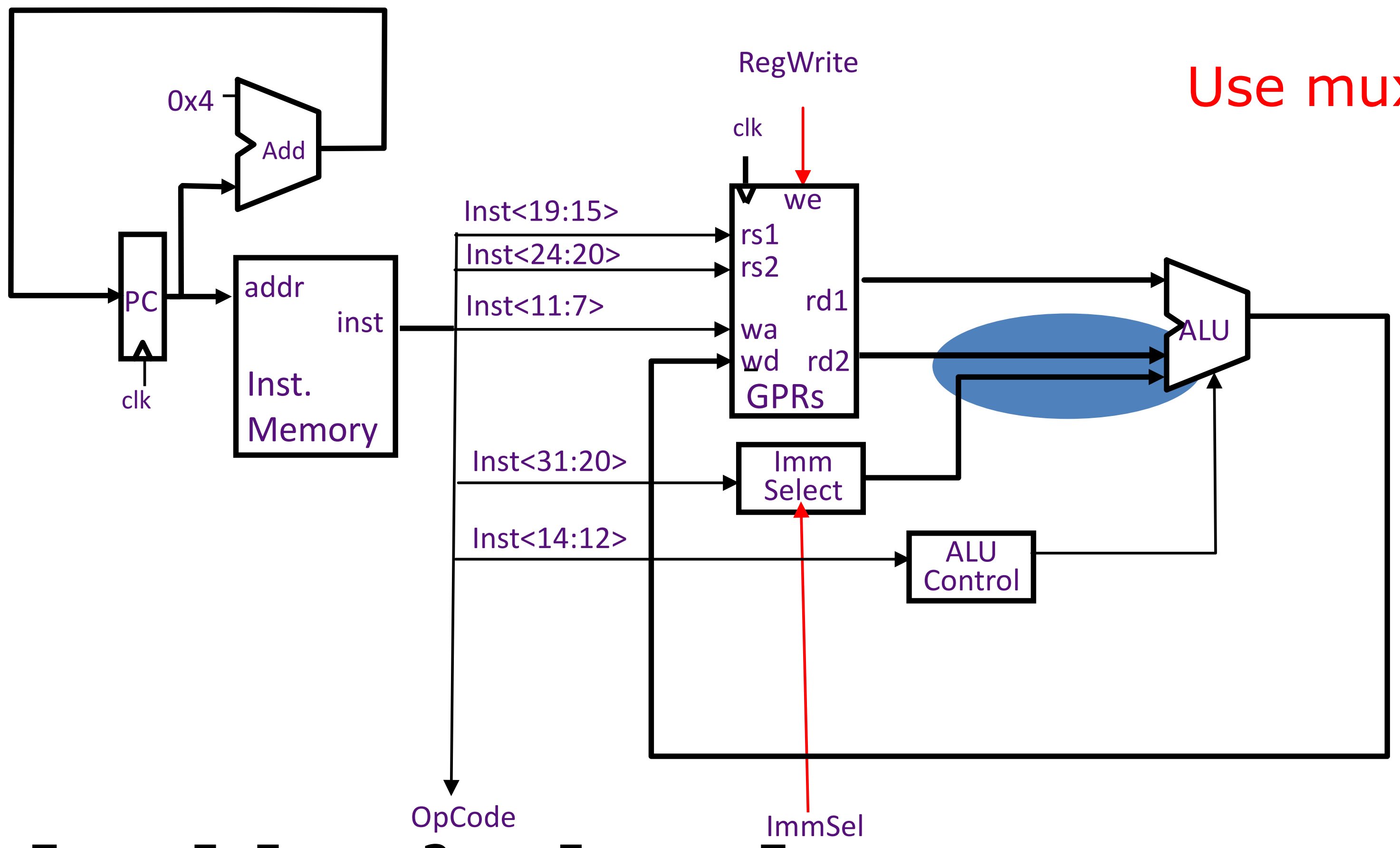
Datapath: Reg-Imm ALU Instructions



12	5	3	5	7
immediate12	rs1	func3	rd	opcode
31 20	19 15	14 12	11 7	6 0

rd ← (rs1) op immediate

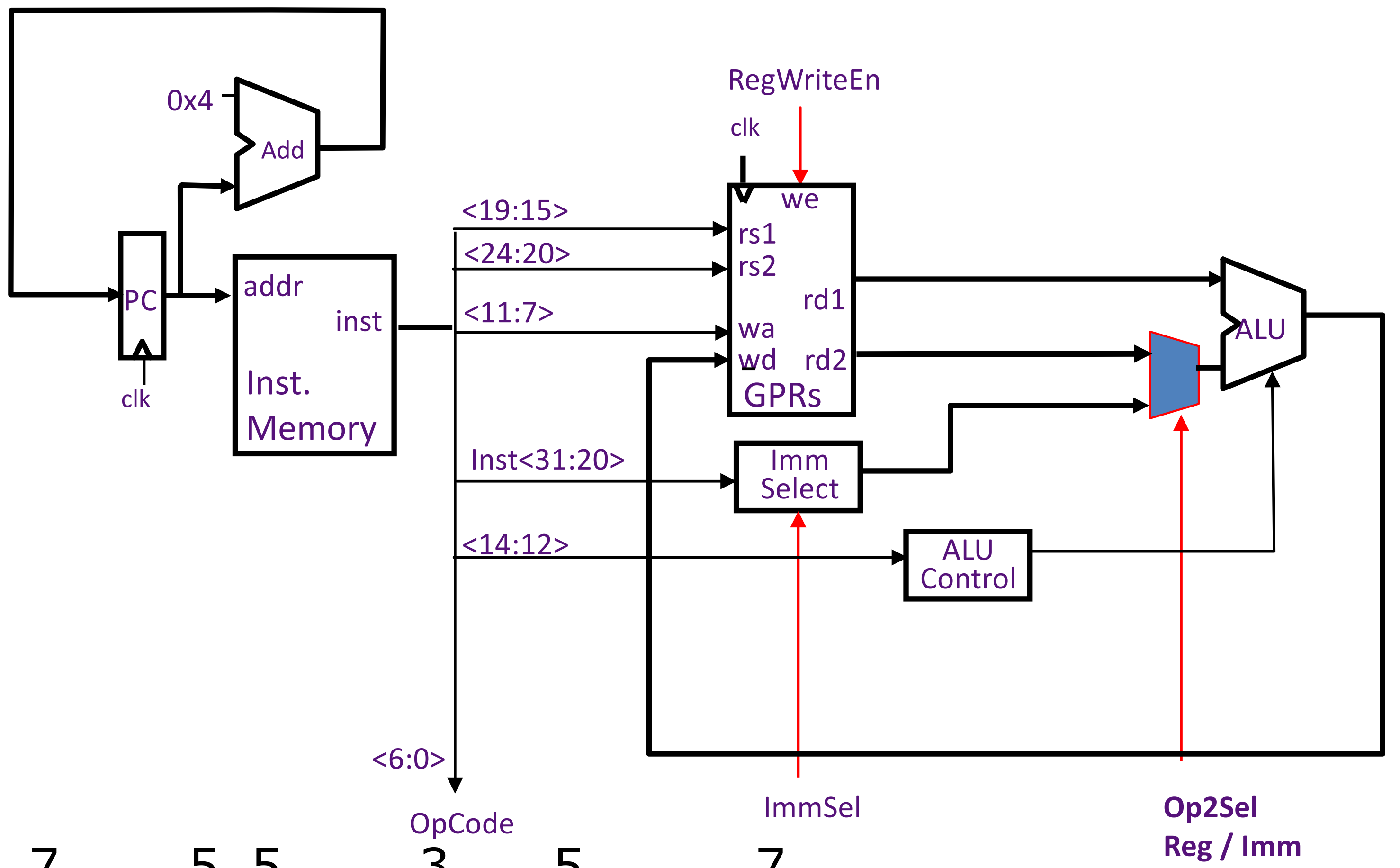
Conflicts in Merging Datapath



Use muxes

7	5	5	3	5	7	
func7	rs2	rs1	func3	rd	opcode	rd ← (rs1) func (rs2)
immediate12		rs1	func3	rd	opcode	rd ← (rs1) op immediate
31	20	19 15	14 12	11 7	6 0	

