KING'S College LONDON

Hardware Design

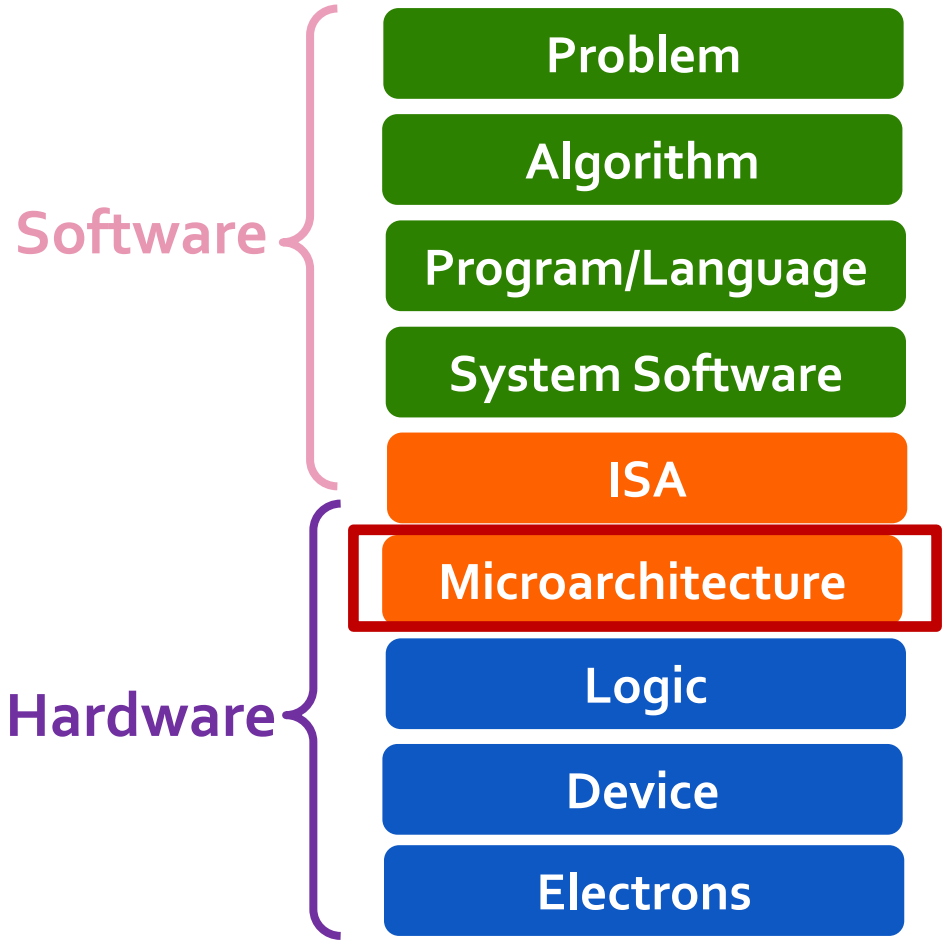
Lecture 5: Multi-Cycle Microarchitecture and Pipelining

Dr. Haiyu Mao

26.02.2026

What We Learned & Will Learn

- The von Neumann model
- Instruction Set Architectures (ISA)
- Assembly programming: LC-3 and MIPS
- Microarchitecture: basics
- Microarchitecture: Single-cycle
- **Microarchitecture: Multi-cycle**
- Pipelining
- Cache and Memory



A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1
- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute
- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

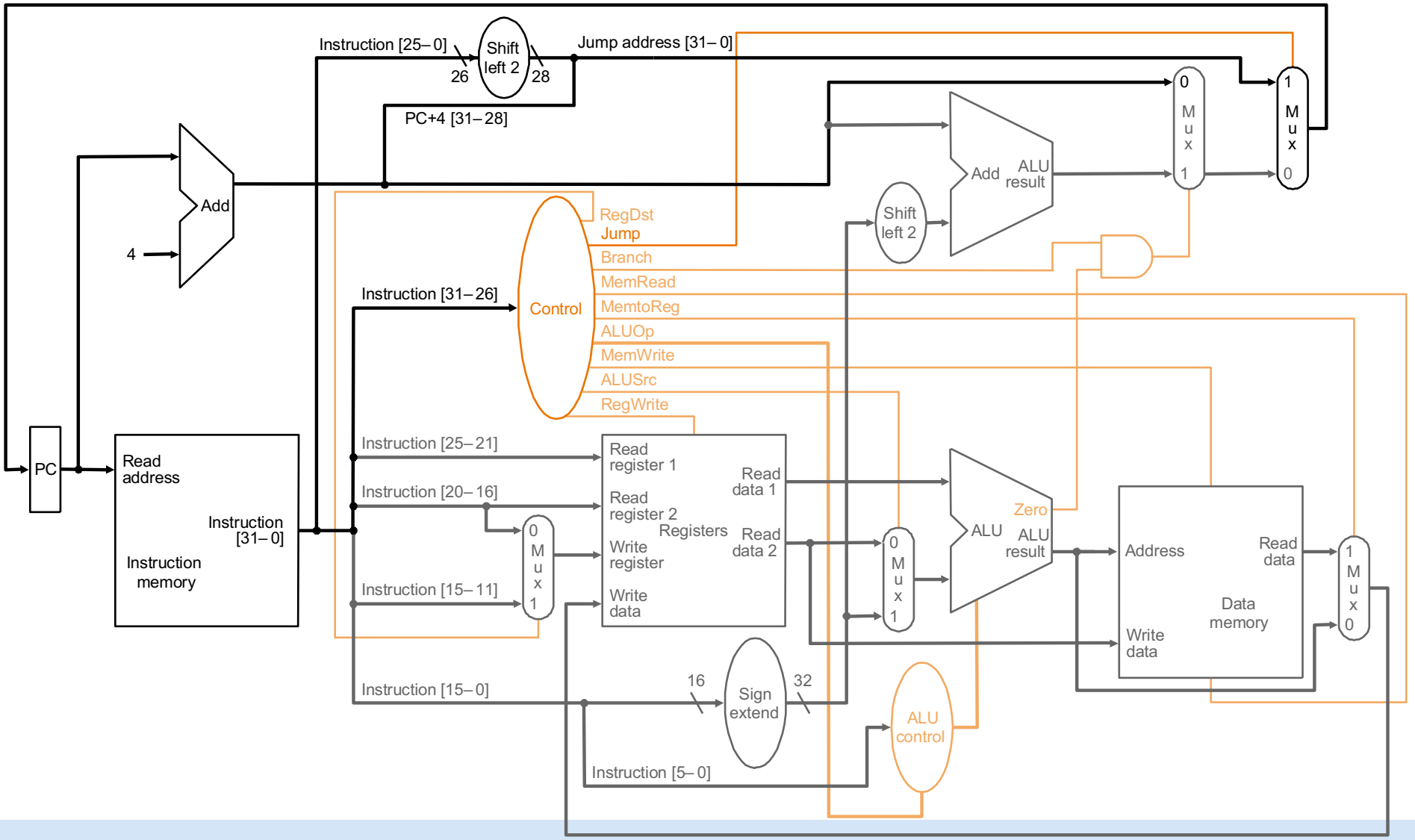
- Let's go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

- Does every instruction take the same time (latency) to complete?

Let's Find the Critical Path

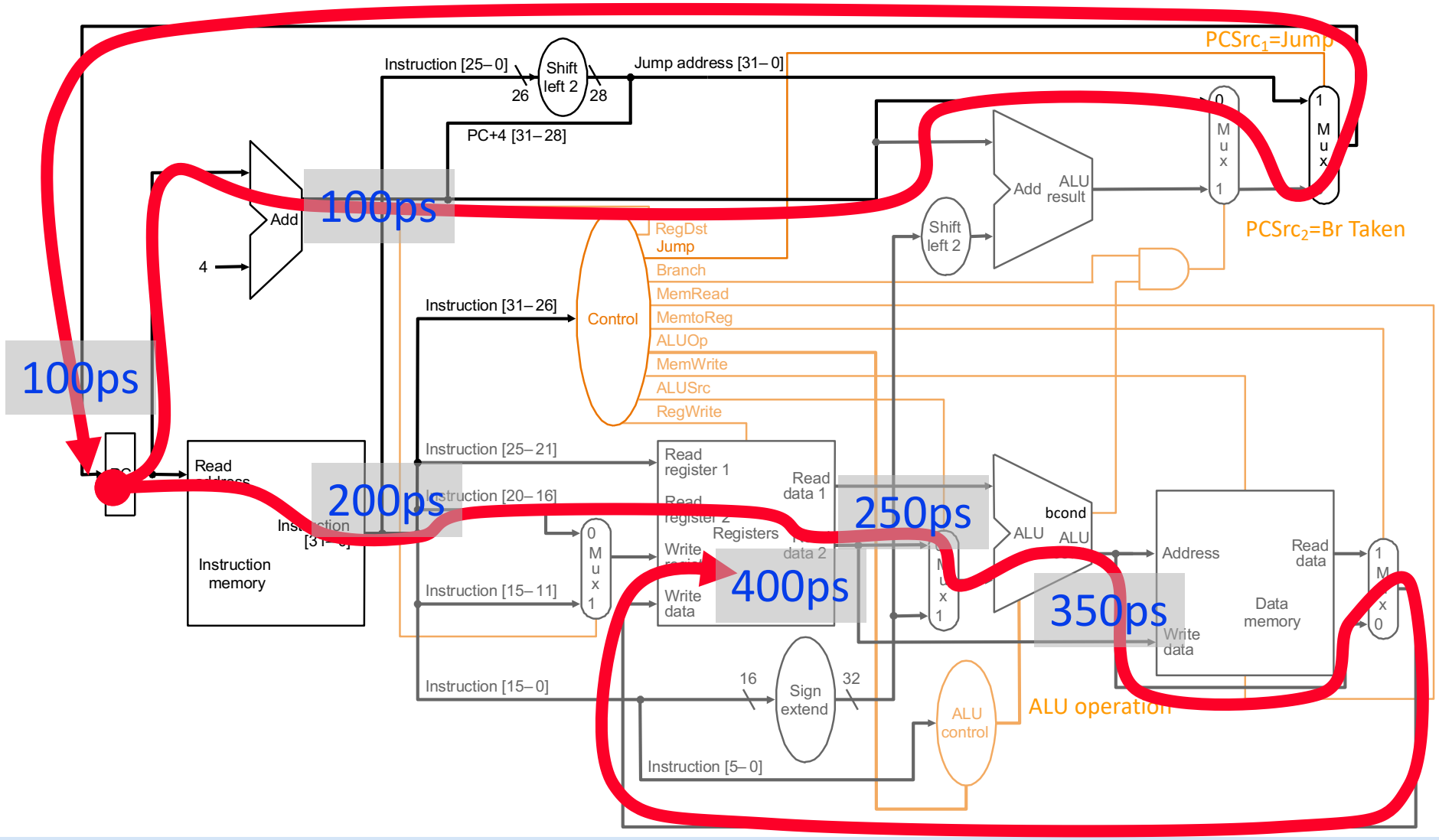


Example Single-Cycle Datapath Analysis

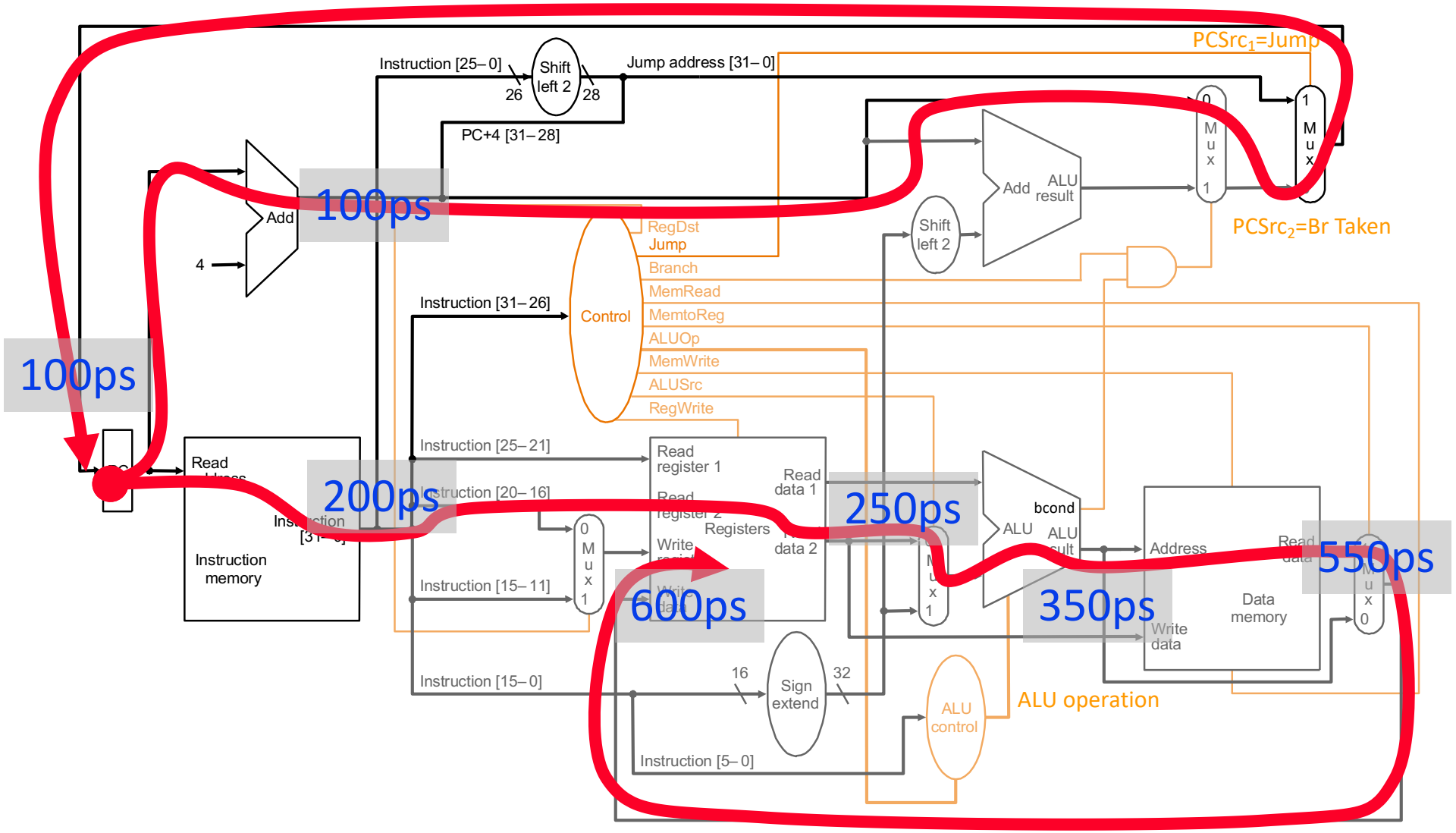
- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other logic or wire delay: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