Datapath for ALU Instructions

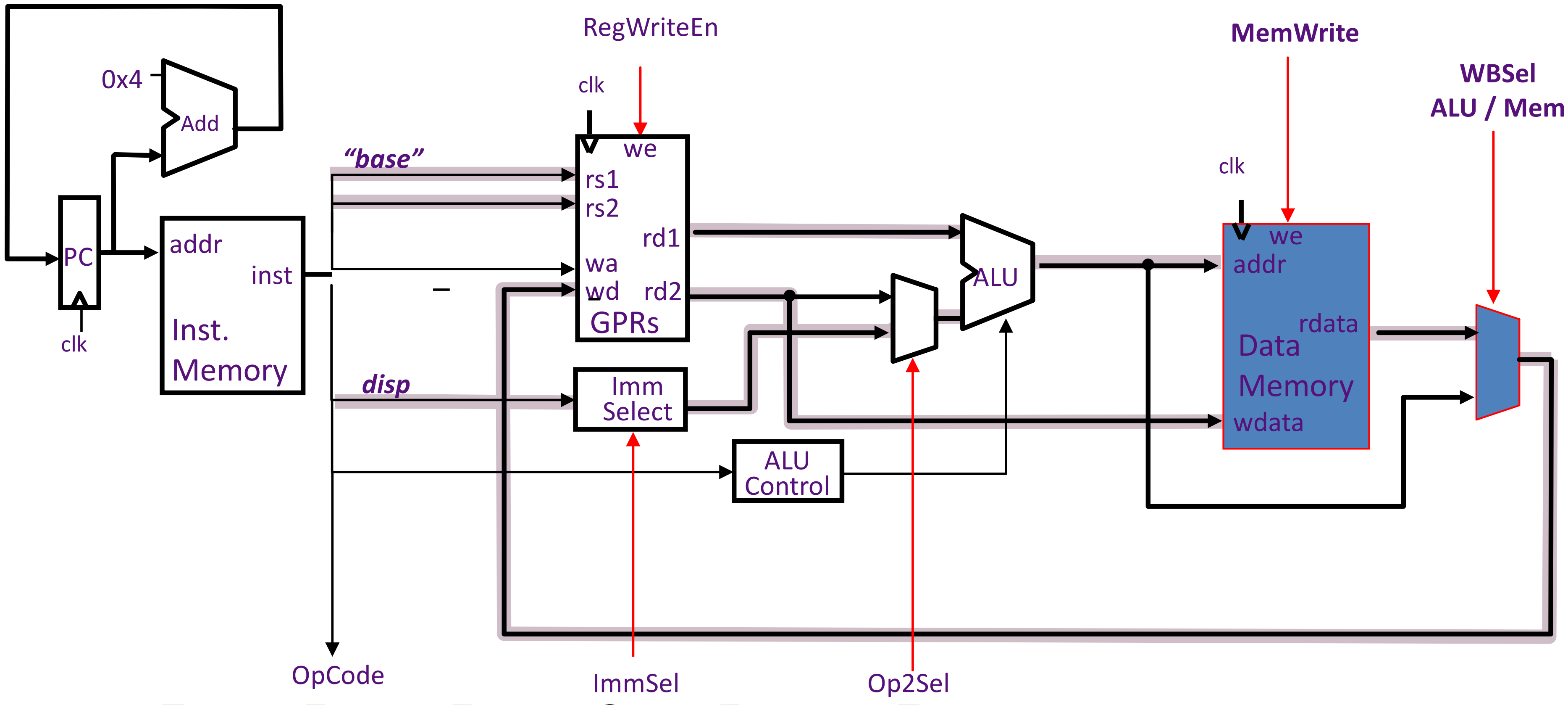


7	5	5	3	5	7
func7	rs2	rs1	func3	rd	opcode
immediate12		rs1	func3	rd	opcode
31	20 19	15 14	12 11	7 6	0

rd ← (rs1) func (rs2)

rd ← (rs1) op immediate

Load/Store Instructions



7	5	5	3	5	7		
imm	rs2	rs1	func3	imm	opcode	Store	(rs1) + displacement
immediate12		rs1	func3	rd	opcode	Load	
31	20 19	15 14	12 11	7 6	0		

rs1 is the base register

rd is the destination of a Load, rs2 is the data source for a Store

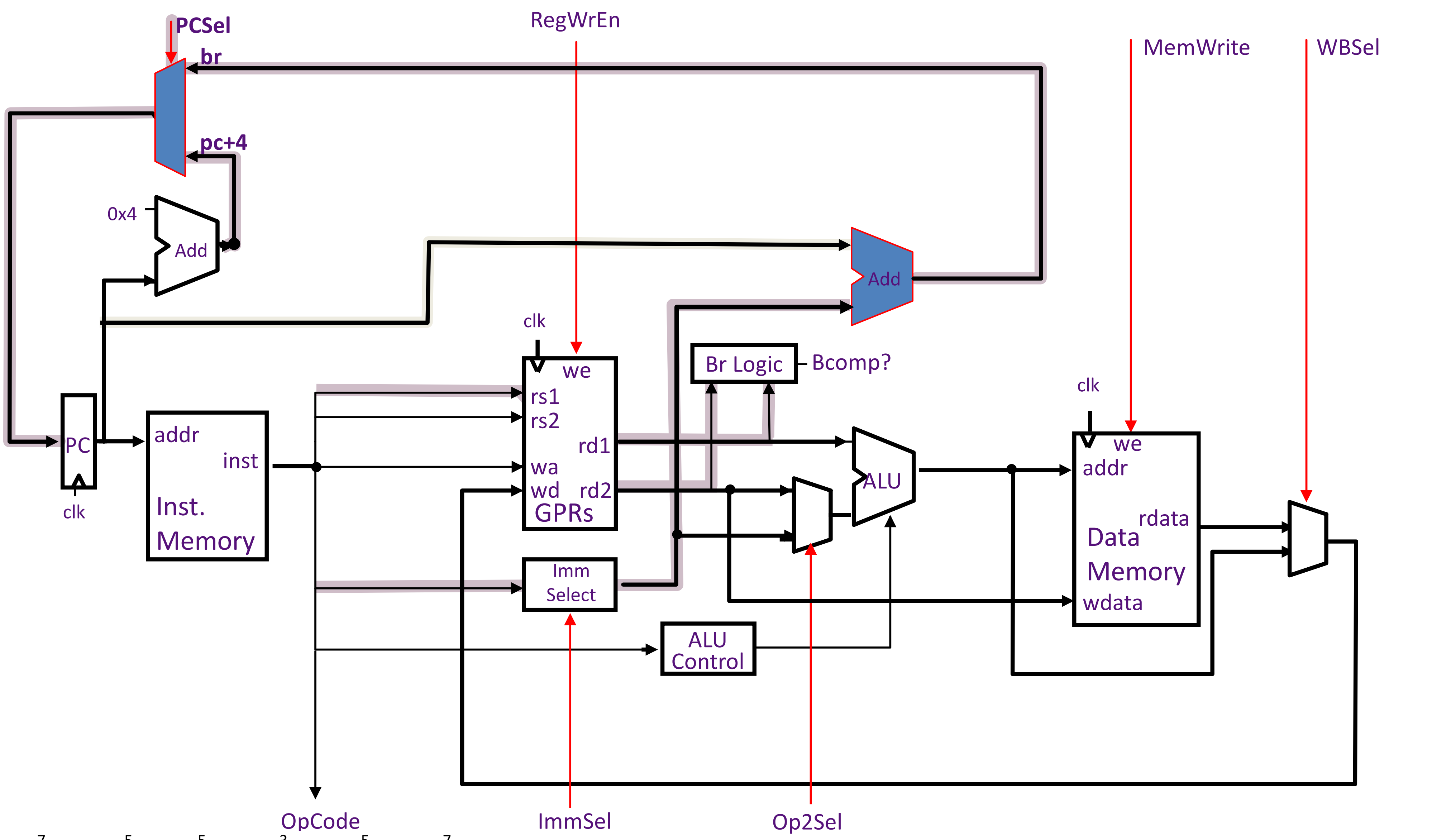
RISC-V Conditional Branches

7	5	5	3	5	7
imm	rs2	rs1	func3	imm	opcode
31 25	24 20	19 15	14 12	11 7	6 0

BEQ/BNE
BLT/BGE
BLTU/BGEU

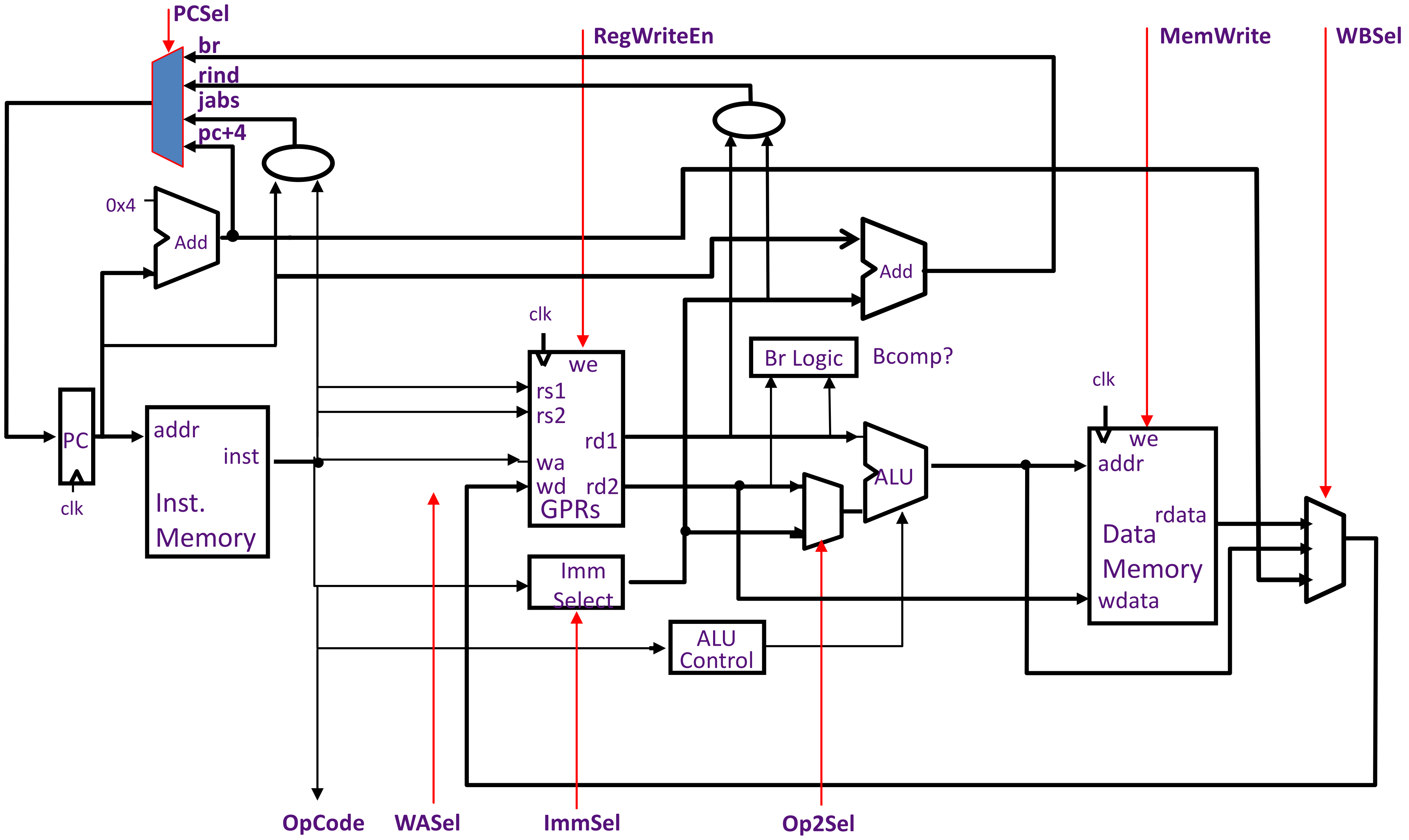
- Compare two integer registers for equality (BEQ/BNE) or signed magnitude (BLT/BGE) or unsigned magnitude (BLTU/BGEU)
- 12-bit immediate encodes branch target address as a signed offset from PC, in units of 16-bits (i.e., shift left by 1 then add to PC).