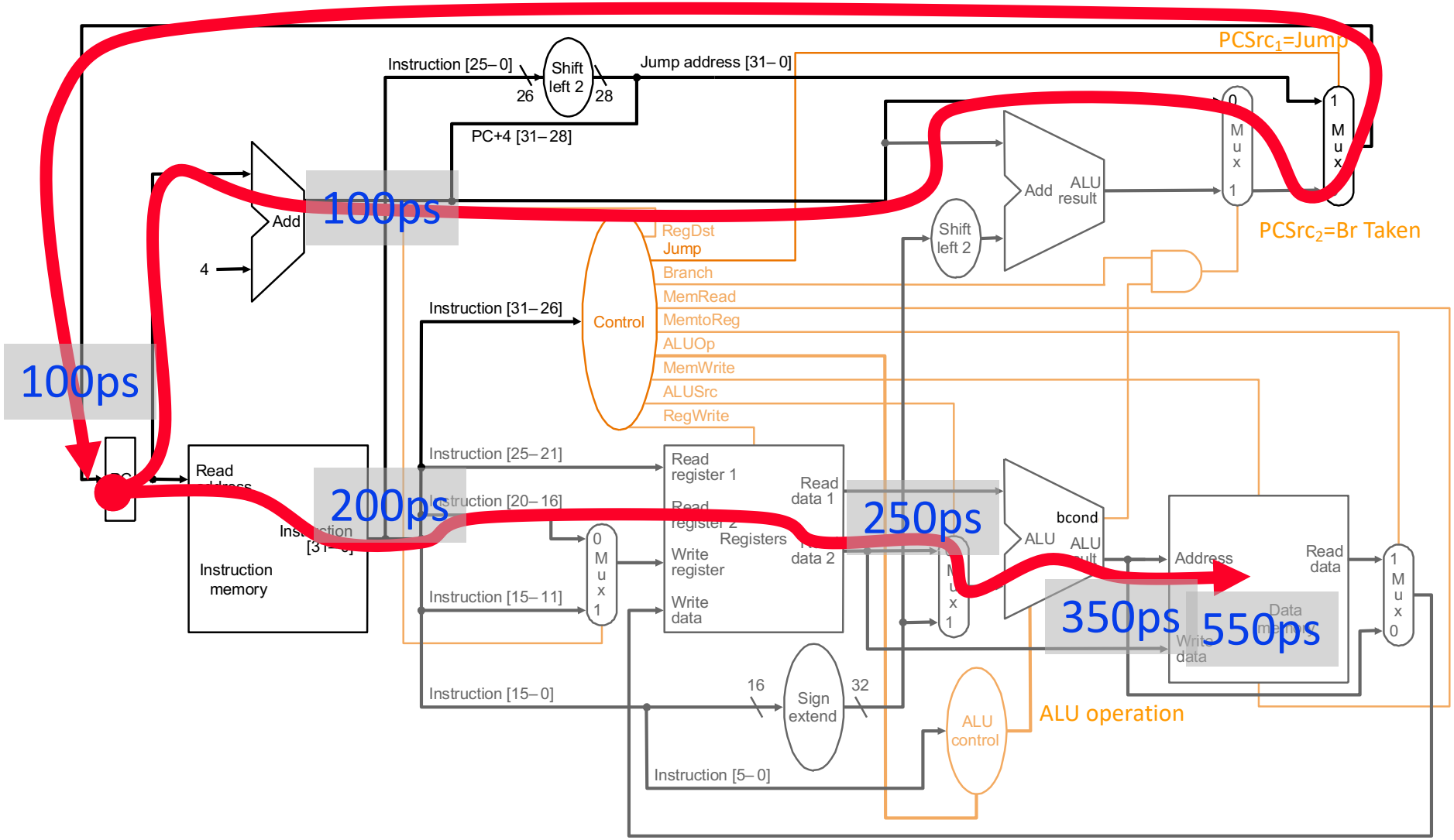
R-Type and I-Type ALU



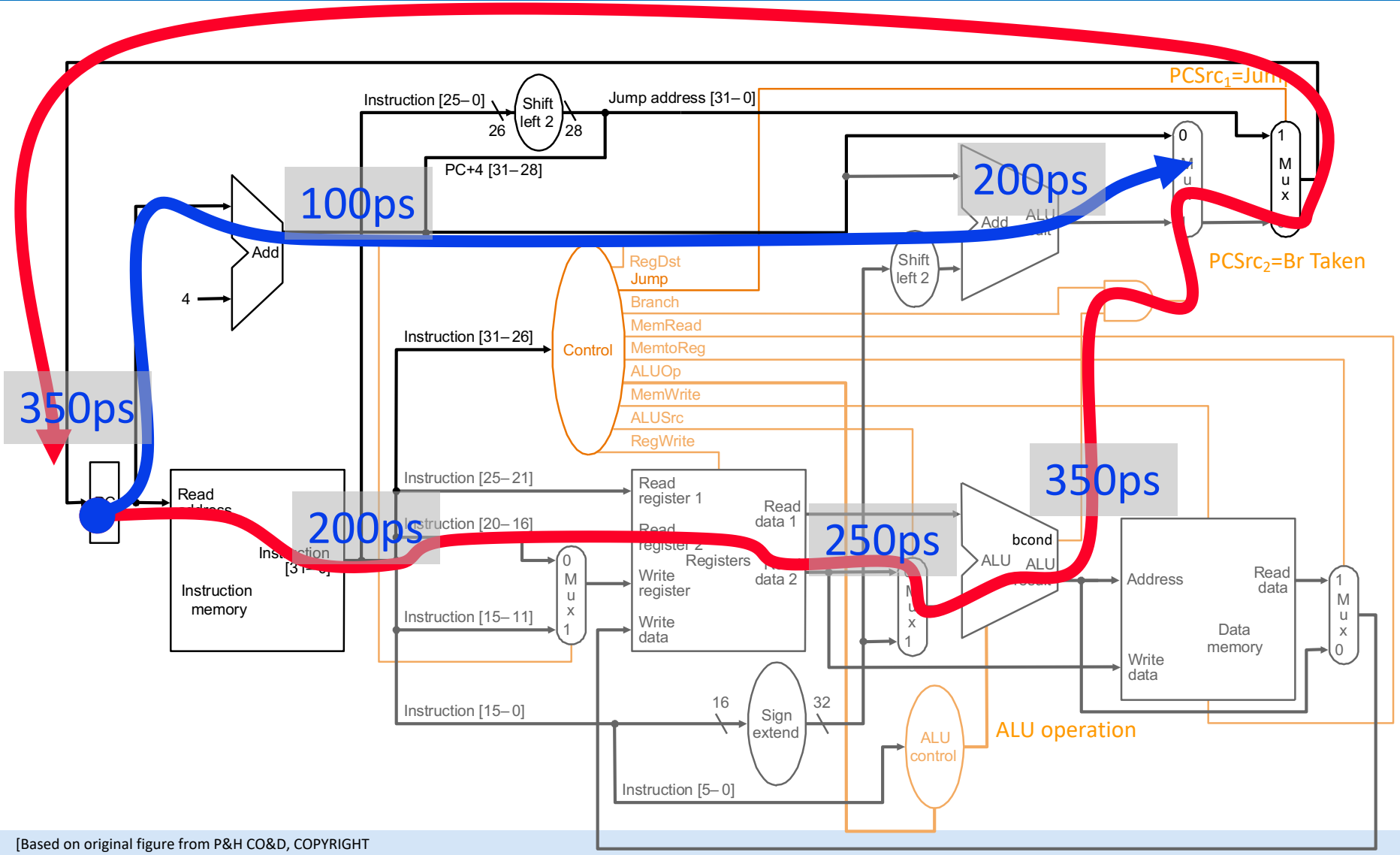
LW



SW

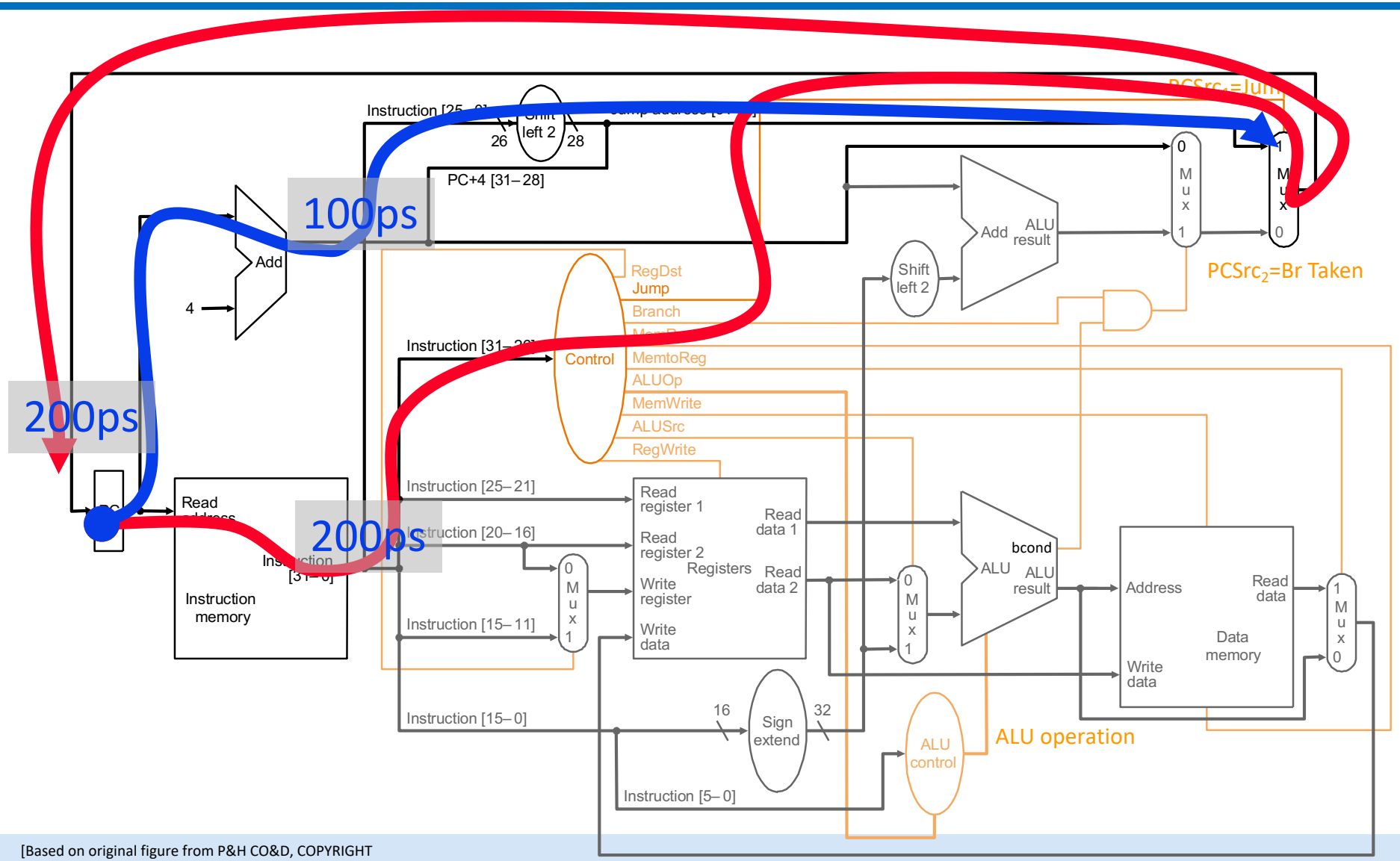


Branch Taken



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Jump



Example Single-Cycle Datapath Analysis

- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other logic or wire delay: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - Historical example:
    - CDC 5600: control store access took too long…

What is Really the Slowest Instruction to Process?

- Real world: **Memory is slow (not magic)**
- What if memory *sometimes* takes 150ns to access?
- Does it make sense to have a simple register to register add or jump to take {150ns + all else to perform a memory operation}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions require this?

Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- Not necessarily the simplest way to implement an ISA
  - Tough for complex instructions, e.g., REP MOVS (x86) or INDEX (VAX)
- Not easy to optimize/improve performance
  - Optimizing the common case (frequent instructions) does not work
  - Need to optimize the worst case all the time

(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long
- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case
- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

How does a single-cycle microarchitecture fare with respect to these principles?

Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles
  - Actually, this is true for **any system design**
    - Real architectures, buildings, bridges, train stations, …
    - Good consumer products
    - Security & safety-critical systems
    - Decision-making systems
    - …

Takeaways

- It all starts from the basic building blocks and design principles

- And, knowledge of how to use, apply, enhance them

- Underlying technology might change (e.g., steel vs. wood)
  - but methods of taking advantage of technology bear resemblance
  - methods used for design depend on the principles employed

Hardware Design

Multi-Cycle Microarchitecture

Multi-Cycle Microarchitectures

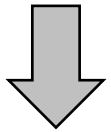
- Goal: Let each instruction take (close to) only as much time as it really needs

- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

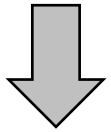
Multi-Cycle Microarchitecture

AS = Architectural (programmer visible) state

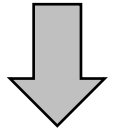
at the beginning of an instruction



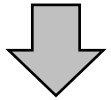
Step 1: Process part of instruction in one clock cycle



Step 2: Process part of instruction in the next clock cycle



...



AS' = Architectural (programmer visible) state

at the end of a clock cycle

Benefits of Multi-Cycle Design

- Critical path design
  - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- Bread and butter (common case) design
  - Can optimize the number of states it takes to execute "important" instructions that make up much of the execution time
- Balanced design
  - No need to provide more capability or resources than really needed
    - An instruction that needs resource X multiple times does **not** require multiple X's to be implemented
    - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for microarchitectural registers
  - Register setup/hold overhead (i.e., sequencing overhead) is paid multiple times for an instruction
- Limited concurrency
  - Only a small part of the machine is used at any point in time

Recall: Performance Analysis

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}** CPI: Cycles Per Instruction
- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**
- Single-cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

In multi-cycle, we have two degrees of freedom to optimize independently

Hardware Design

A Multi-Cycle Microarchitecture:
A Closer Look

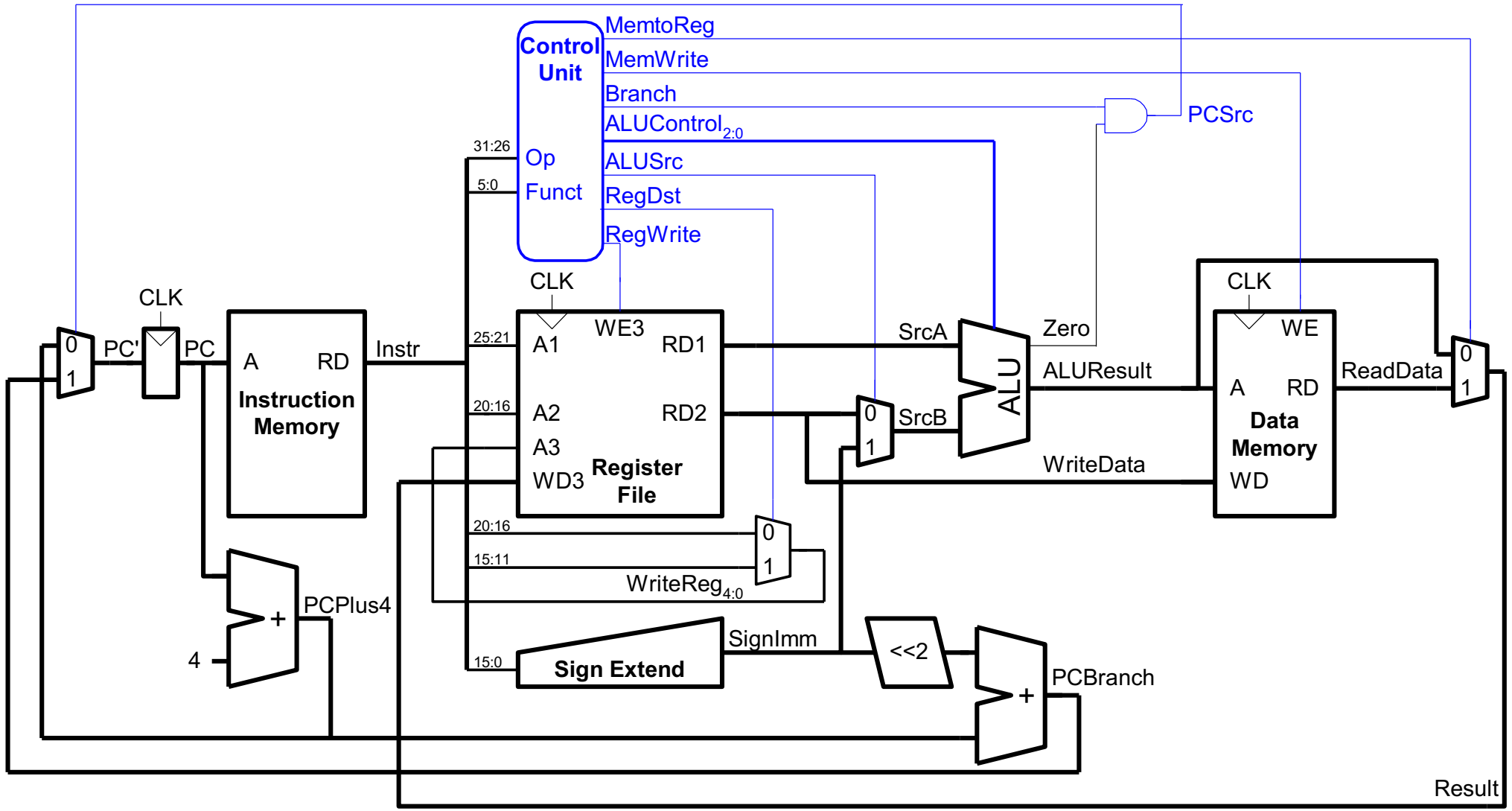
Multi-Cycle Microarchitectures

- Key Idea for Realization
  - One can implement the "process instruction" step as a finite state machine that sequences between states and eventually returns to the "fetch instruction" state
  - A state is defined by the control signals asserted in it
  - Control signals for the next state are determined in the current state

A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into "states"
  - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
  - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a finite state machine
- In a state (clock cycle), control signals control two things:
  - How the datapath should process the data
  - How to generate the control signals for the (next) clock cycle

Remember the Single-Cycle Uarch



Why Do We Want Multi-Cycle?

- Single-cycle microarchitecture:
  -- cycle time limited by the longest instruction (`lw`) → low clock frequency
  -- three adders/ALUs and two memories → high hardware cost

- Multi-cycle microarchitecture:
  + higher clock frequency
  + simpler instructions take only a few clock cycles
  + reuse expensive hardware across multiple cycles
  -- hardware overhead for storing intermediate results
  -- sequential logic overhead is paid many times for each instruction

- Multi-cycle requires the same design steps as a single cycle:
  - datapath
  - control logic

What Can We Optimize with Multi-Cycle

- Single-cycle microarchitecture uses two memories
  - One memory stores instructions, the other data
  - We want to use a single memory (lower cost)
- Single-cycle microarchitecture needs three adders
  - ALU, PC, Branch address calculation
  - We want to use only one ALU for all operations (lower cost)
- Single-cycle microarchitecture: each instruction takes one cycle
  - The slowest instruction slows down every single instruction
  - We want to determine clock cycle time independently of instruction processing time
    - Divide each instruction into multiple clock cycles
    - Simpler instructions can be very fast (compared to the slowest)

Hardware Design

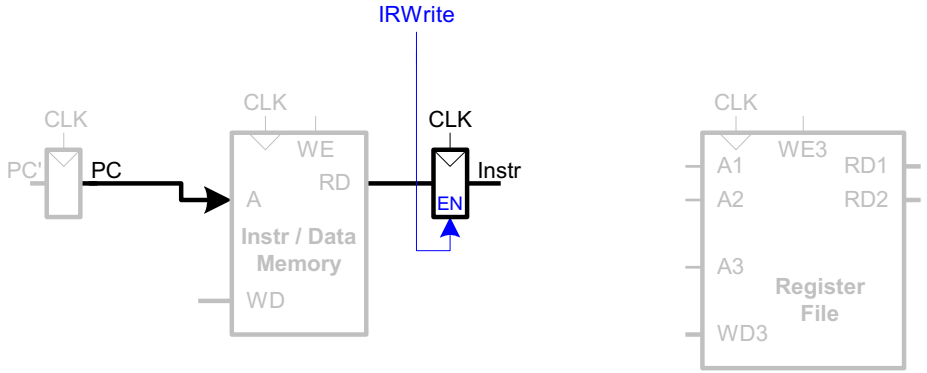
Multi-Cycle Datapath

Consider the lw Instruction

- For an instruction such as: `lw $t0, 0x20($t1)`
- We need to:
  - Read the instruction from memory
  - Then read **`$t1`** from the register array
  - Add the immediate value (**`0x20`**) to calculate the memory address
  - Read the content of this address
  - Write to the register **`$t0`** this content

Multi-Cycle Datapath: Instruction Fetch

- We will consider lw, but fetch is the same for all instructions
  - STEP 1: Fetch instruction



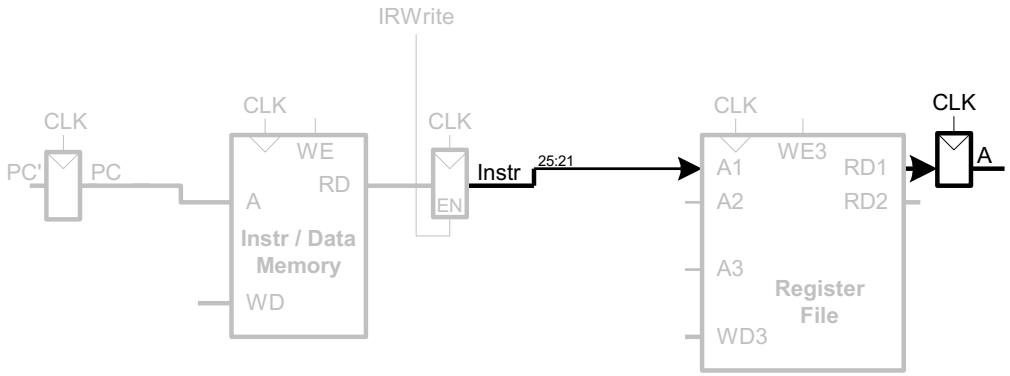

```
read from the memory location [rs]+imm to location [rt]
```