Conditional Branches
(BEQ/BNE/BLT/BGE/BLTU/BGEU)

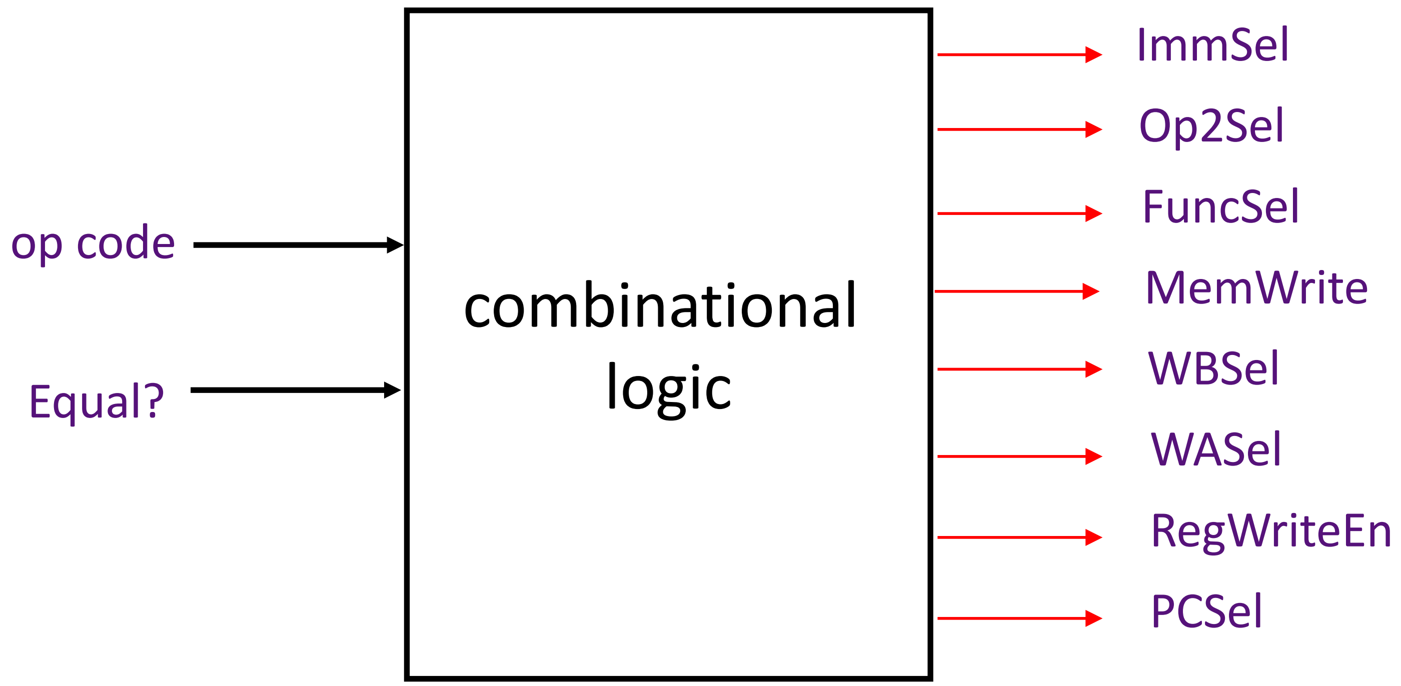


7	5	5	3	5	7
imm	rs2	rs1	func3	imm	opcode
31 25	24 20	19 15	14 12	11 7	6 0

Full Datapath for Unpipelined RISC-V



Hardwired Control Combinational Logic



Hardwired Control Table

Opcode	ImmSel	Op2Sel	FuncSel	MemWr	RFWen	WBSel	WASel	PCSel
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	IType_{12}	Imm	Op	no	yes	ALU	rd	pc+4
LW	IType_{12}	Imm	+	no	yes	Mem	rd	pc+4
SW	SType_{12}	Imm	+	yes	no	*	*	pc+4
BEQ_{true}	SBType_{12}	*	*	no	no	*	*	br
$\text{BEQ}_{\text{false}}$	SBType_{12}	*	*	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	X1	jabs
JALR	*	*	*	no	yes	PC	rd	rind