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: lw register read

STEP 2: Fetch Data

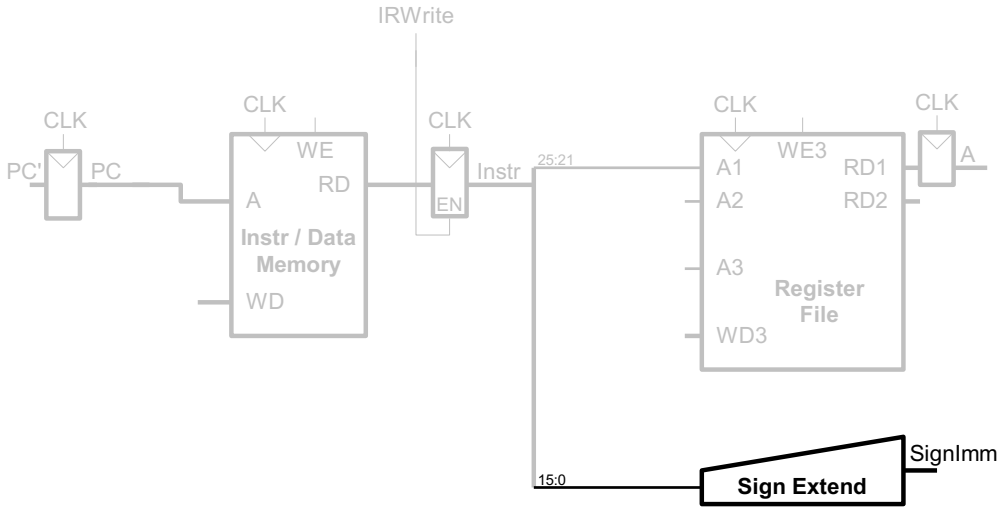


I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: lw immediate

STEP 2: Extend Sign

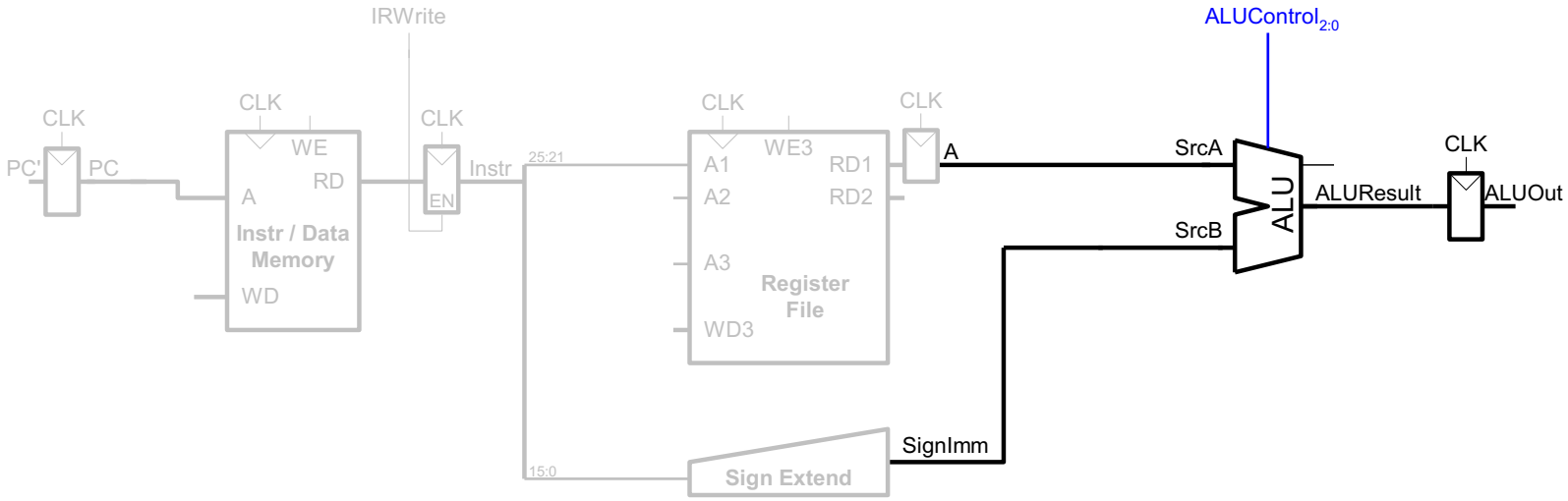


I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: lw address

STEP 3: Calculate Address



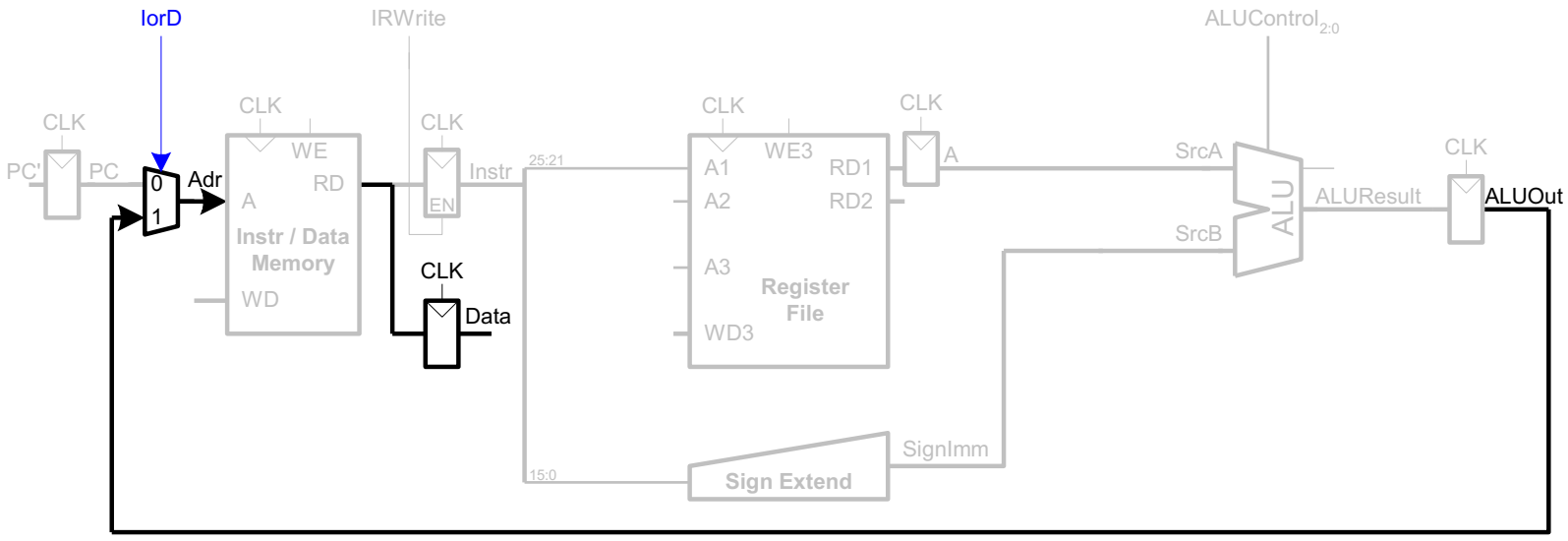
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: lw memory read

STEP 4: Fetch Data

Now we can use **a single memory** to
both fetch an instruction and fetch an operand
(in different clock cycles)

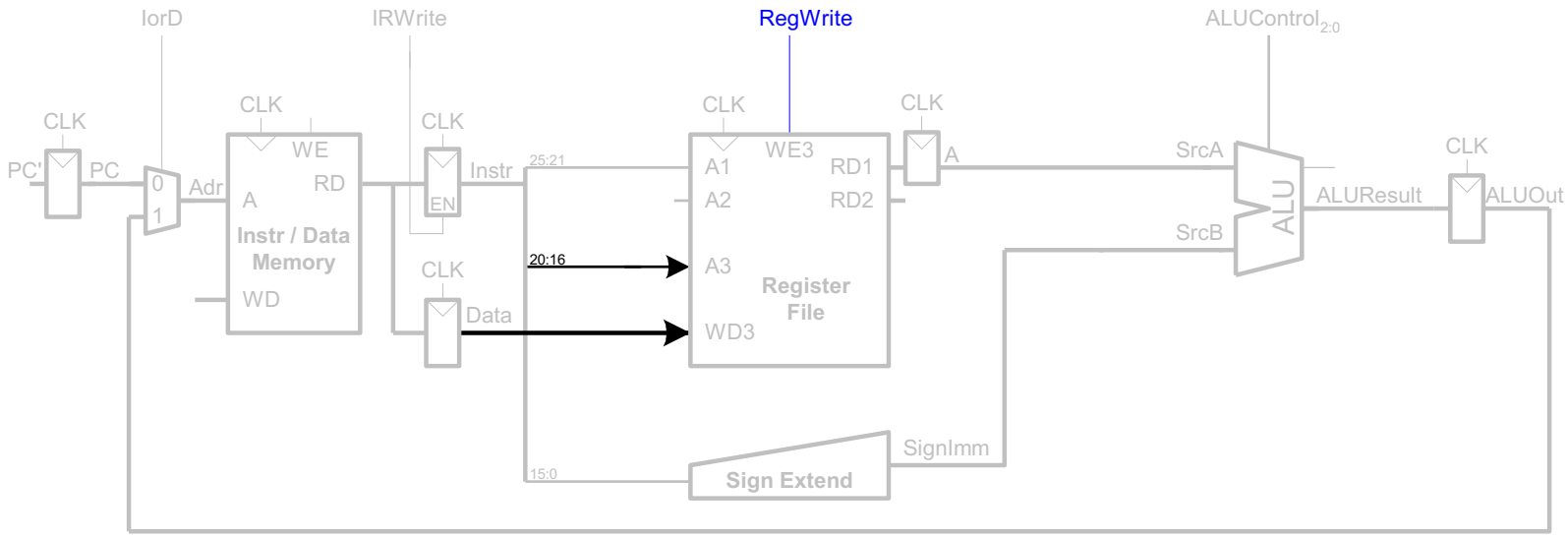


I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: lw write register

STEP 5: Write Register



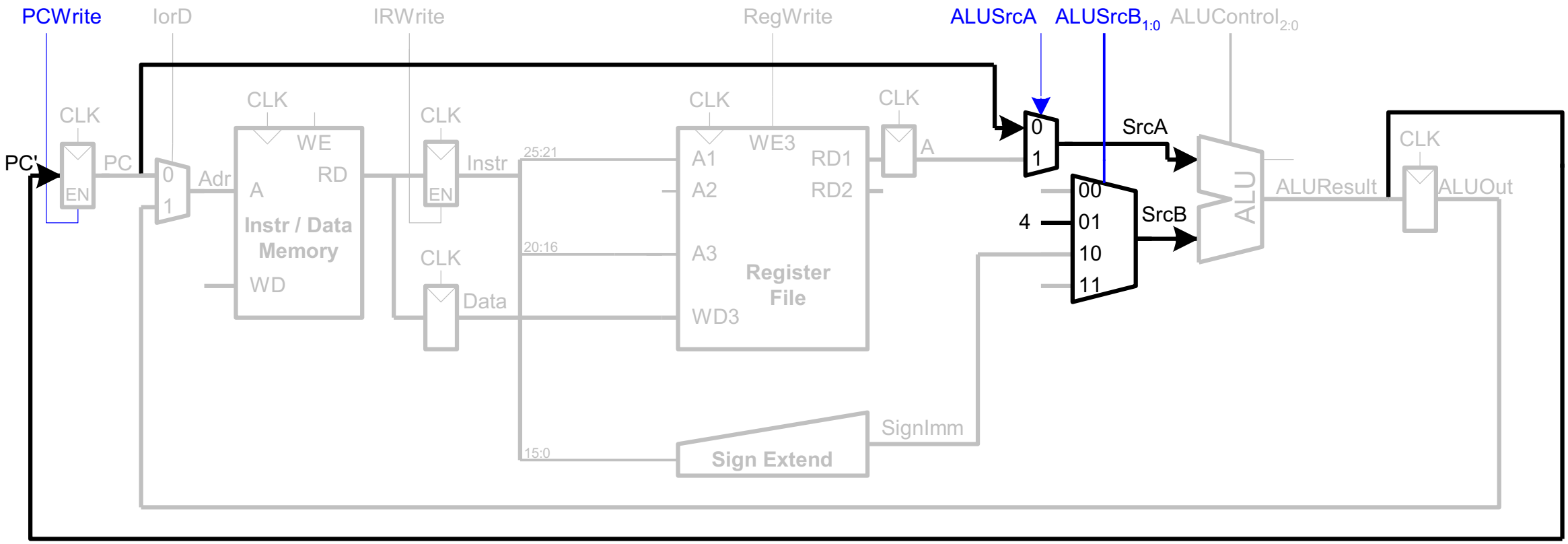
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: increment PC

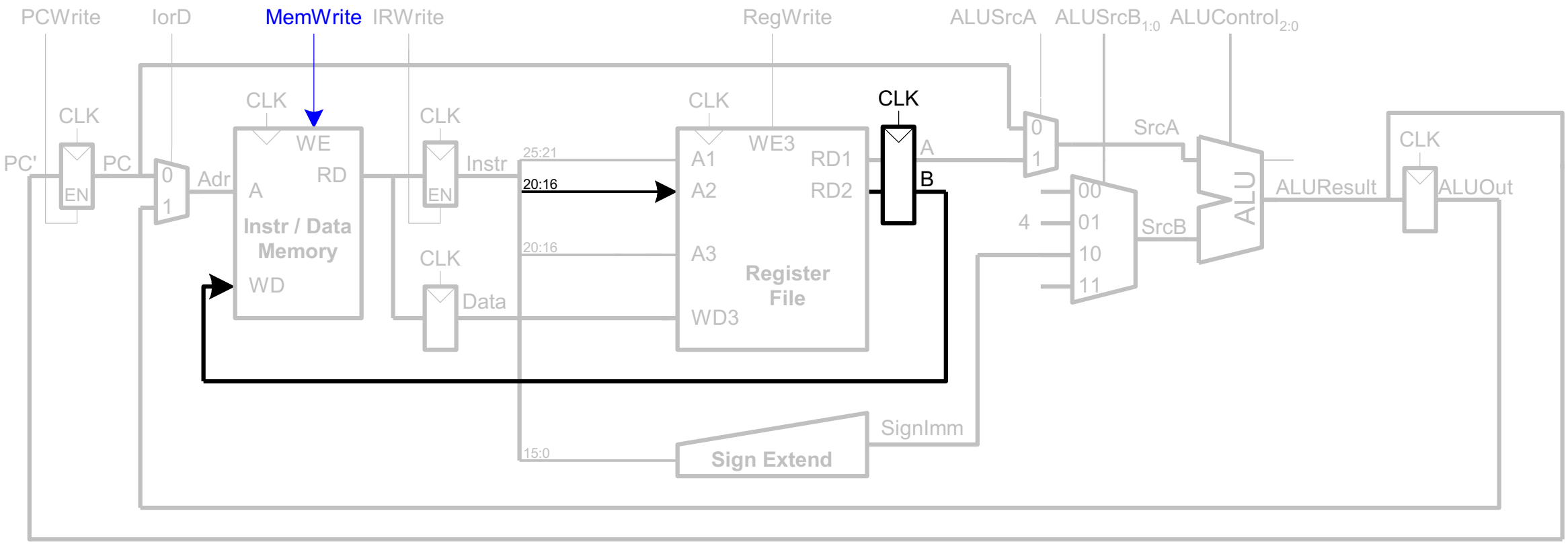
Concurrent Step

Now we can use **a single ALU** to both increment PC and do address calculation or arithmetic operations (in different clock cycles)



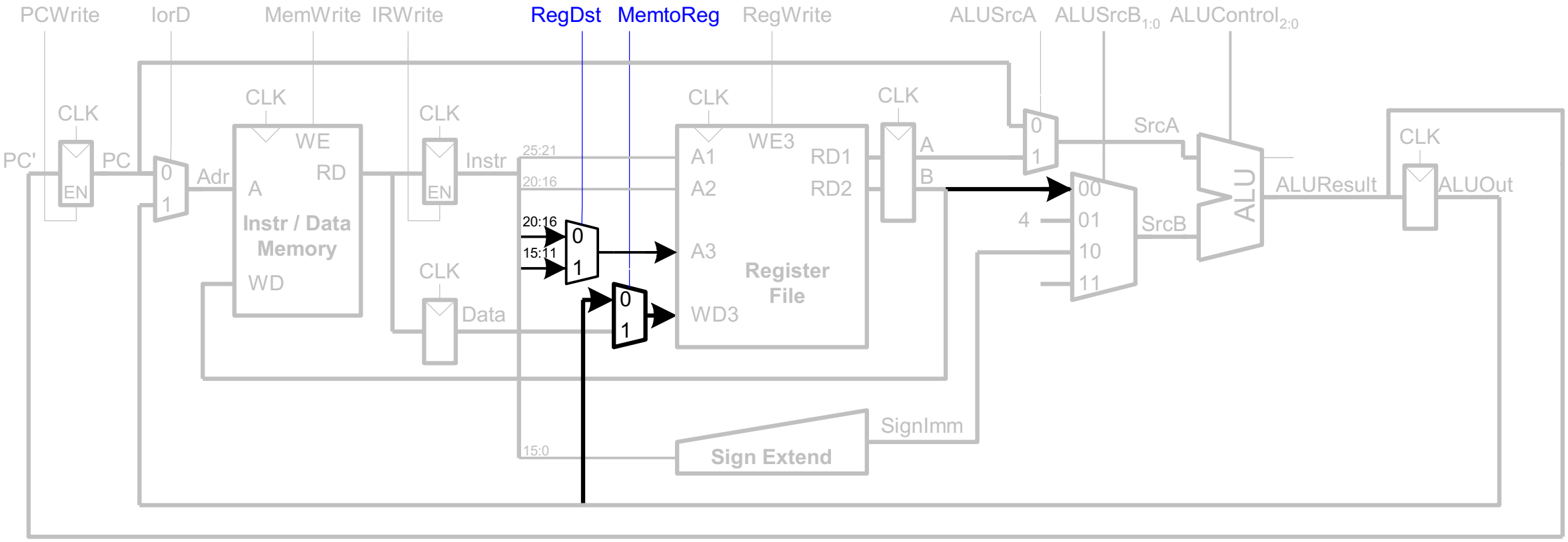
Multi-Cycle Datapath: sw

- Write data in rt to memory



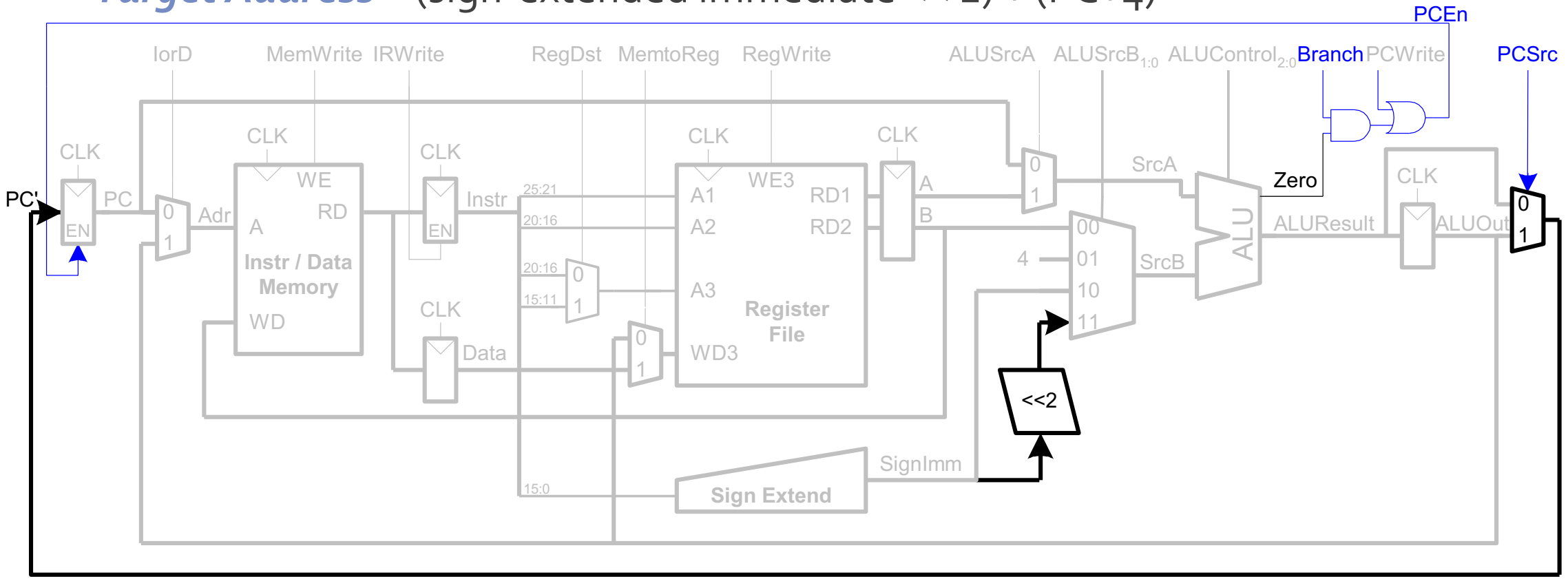
Multi-Cycle Datapath: R-type Instructions

- Read from rs and rt
  - Write ALUOut to register file
  - Write to rd (instead of rt)

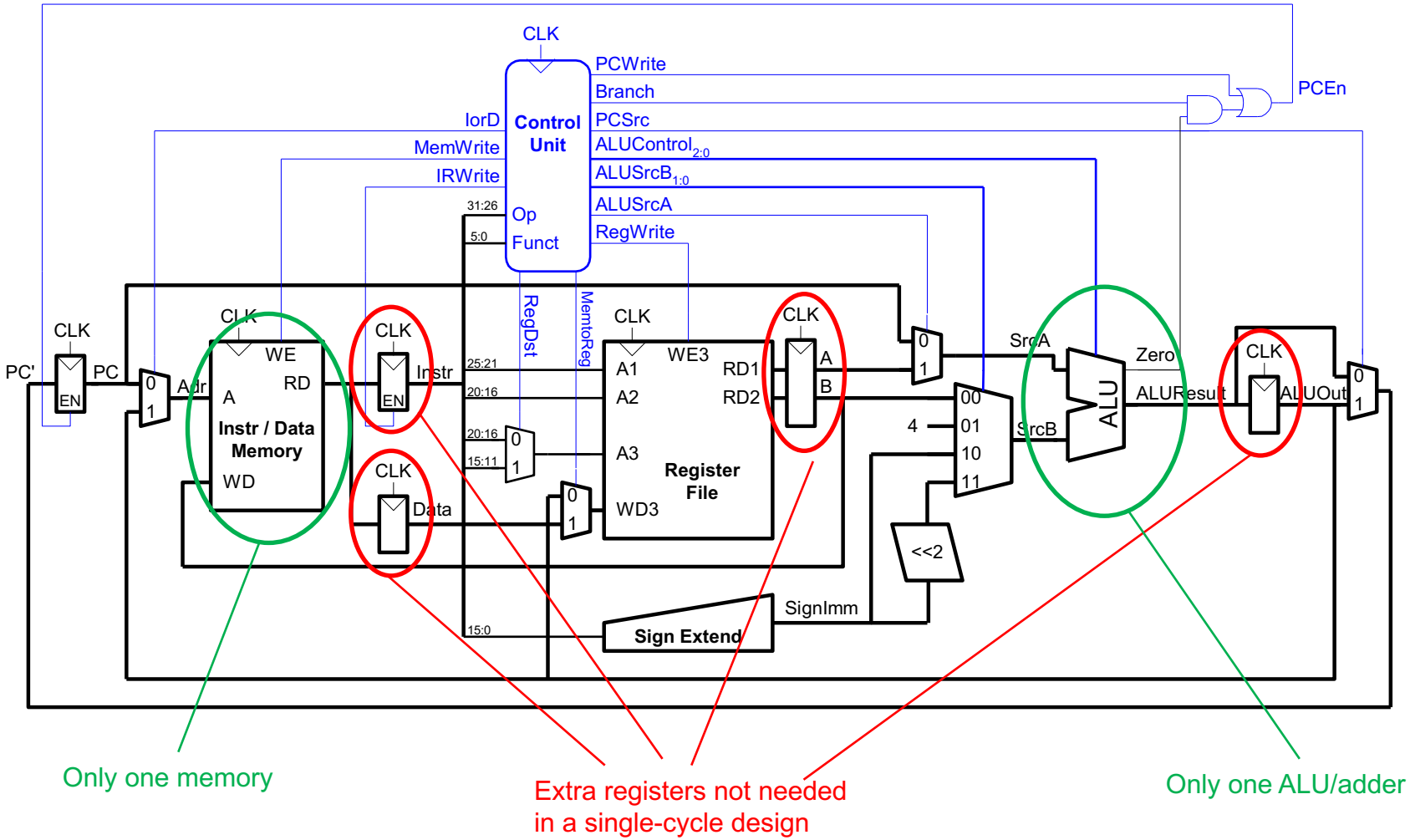


Multi-Cycle Datapath: beq

- Determine whether values in rs and rt are equal
  - Calculate branch target address:
    Target Address = (sign-extended immediate << 2) + (PC+4)