Op2Sel= Reg / Imm **WBSel = ALU / Mem / PC**

WASel = rd / X1 **PCSel = pc+4 / br / rind / jabs**

ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Implementation in Real

- Load-Store RISC ISAs designed for efficient pipelined implementations
  - Inspired by earlier Cray machines (CDC 6600/7600)
- RISC-V ISA implemented using Chisel hardware construction language
  - Chisel: https://chisel.eecs.berkeley.edu/
  - Getting started:
    - https://chisel.eecs.berkeley.edu/2.2.0/getting-started.html
  - Check resource page for slides and other info

Chisel in one slides

- Module
- IO
- Wire
- Reg
- Mem

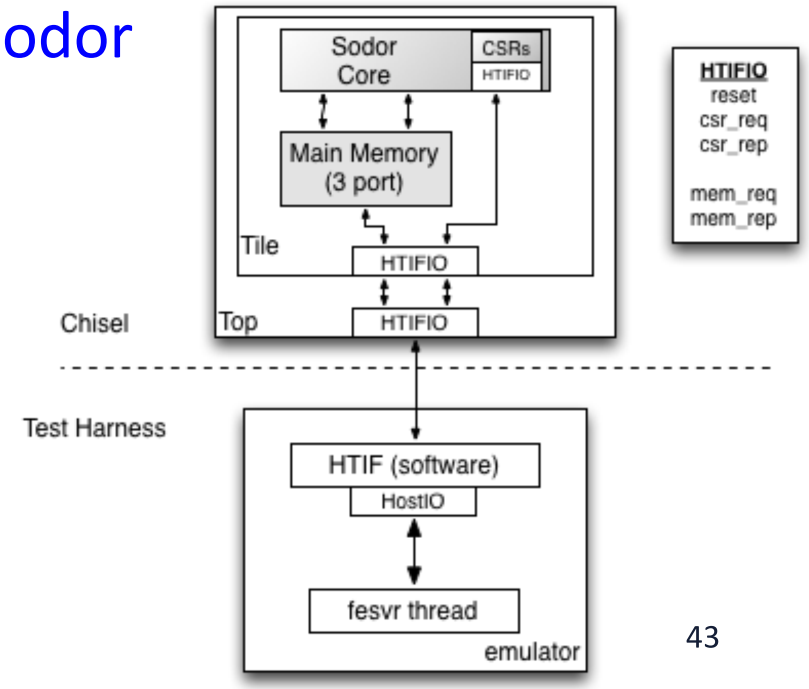
```
import Chisel._

class GCD extends Module {
  val io = new Bundle {
    val a  = UInt(INPUT,  16)
    val b  = UInt(INPUT,  16)
    val e  = Bool(INPUT)
    val z  = UInt(OUTPUT, 16)
    val v  = Bool(OUTPUT)
  }
  val x  = Reg(UInt())
  val y  = Reg(UInt())
  when   (x > y) { x := x - y }
  unless (x > y) { y := y - x }
  when (io.e) { x := io.a; y := io.b }
  io.z := x
  io.v := y === UInt(0)
} ...
```

UCB RISC-V Sodor

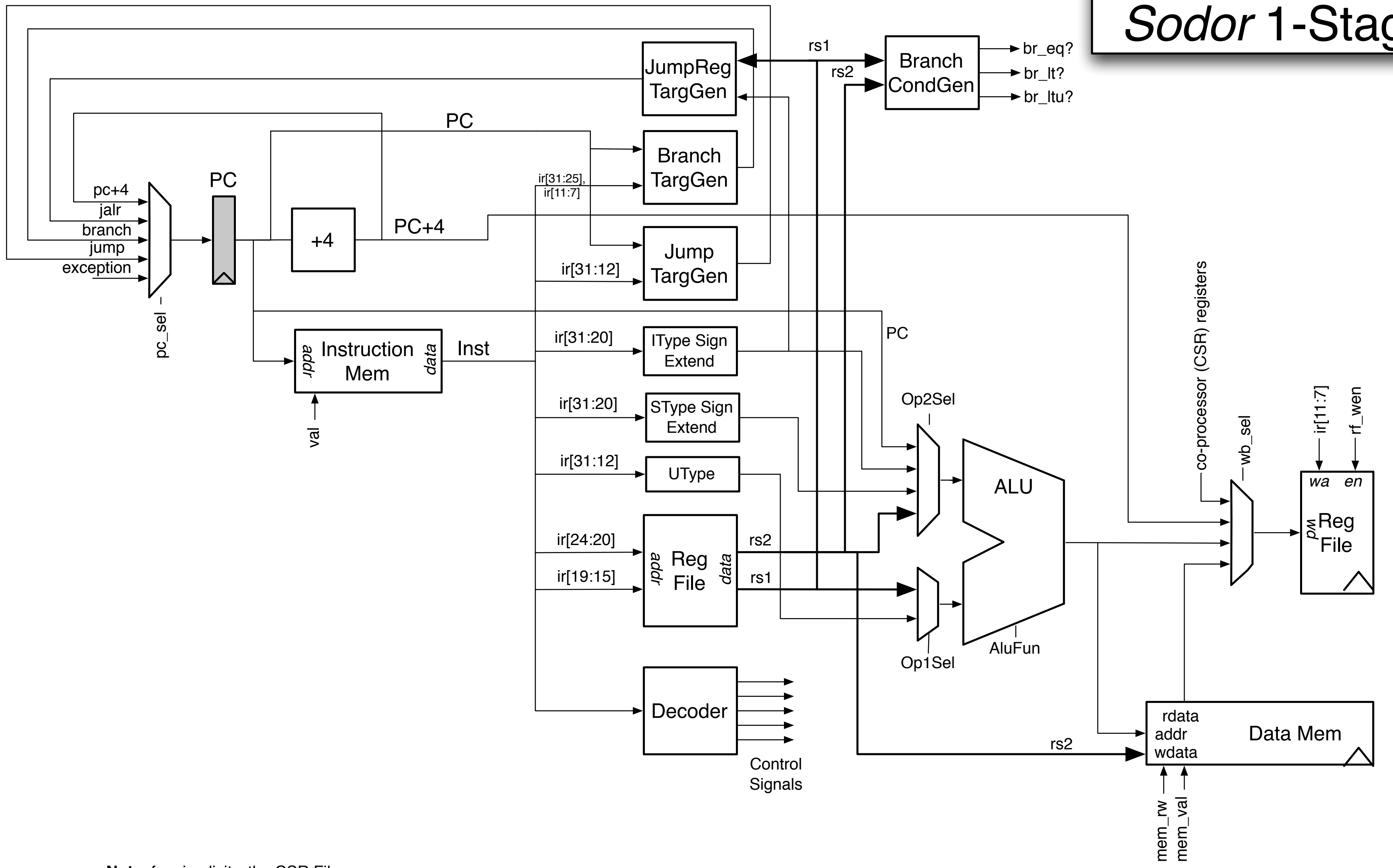
- https://github.com/ucb-bar/riscv-sodor
  - Single-cycle:
    - https://github.com/ucb-bar/riscv-sodor/tree/master/src/rv32_1stage