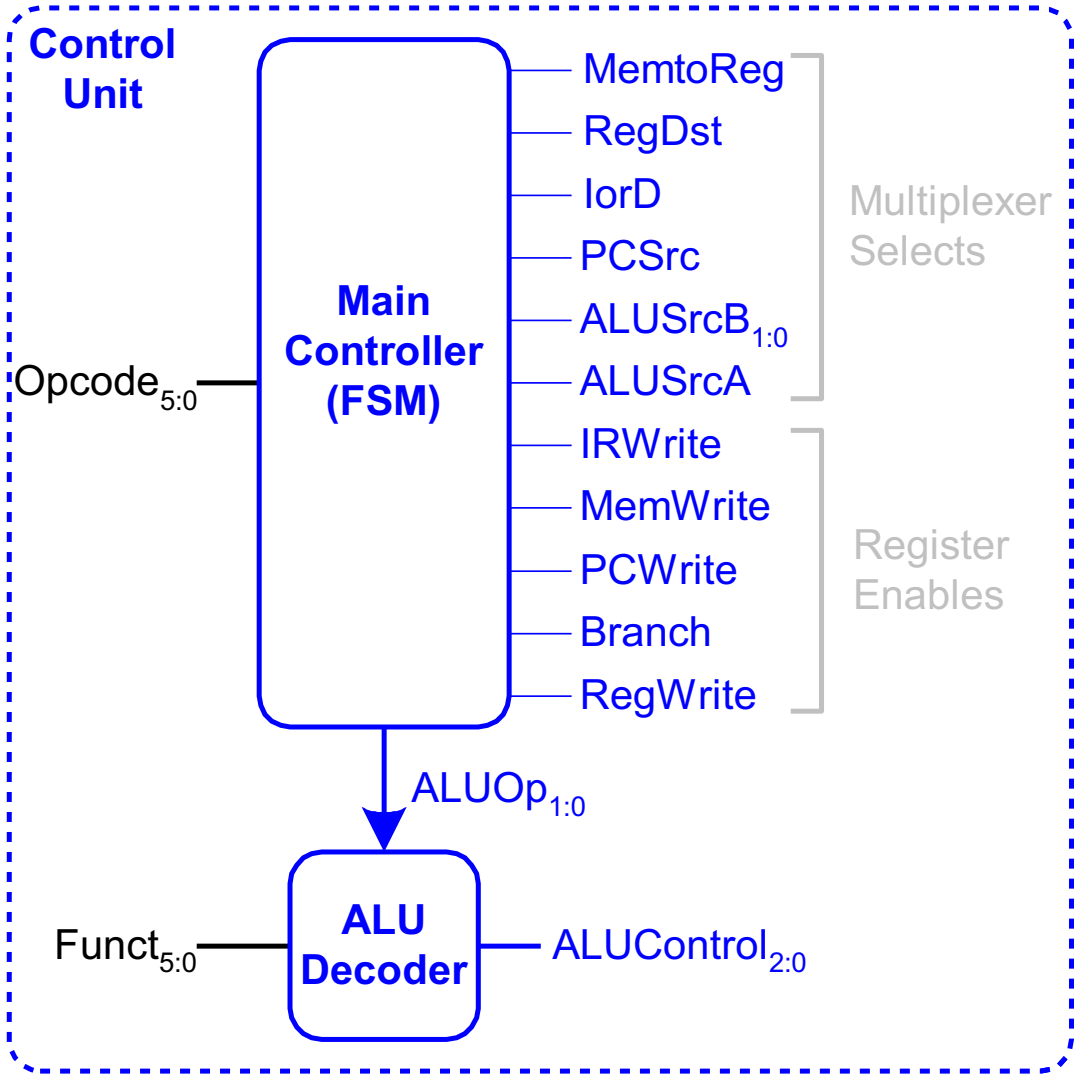
Complete Multi-Cycle Processor



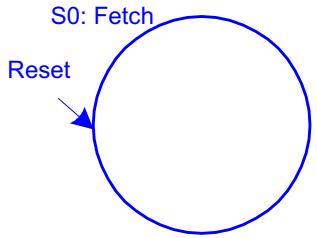
Hardware Design

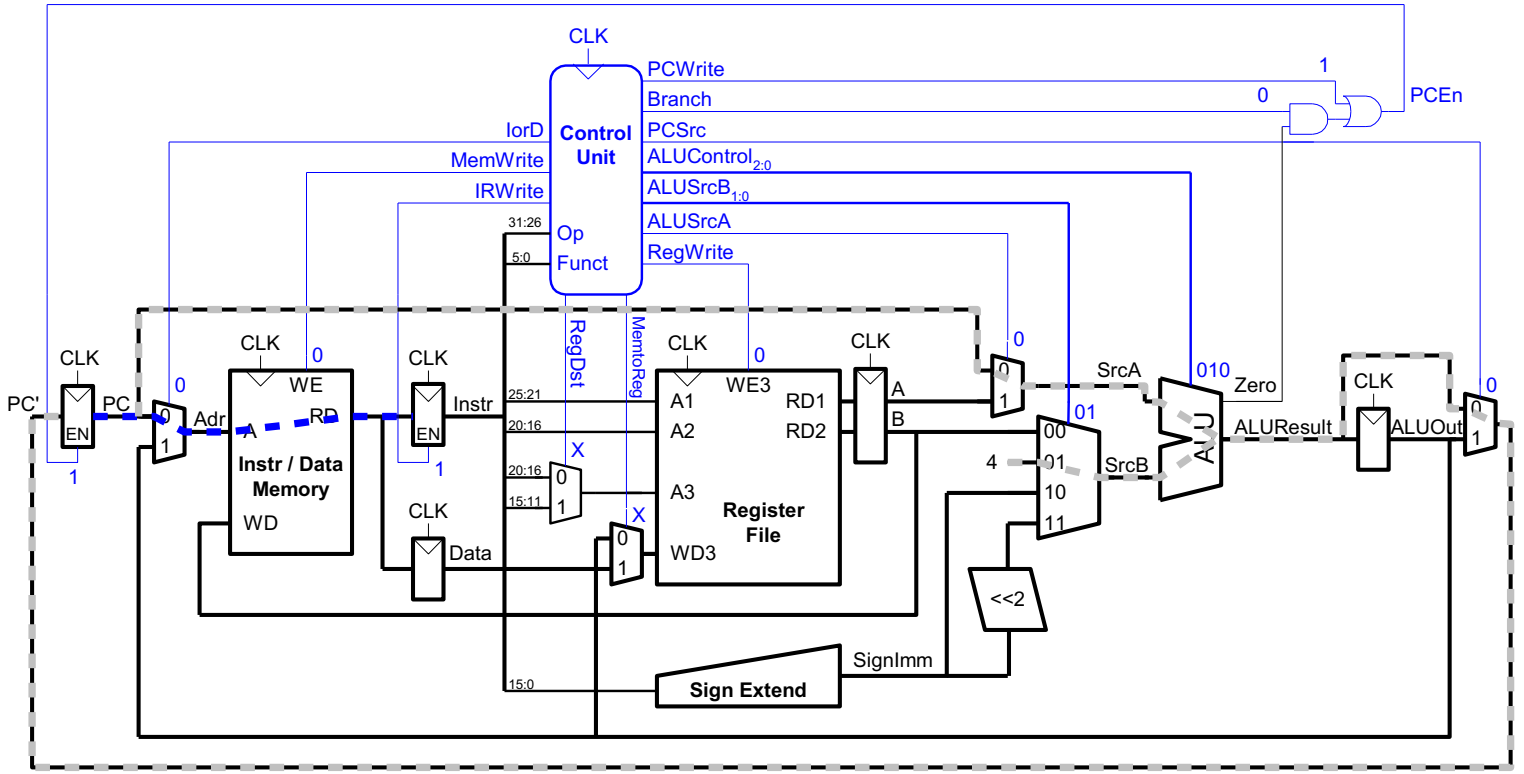
Multi-Cycle Control Logic

Control Unit

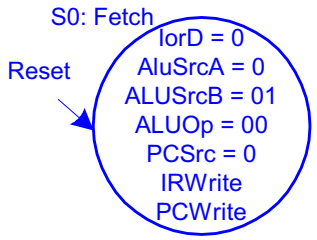


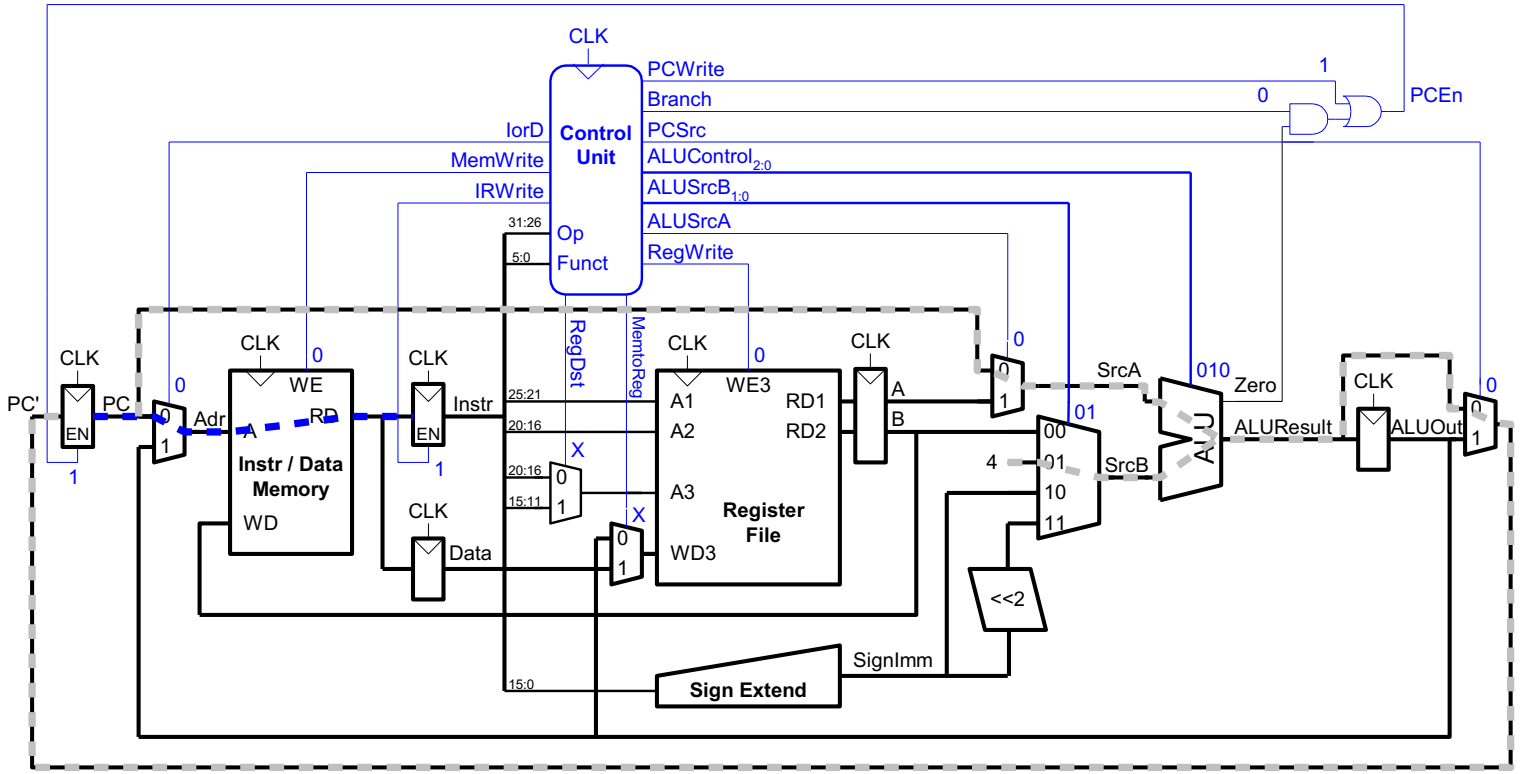
Main Controller FSM: Fetch



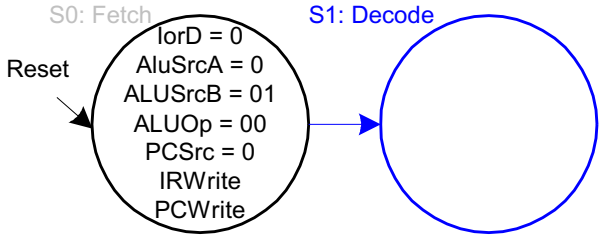


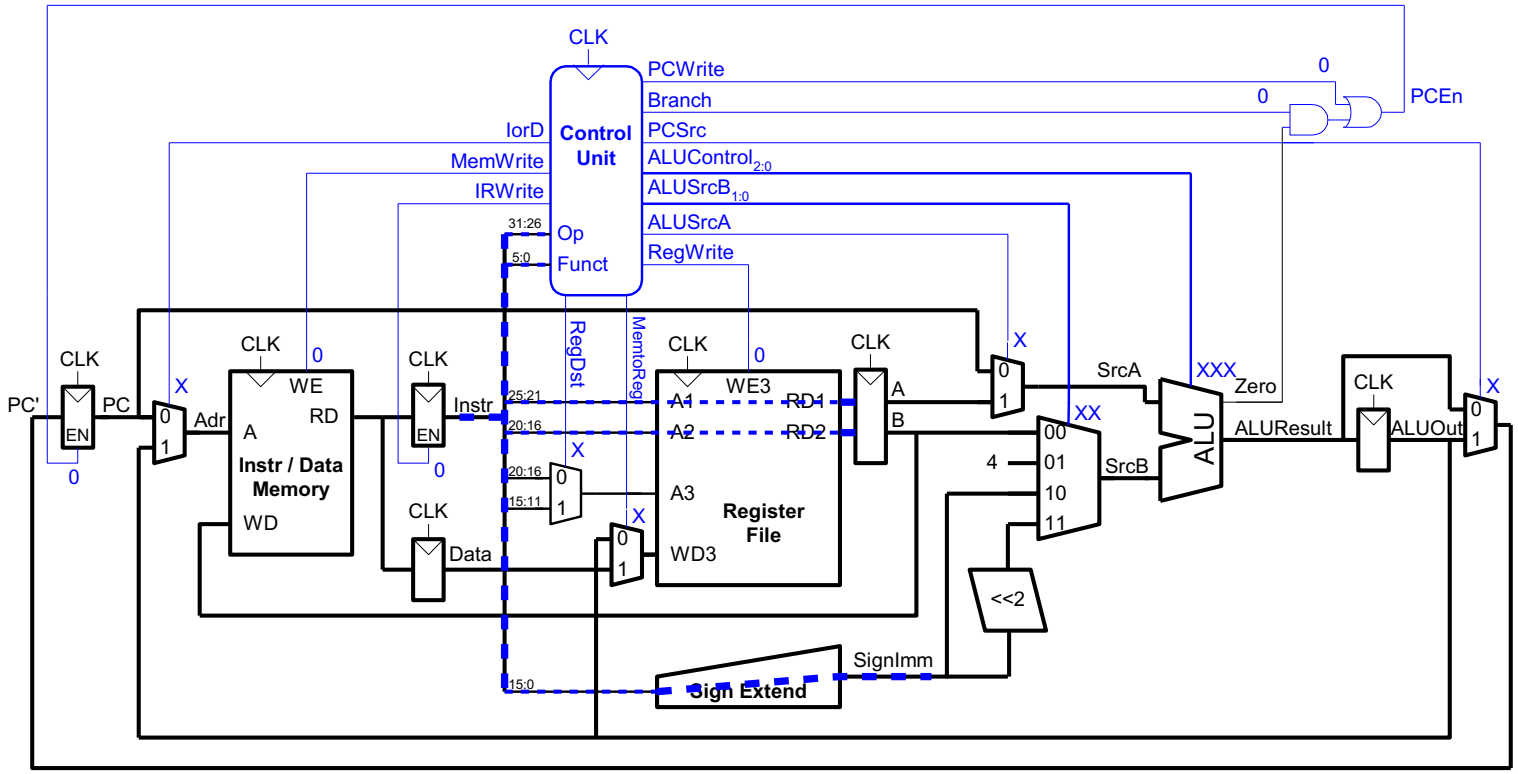
Main Controller FSM: Fetch



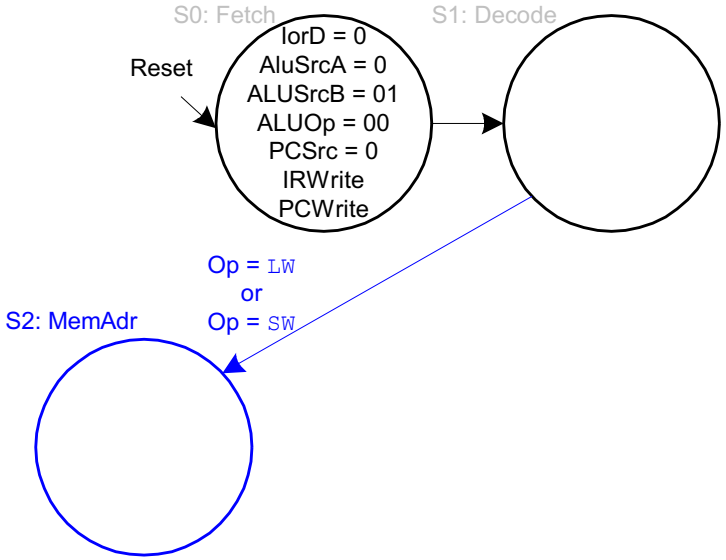


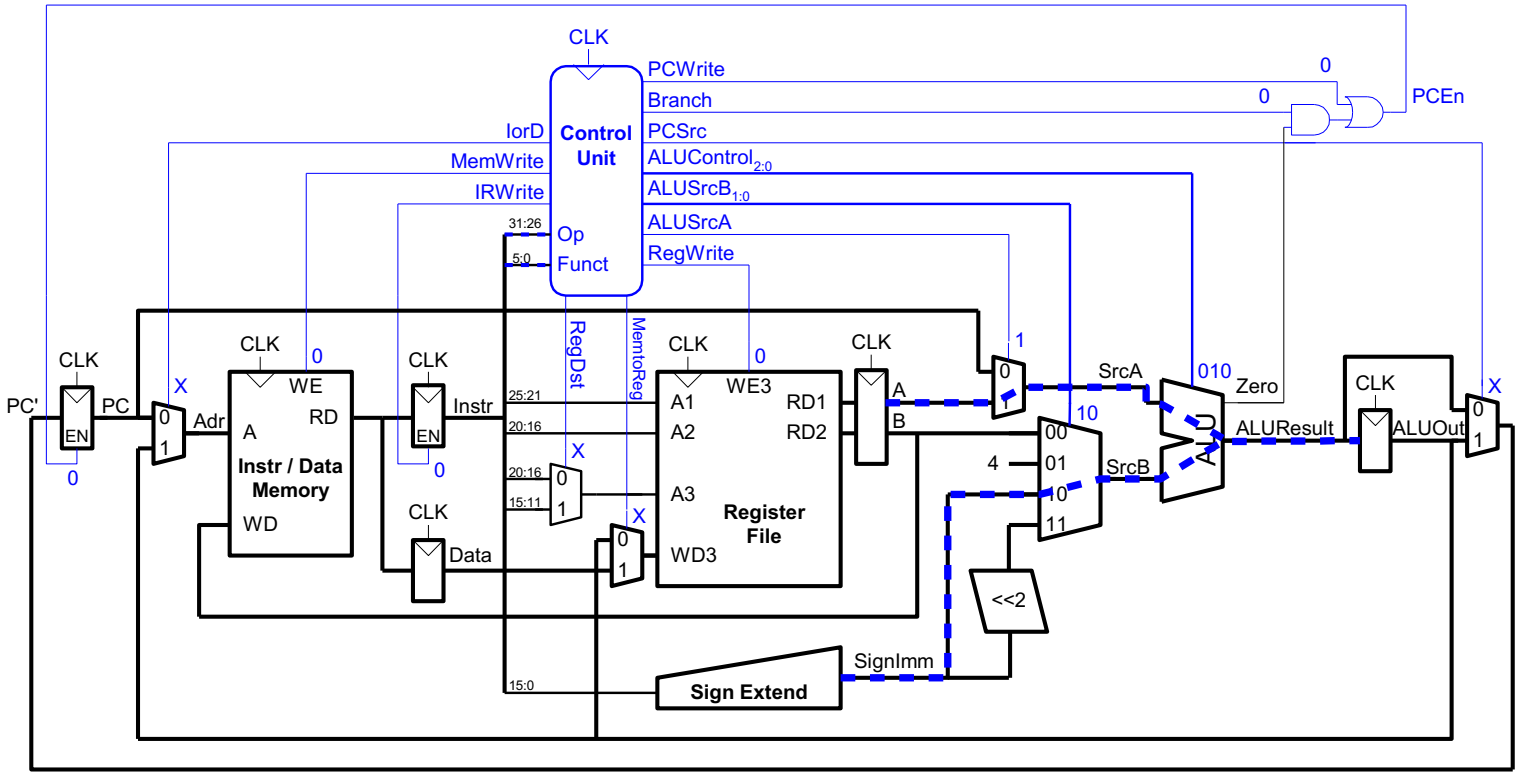
Main Controller FSM: Decode



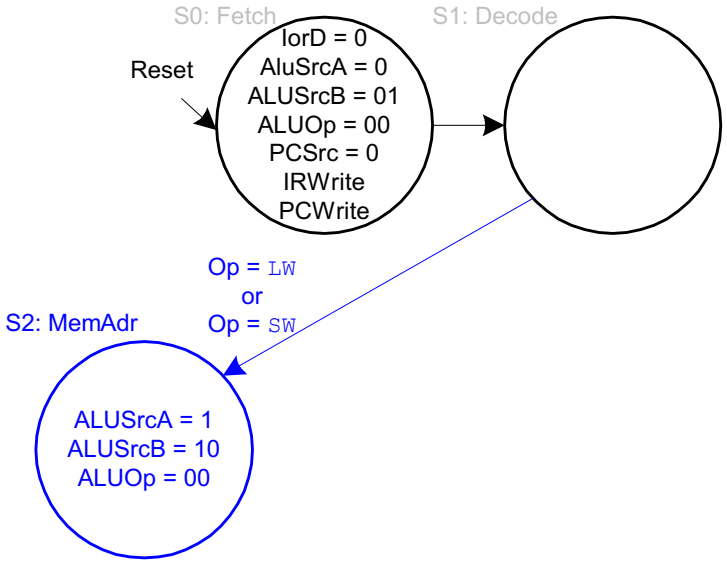


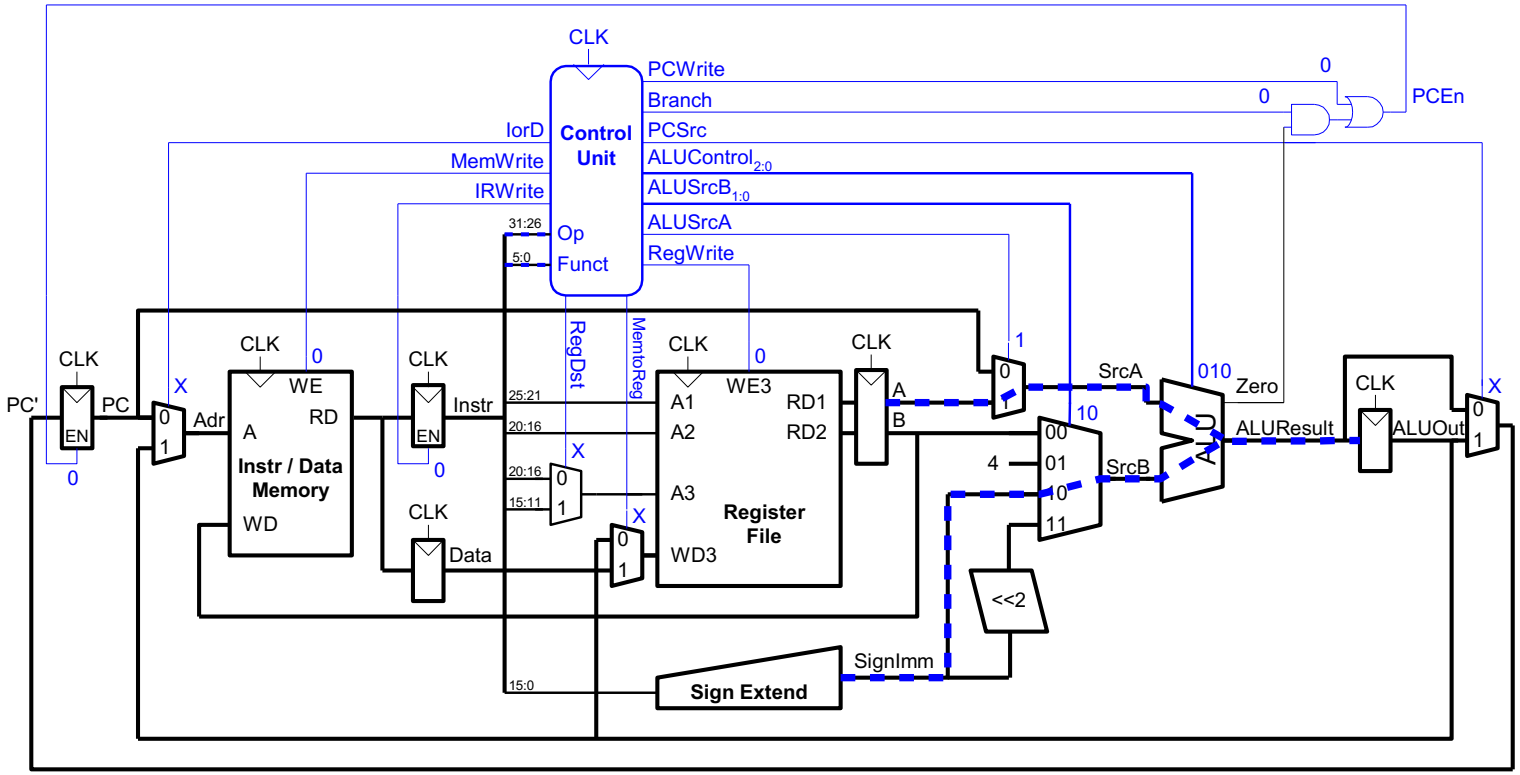
Main Controller FSM: Address Calculation



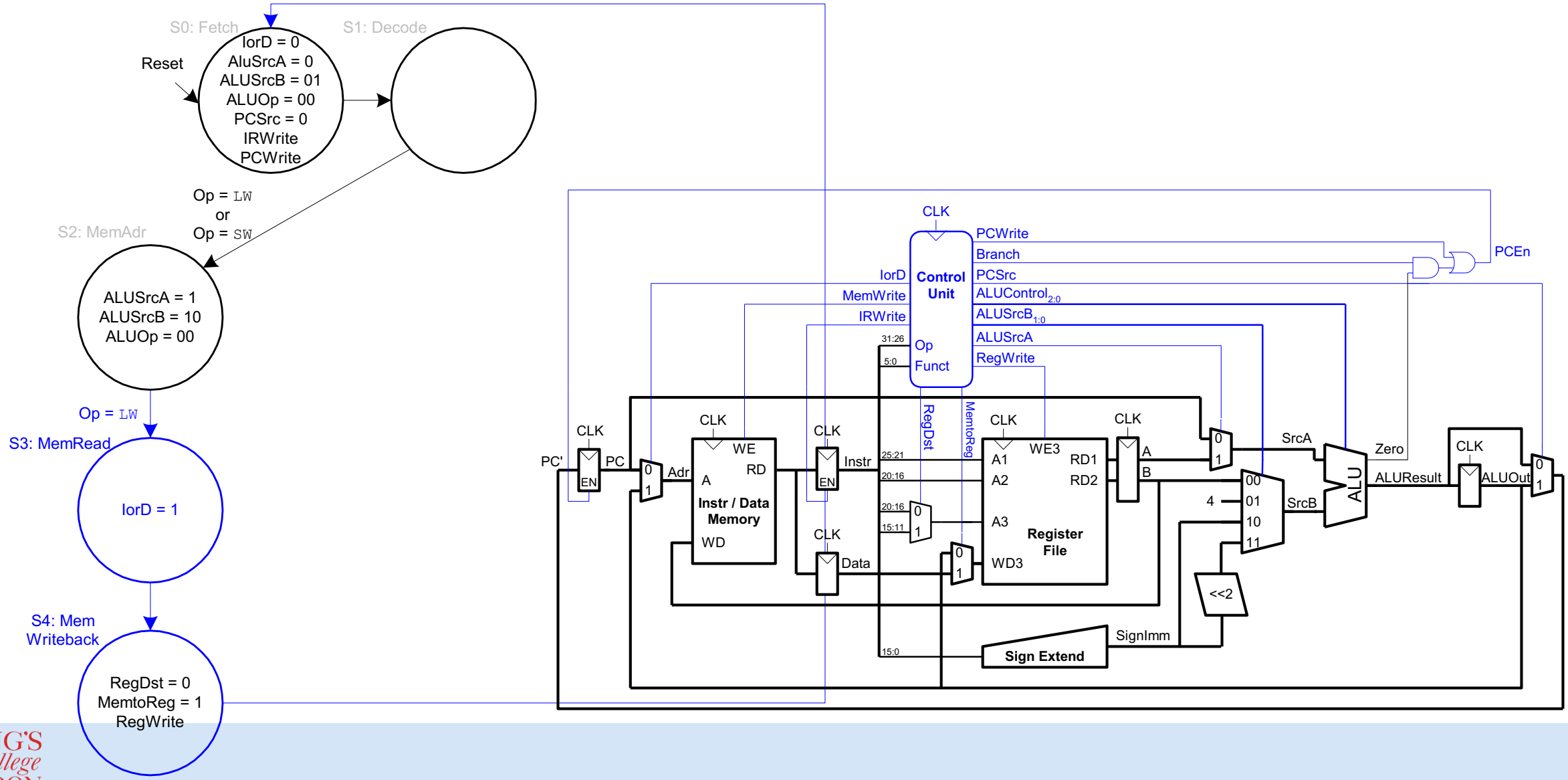


Main Controller FSM: Address Calculation

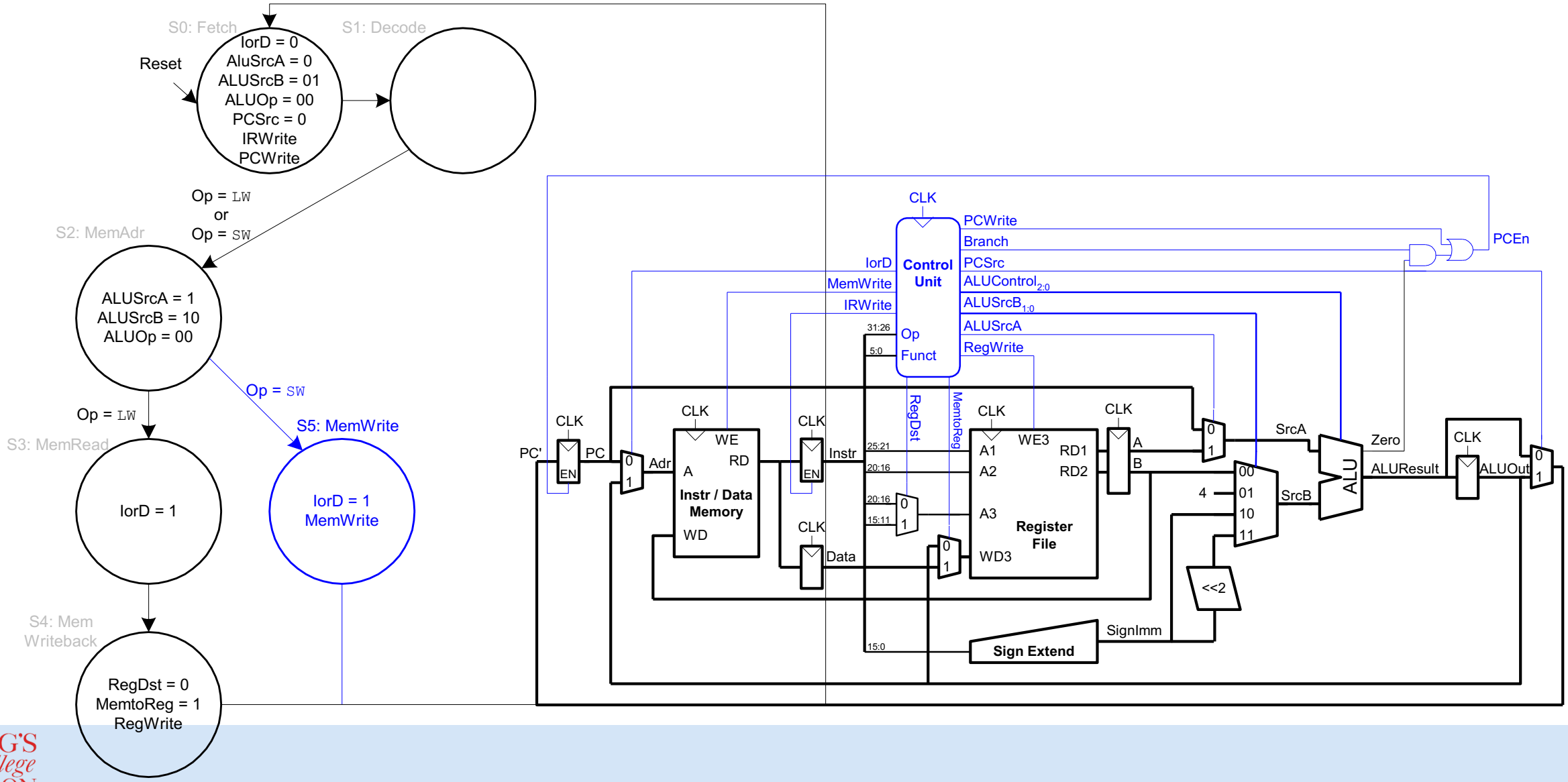




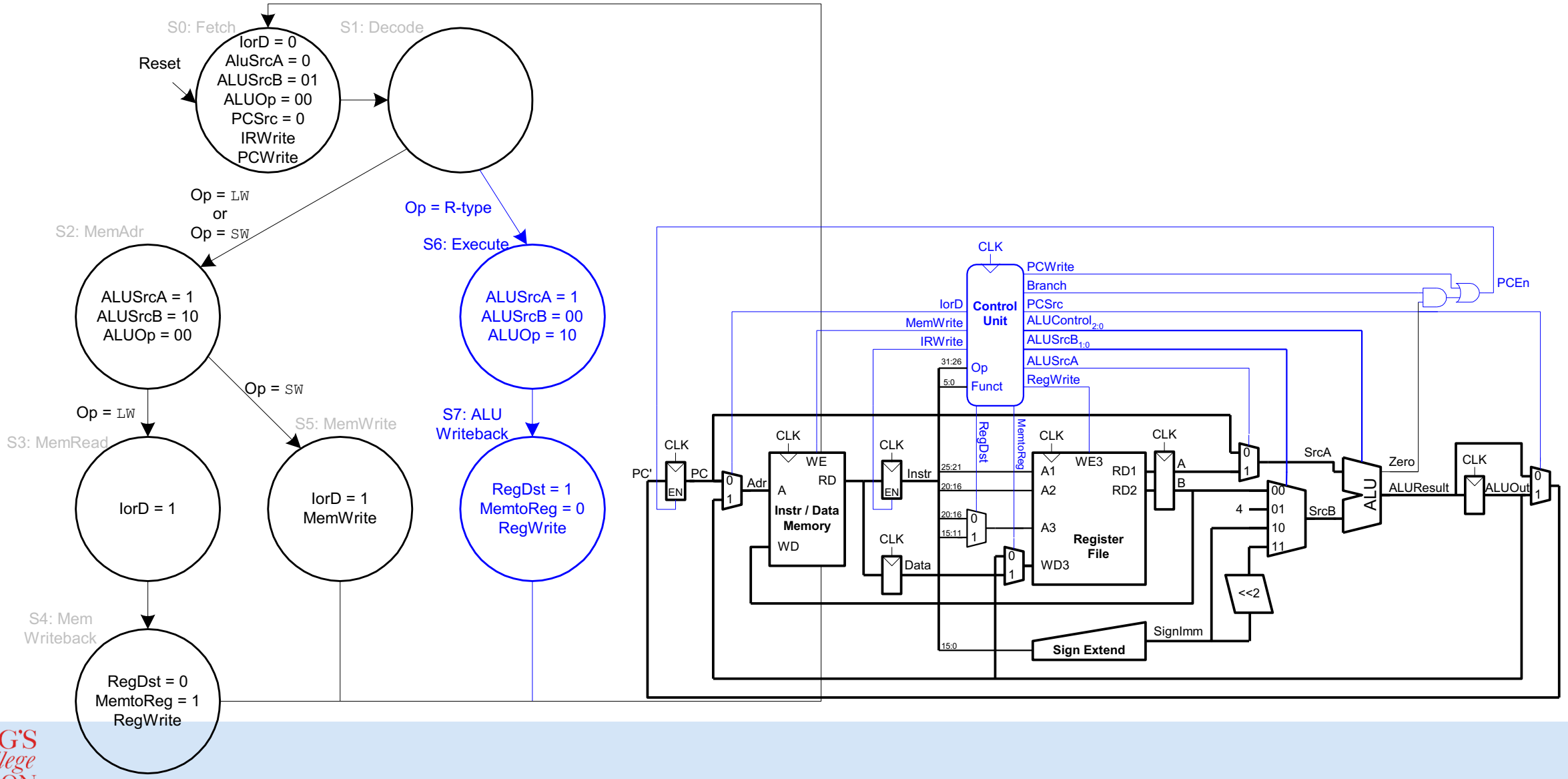
Main Controller FSM: lw



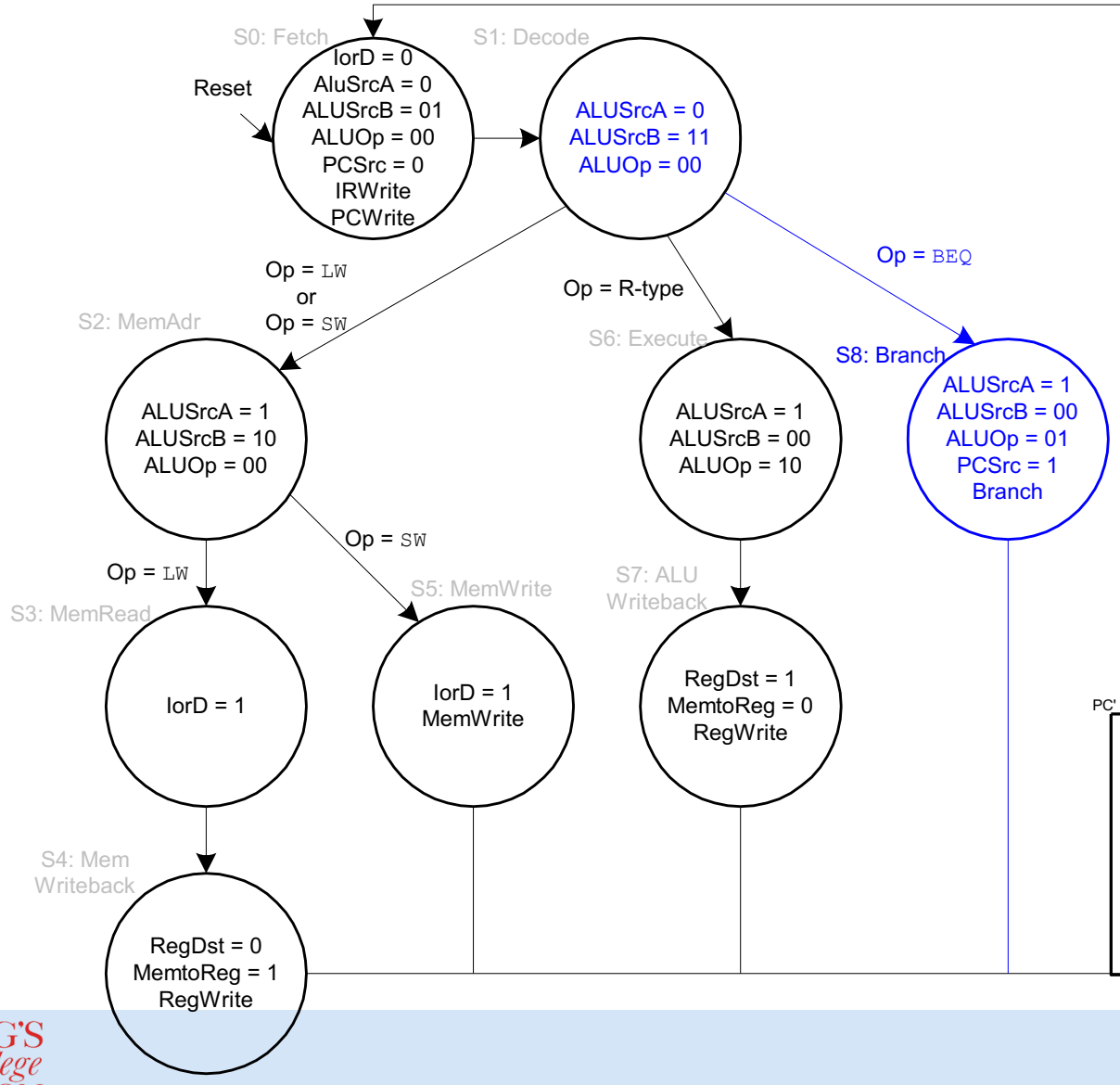
Main Controller FSM: sw

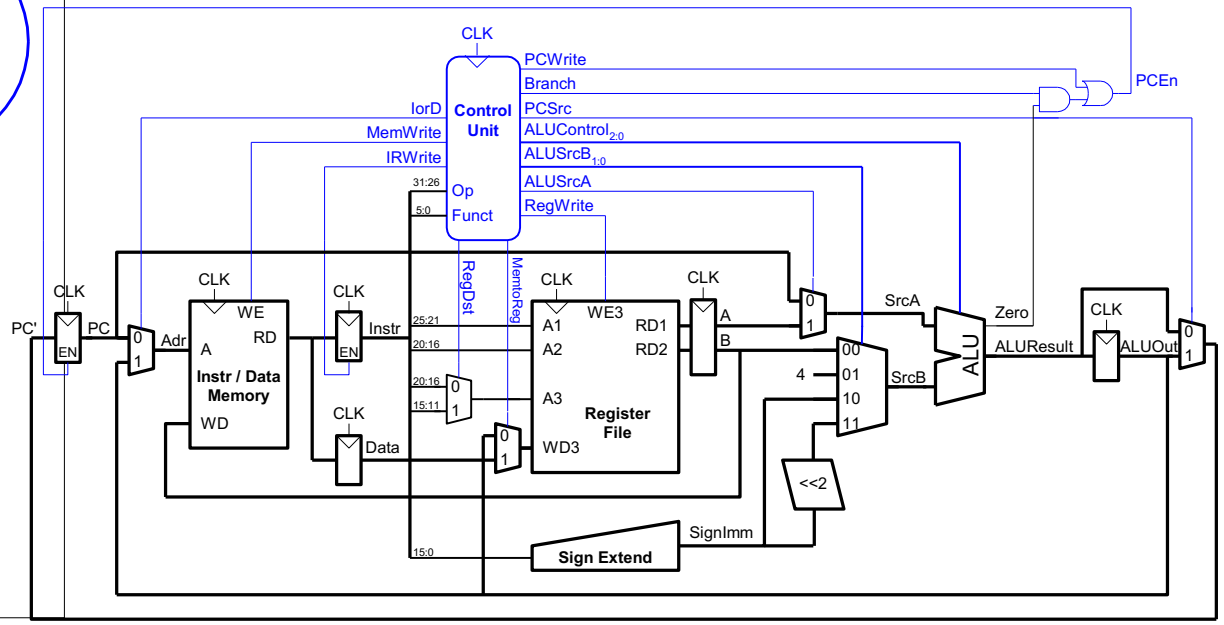


Main Controller FSM: R-Type

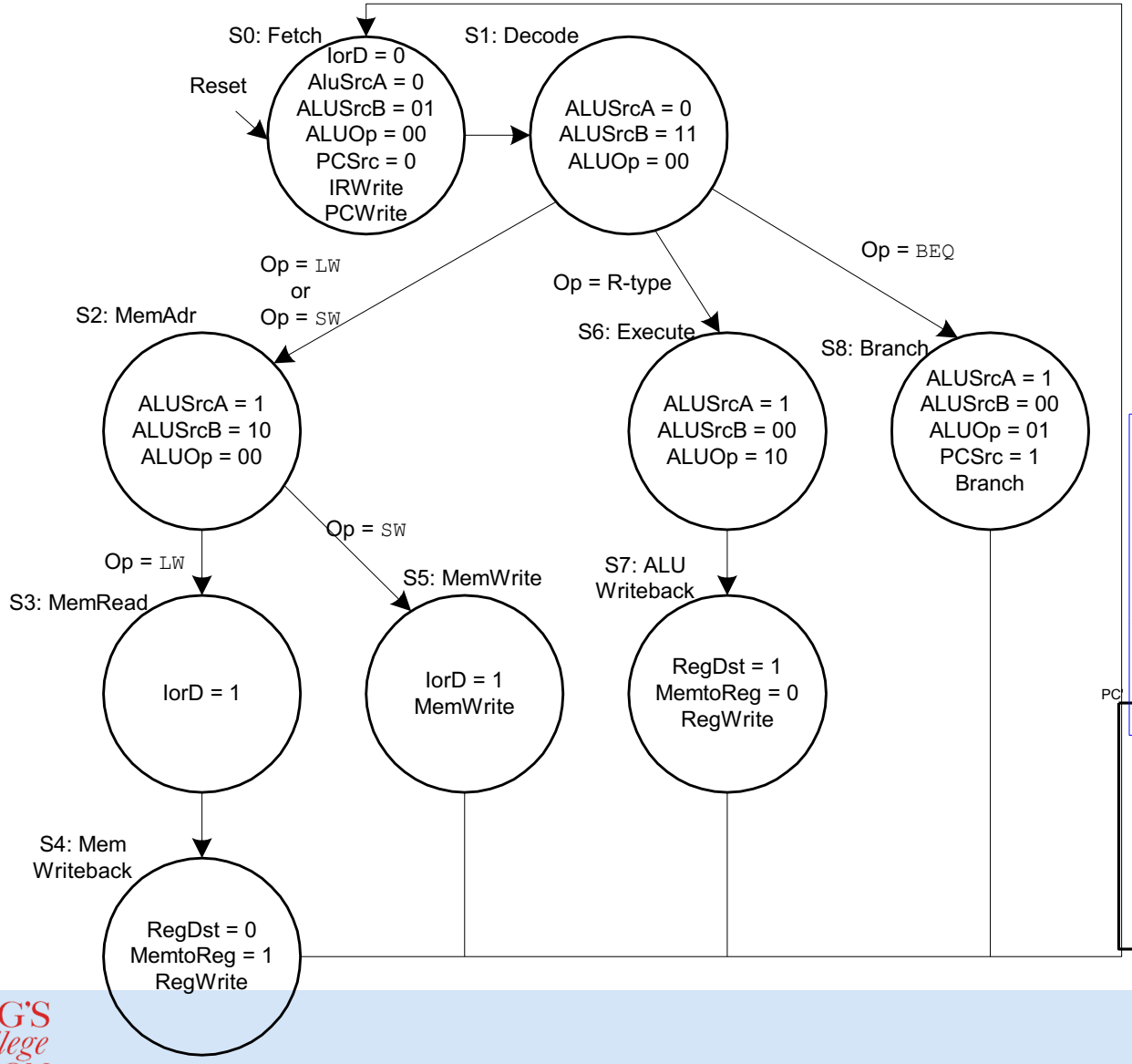


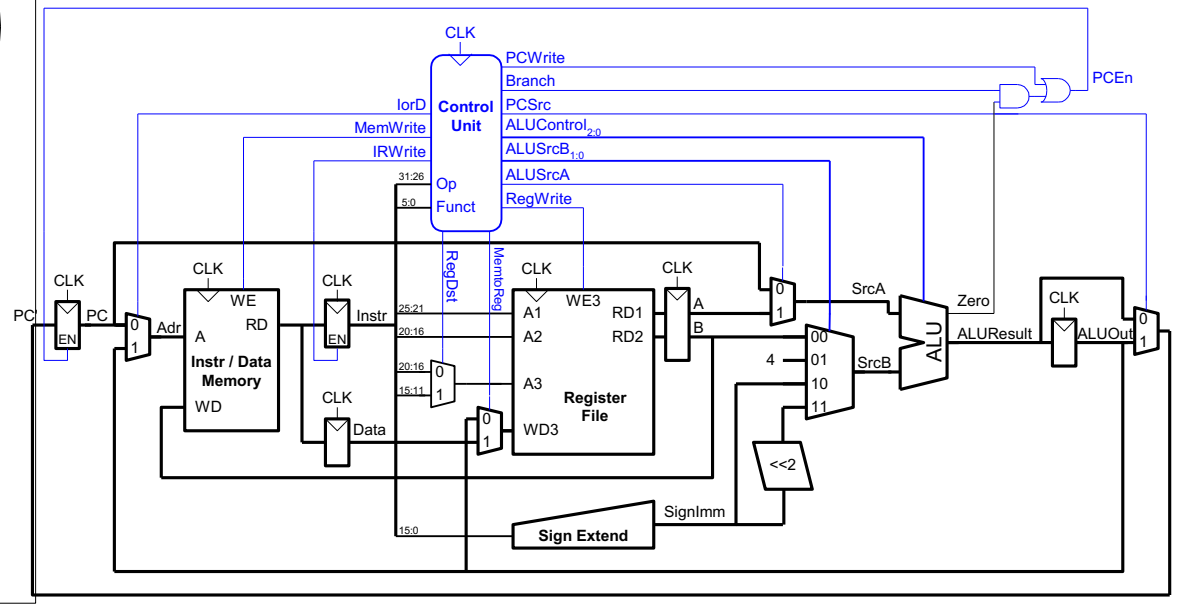
Main Controller FSM: beq



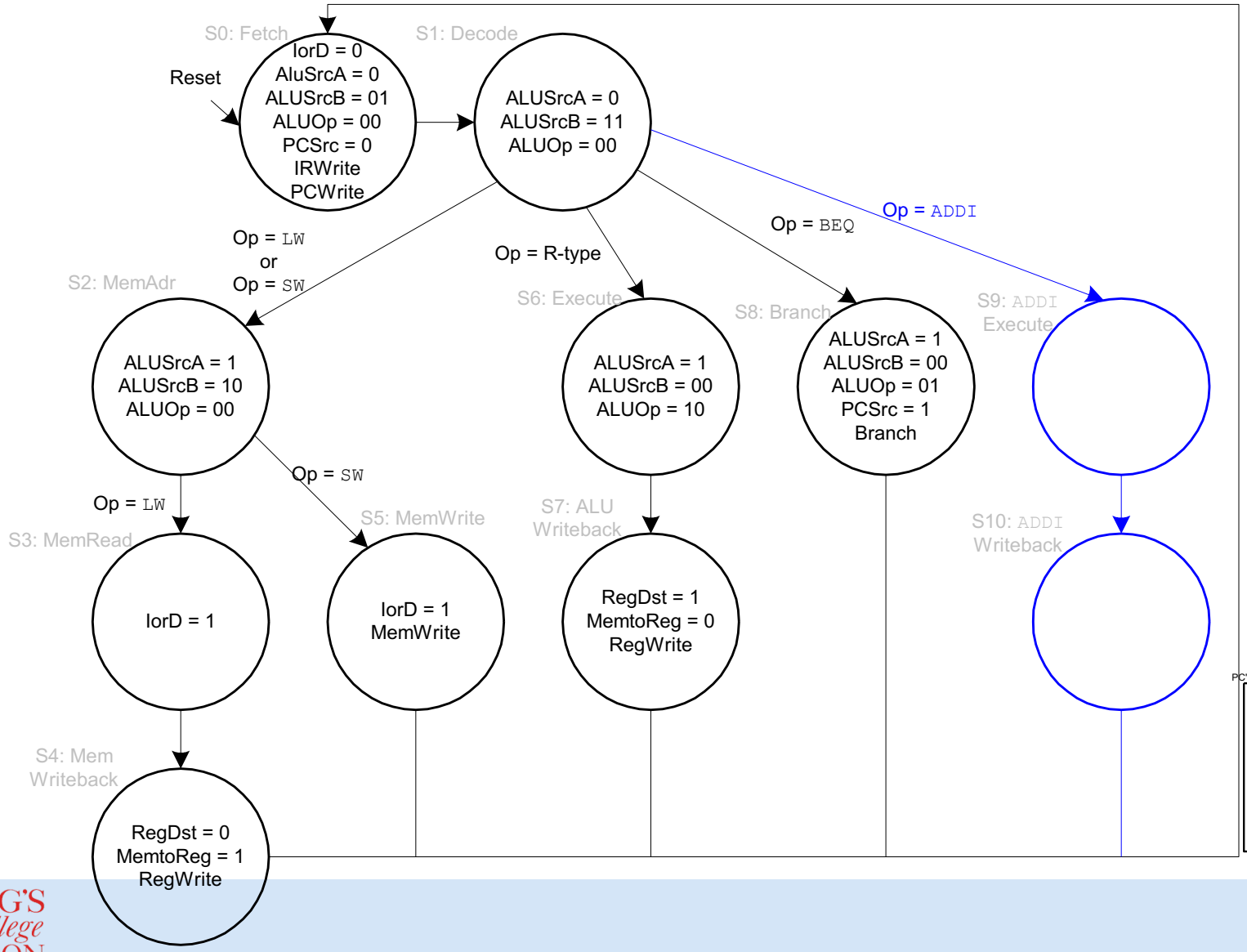


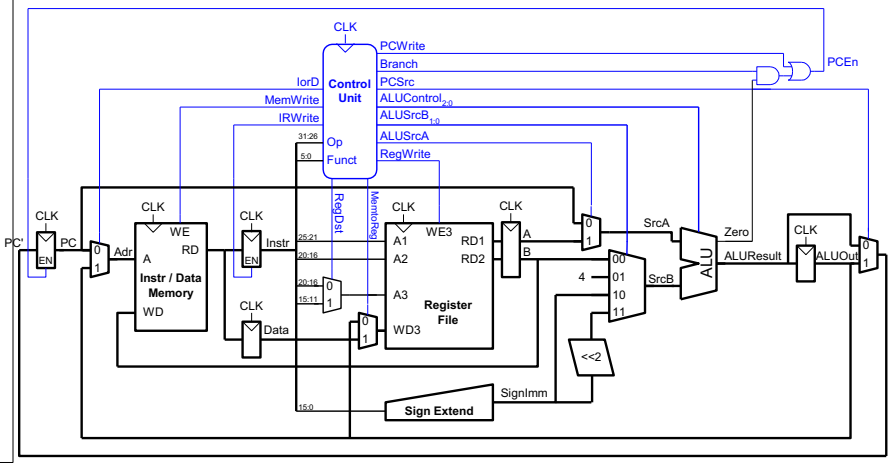
Complete Multi-Cycle Controller FSM



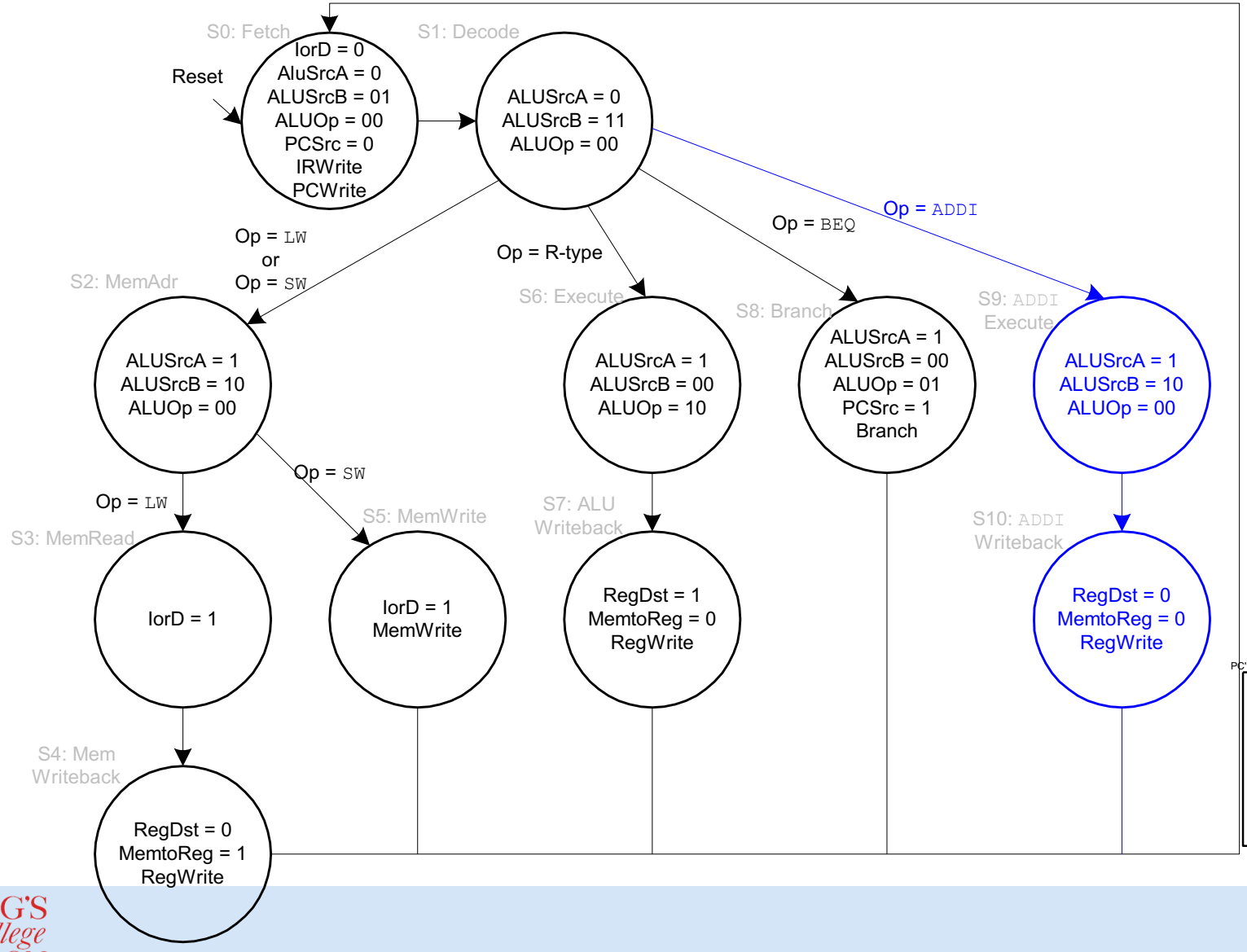


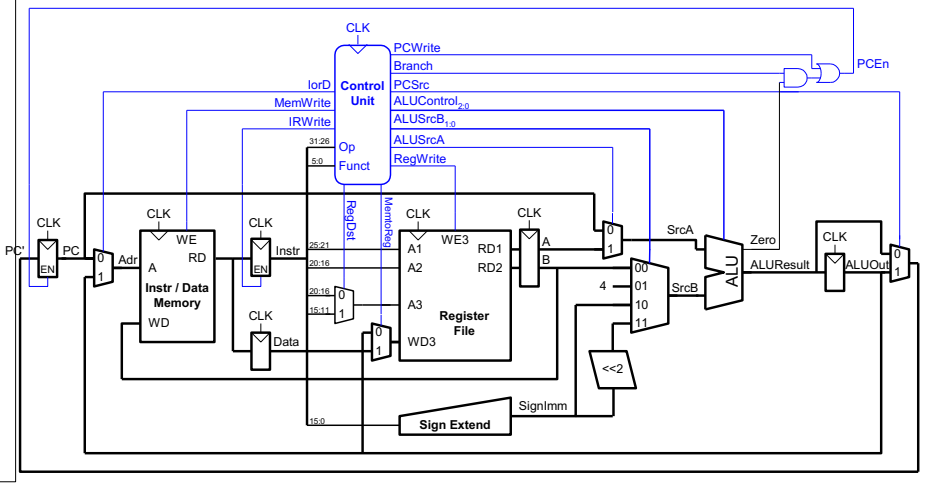
Main Controller FSM: addi



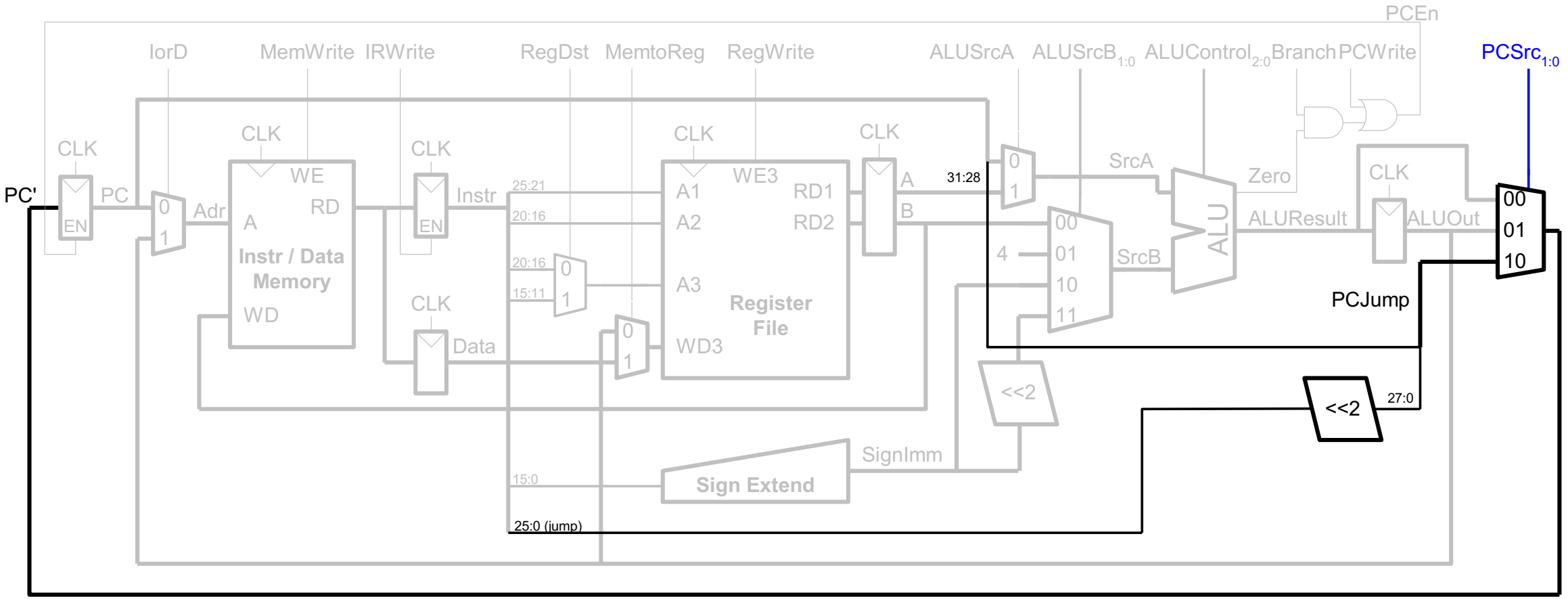


Main Controller FSM: addi

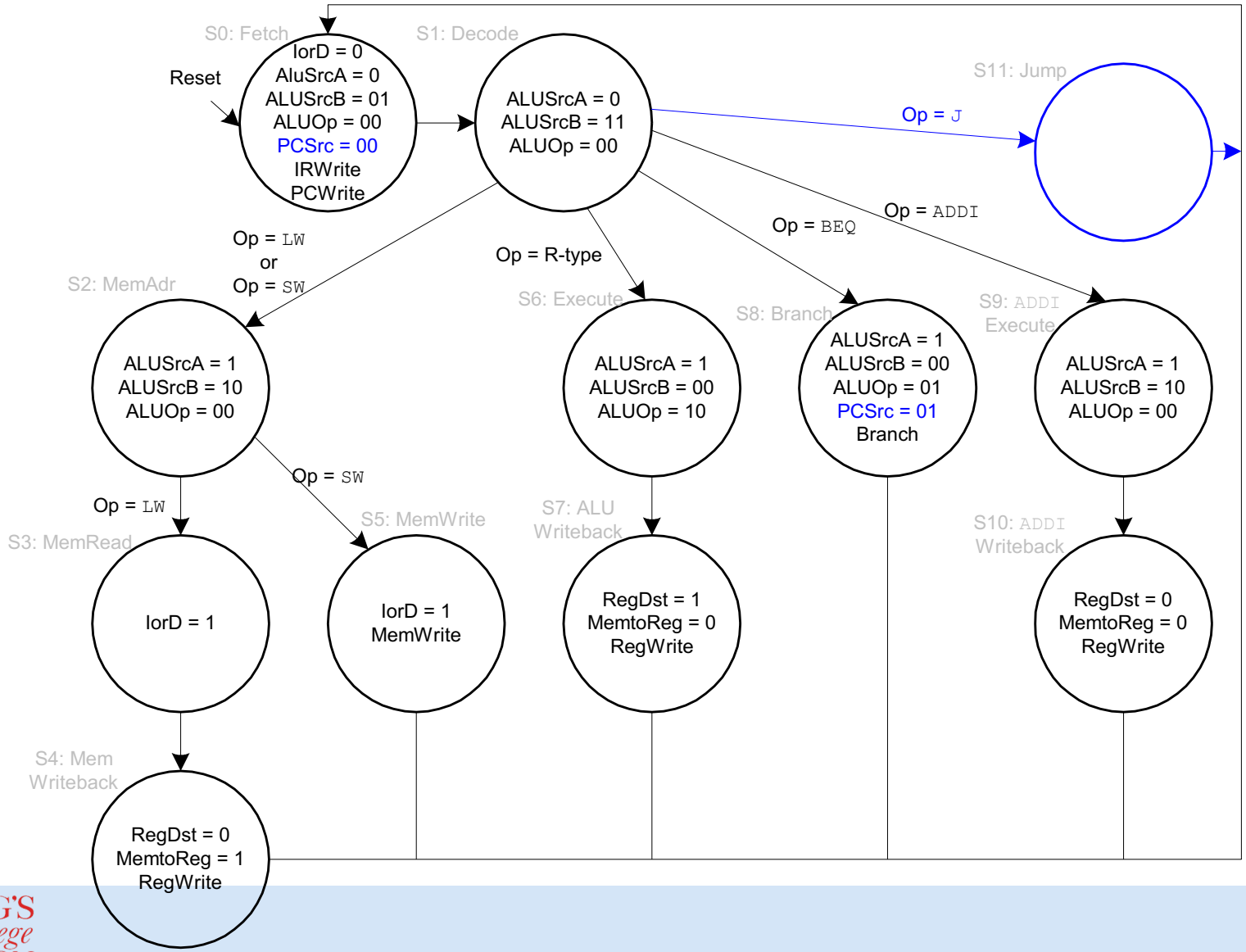


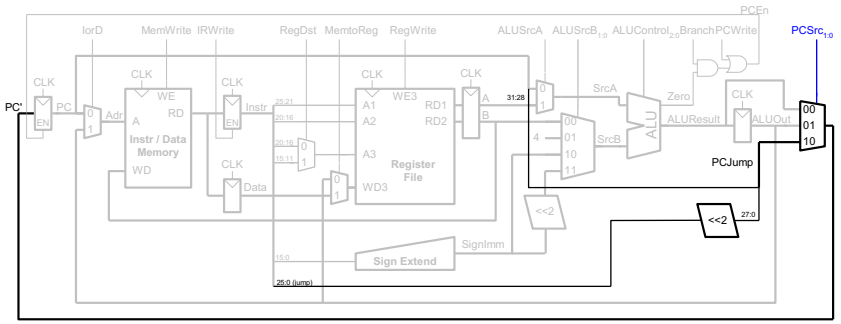


Extended Functionality: j

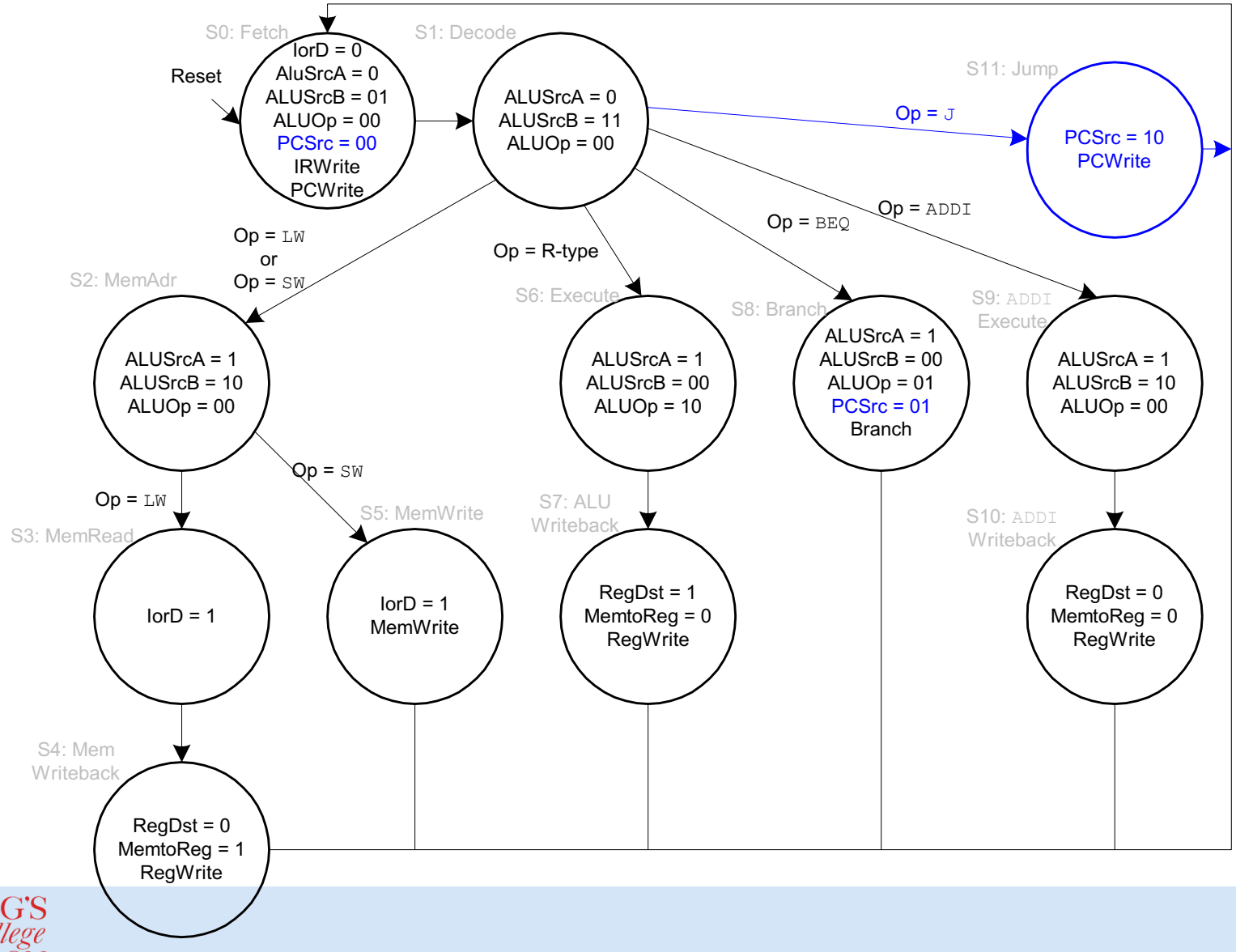


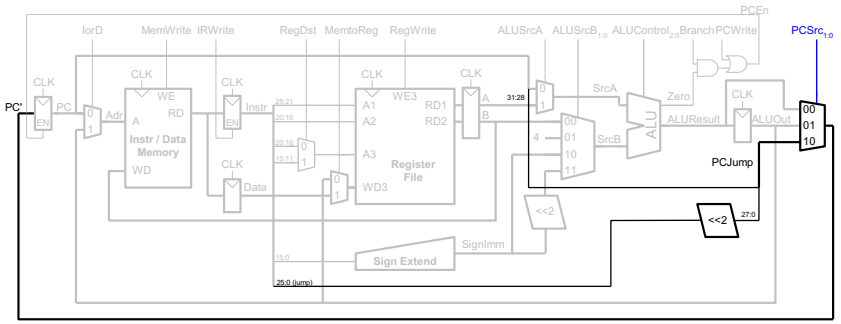
Control FSM: j



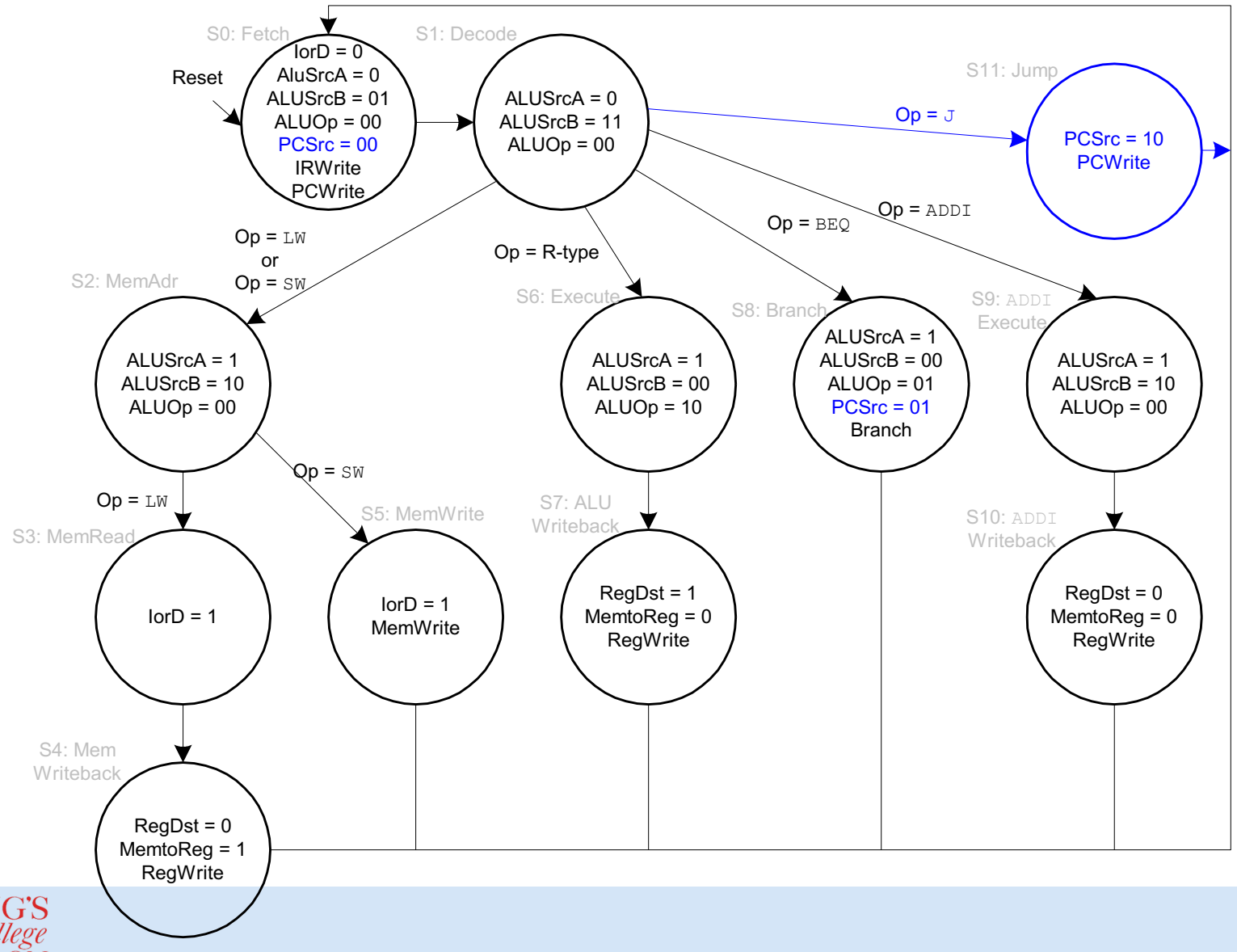


Control FSM: j