- Assignment 2 uses an older version
  - https://github.com/passlab/riscv-sodor



Full RISCV1Stage Datapath

RISC-V
Sodor 1-Stage



Note: for simplicity, the CSR File (control and status registers) and associated datapath is not shown

Execute Stage

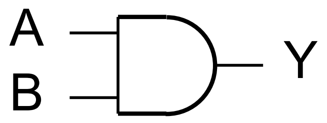
Additional Materials

Logic Design Basics

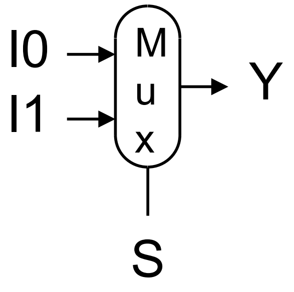
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational circuit
  - Operate on data
  - Output is a function of input
- State (sequential) circuit
  - Store information

Combinational Circuits

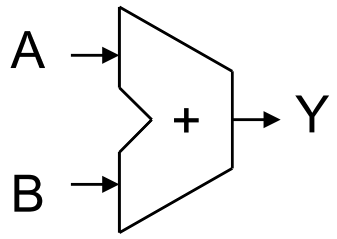
- AND-gate
  - Y = A & B



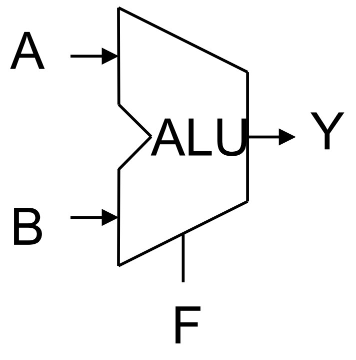
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

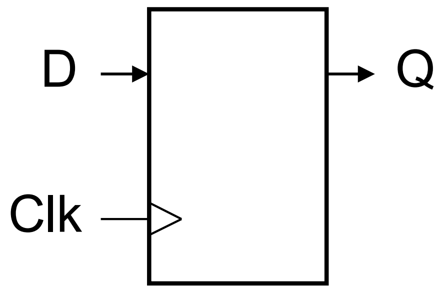


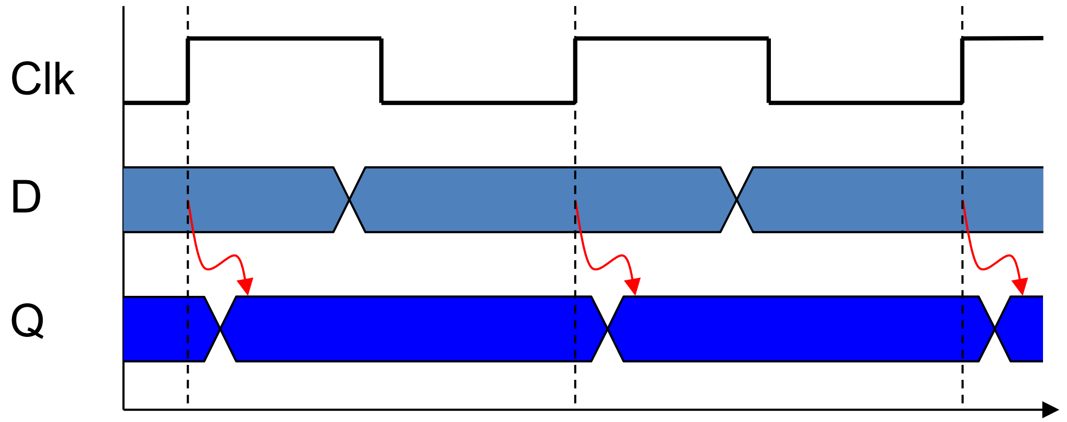
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Circuits