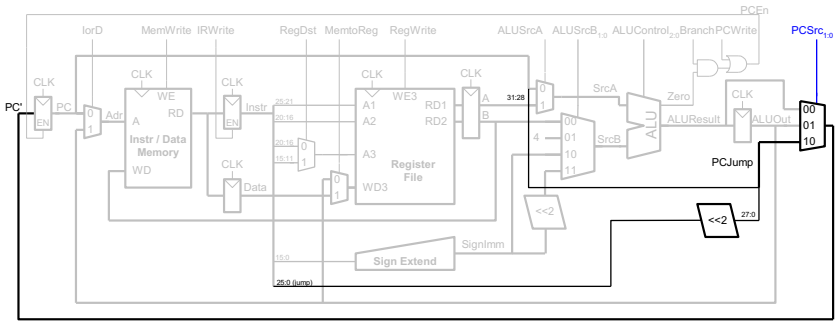




Review: Multi-Cycle MIPS FSM



What does this design assume about memory?



What If Memory Takes > One Cycle?

- Stay in the same "memory access" state until memory returns the data

- "Memory Ready?" bit is an input to the control logic

  that determines the next state

Recall: Full State Machine for LC-3b

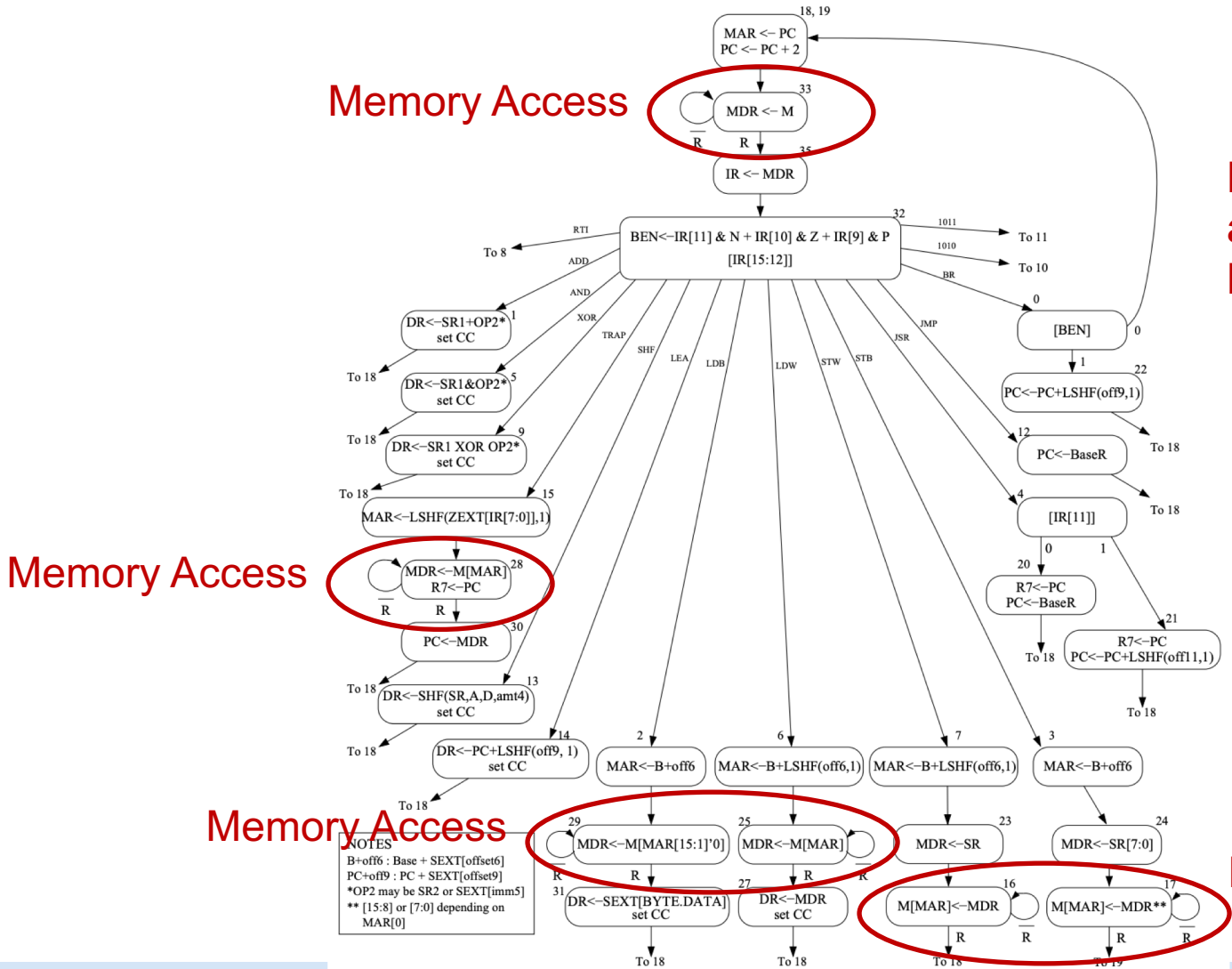


Figure C.2: A state machine for the LC-3b

Full FSM Controlling a Multi-Cycle LC-3b Microarchitecture

Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of the instruction processing cycle
  - "Fetch" logic is idle when an instruction is being "decoded" or "executed"
  - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: More concurrency → Higher instruction throughput (i.e., more "work" completed in one cycle)

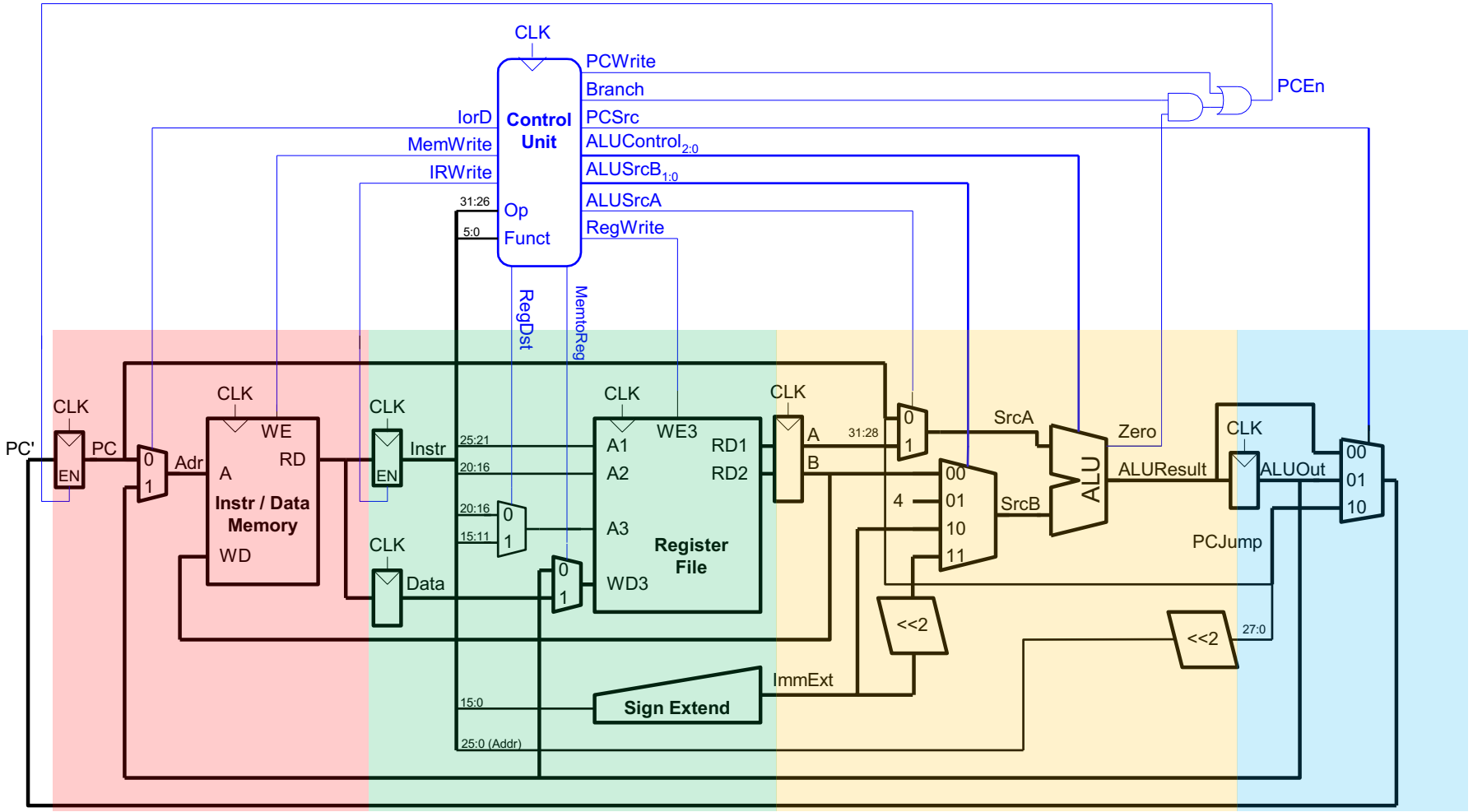
- Idea: When an instruction is using some resources in its processing phase, process **other instructions** on **idle resources** not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Can Have Different Instructions in Different Stages