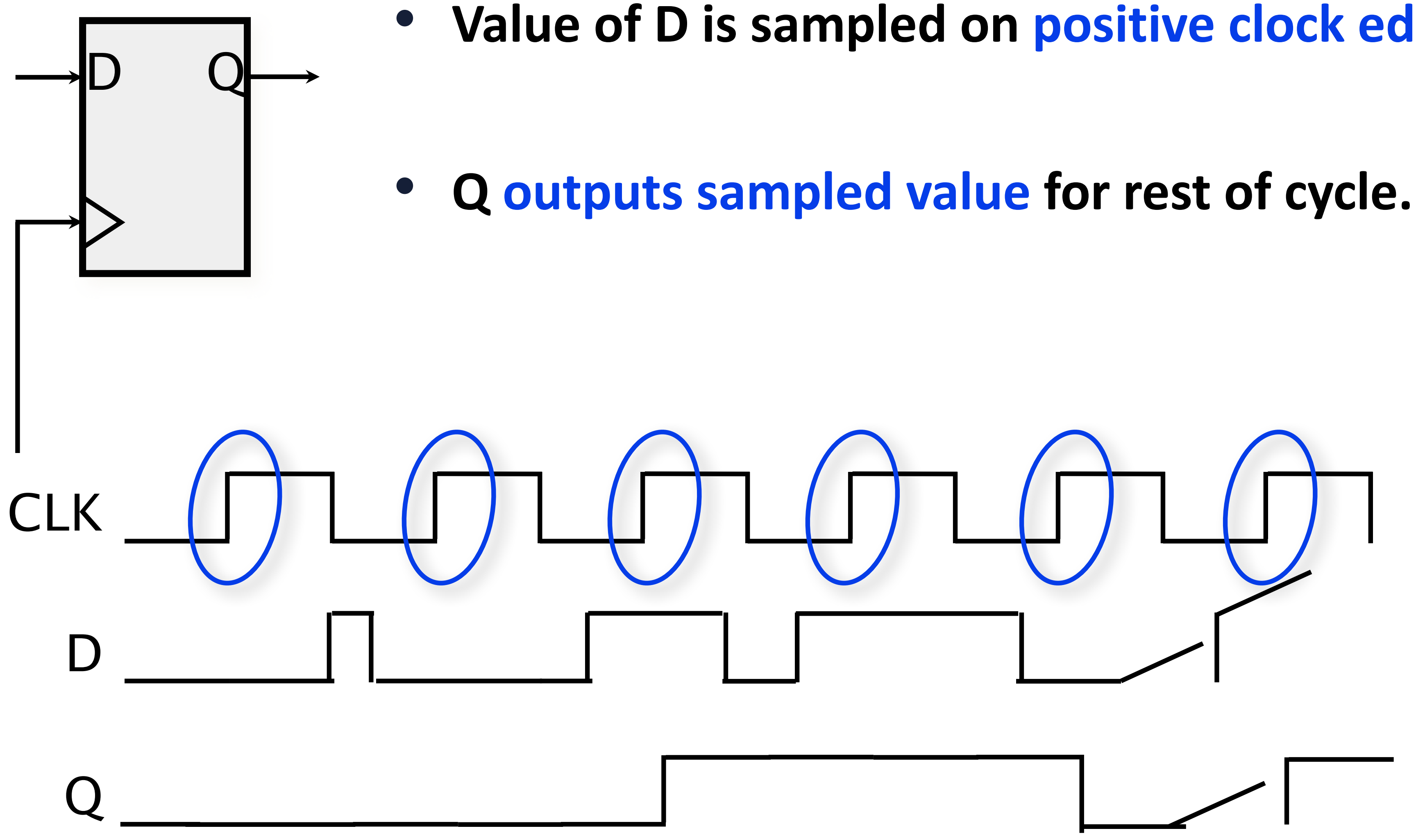
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





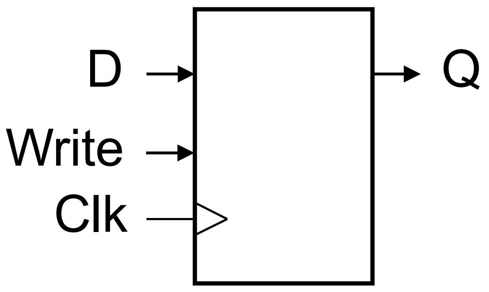
Edge-Triggered D Flip Flops

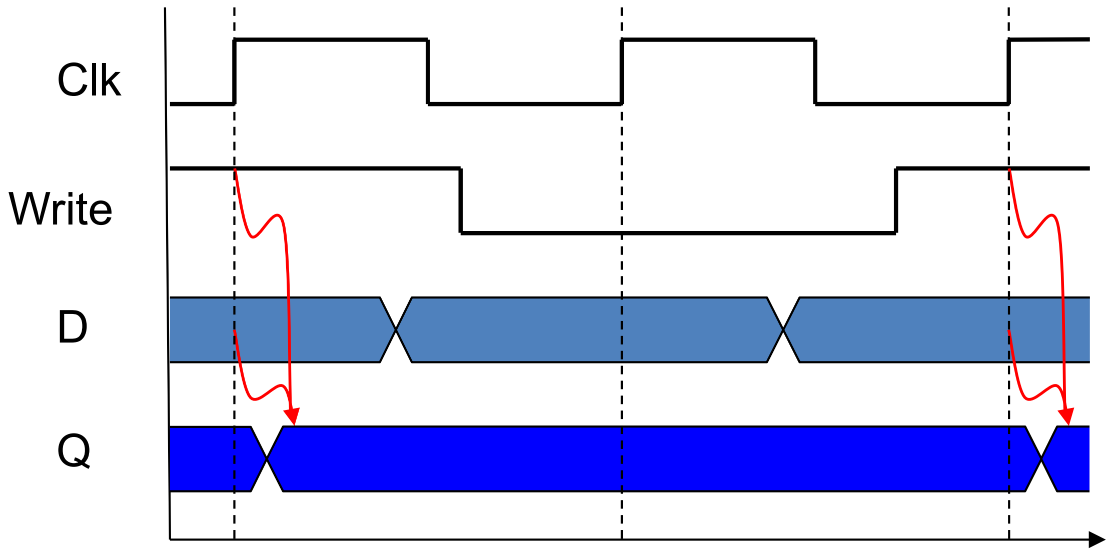
- **Value of D is sampled on positive clock edge.**
- **Q outputs sampled value for rest of cycle.**



Sequential Circuits

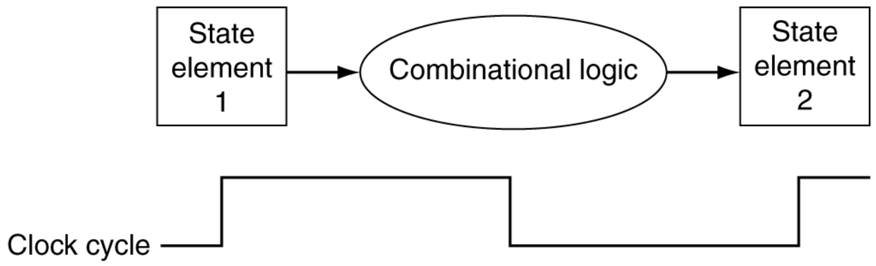
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

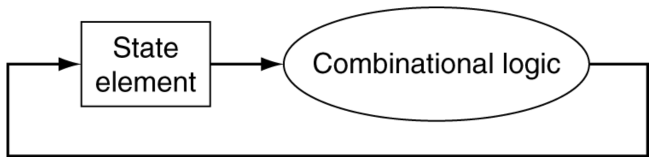




Clocking Methodology

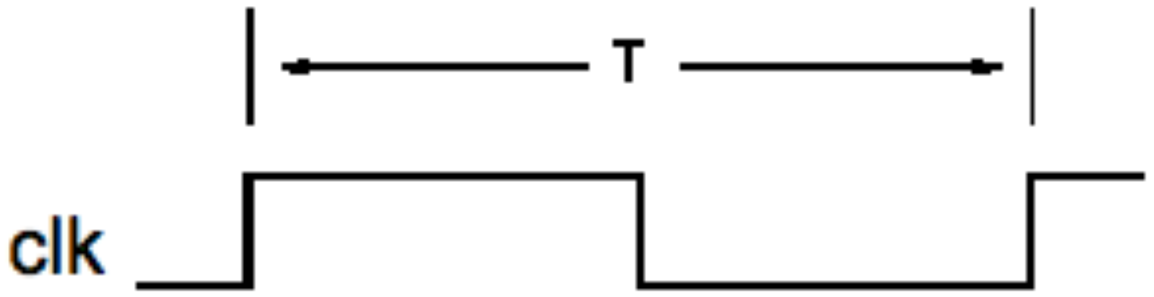
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period





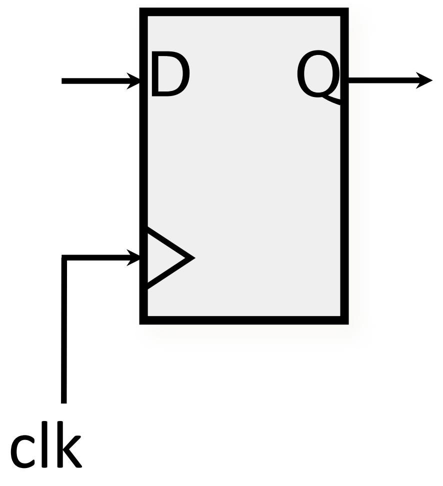
Single cycle data paths

Processor uses synchronous logic design (a “clock”).



f	T
1 MHz	1 µs
10 MHz	100 ns
100 MHz	10 ns
1 GHz	1 ns

All state elements act like positive edge-triggered flip flops.



Reset?

Single-Cycle Hardwired Control

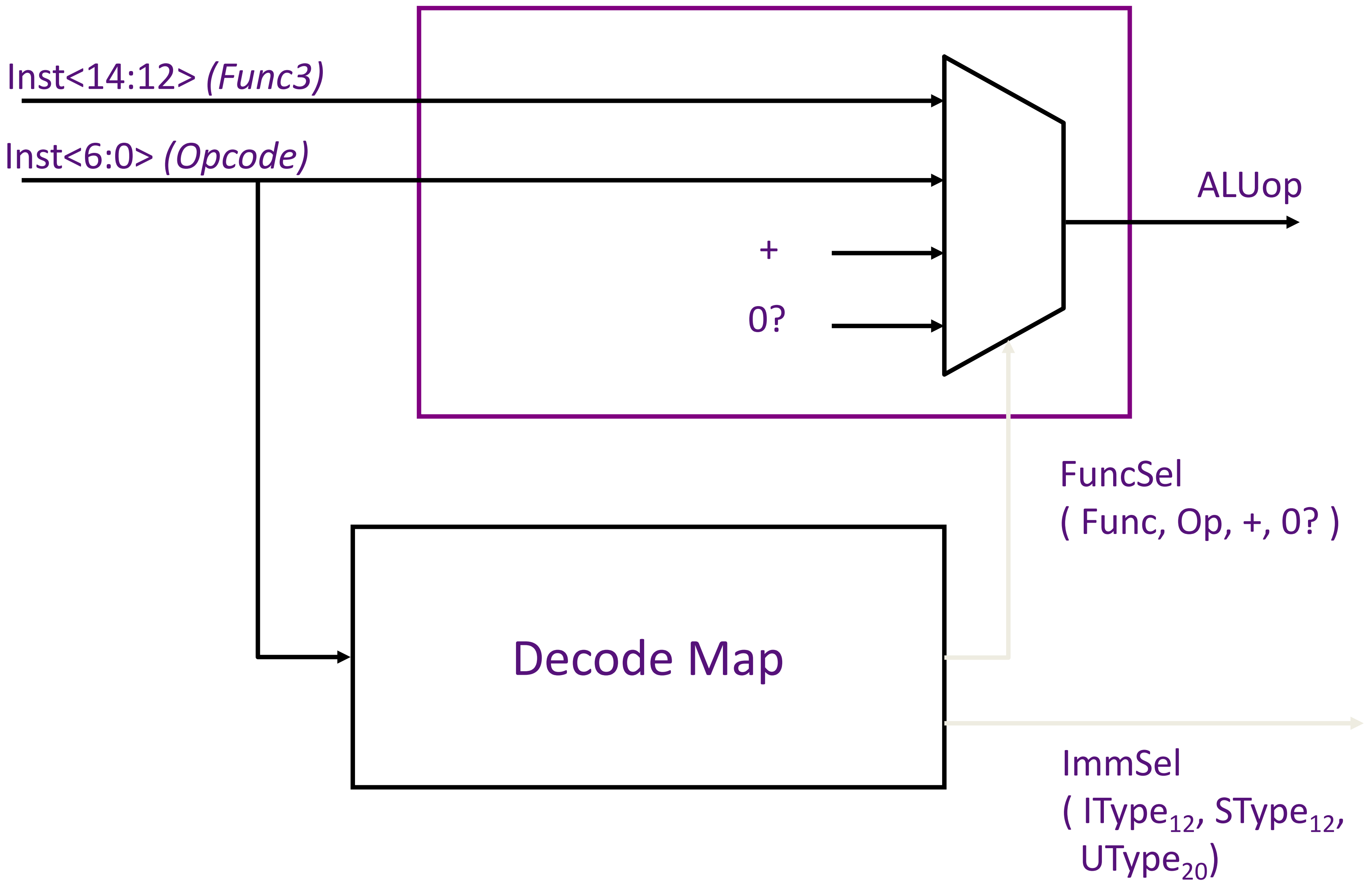
Clock period is sufficiently long for all of the following steps to be “completed”:

1. Instruction fetch
2. Decode and register fetch
3. ALU operation
4. Data fetch if required
5. Register write-back setup time

=> tC > tIFetch + tRFetch + tALU+ tDMem+ tRWB

At the rising edge of the following clock, the PC, register file and memory are updated

ALU Control & Immediate Extension


Inst<14:12> *(Func3)*
Inst<6:0> *(Opcode)*
+
0?
ALUop
FuncSel
(Func, Op, +, 0?)
Decode Map
ImmSel
($IType_{12}$, $SType_{12}$, $UType_{20}$)