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

1. Instruction fetch (IF)
2. Instruction decode and
   register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

Can Have Different Instructions in Different Stages



Of course, we need to be more careful than this!

Hardware Design

Pipelining

Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions

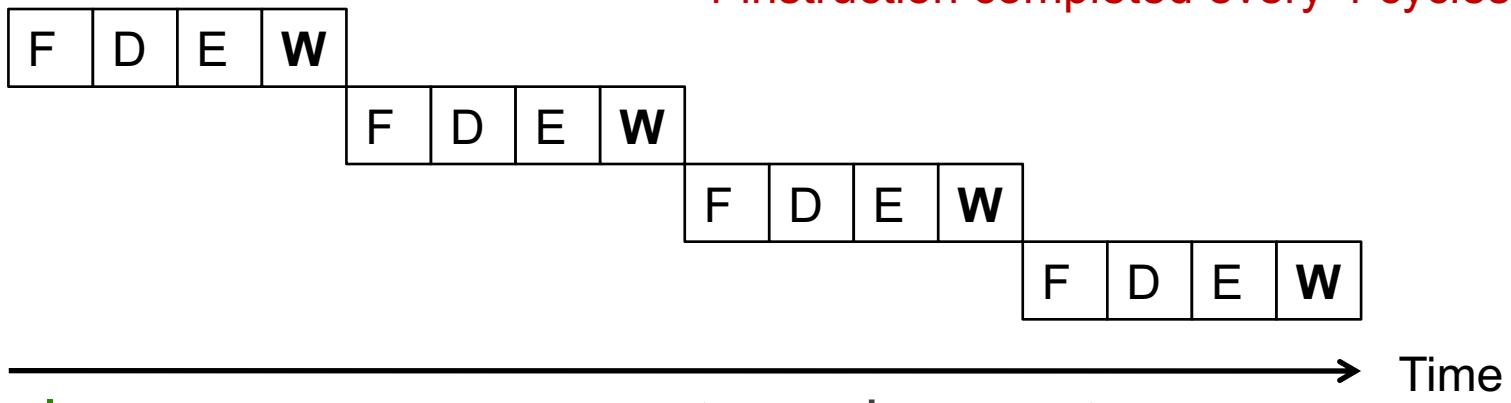
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages

- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this…

Example: Execution of Four Independent ADDs

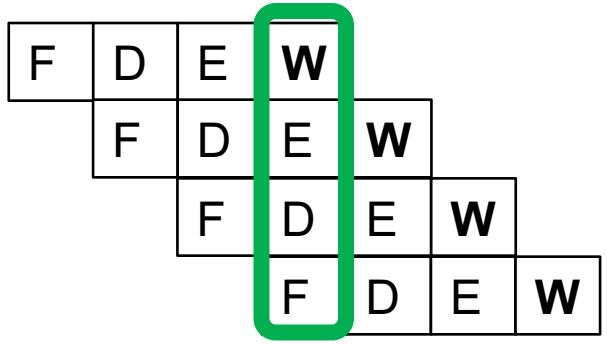
- Multi-cycle: 4 cycles per instruction

1 instruction completed every 4 cycles



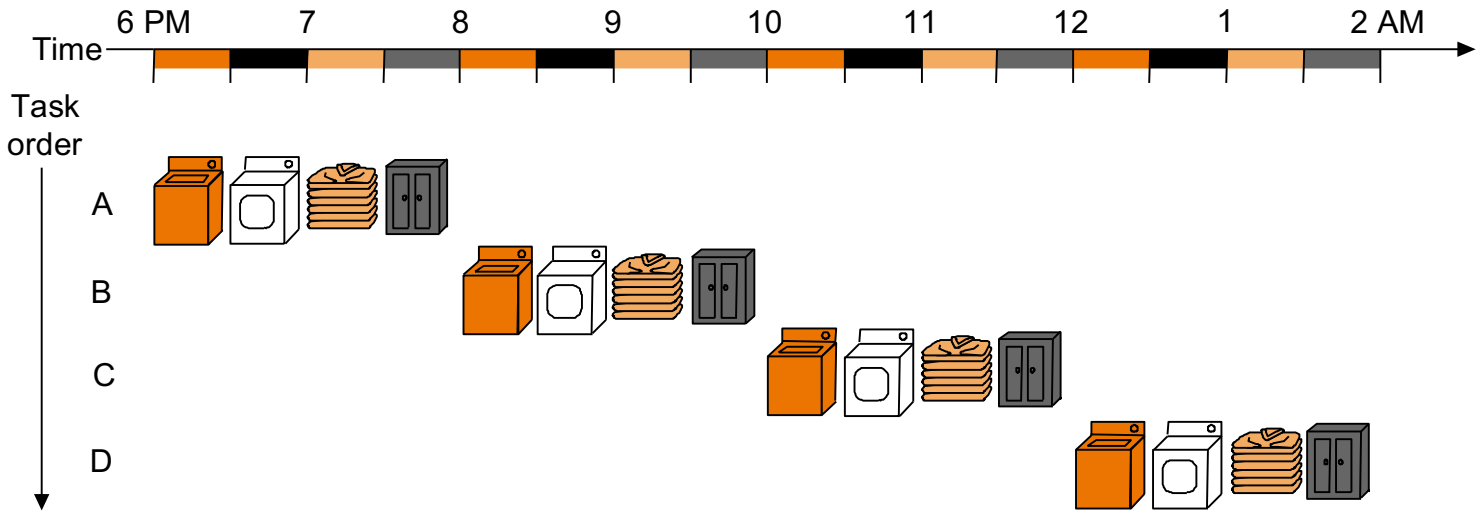
- Pipelined: 4 cycles per 4 instructions (steady state)

1 instruction completed every 1 cycle



Is life always this beautiful?

The Laundry Analogy

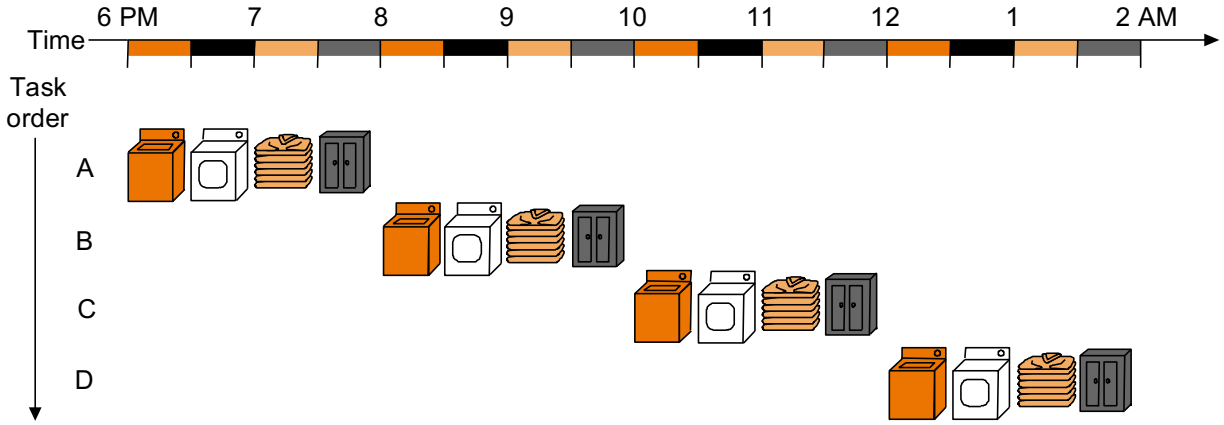


- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, put the clothes away"

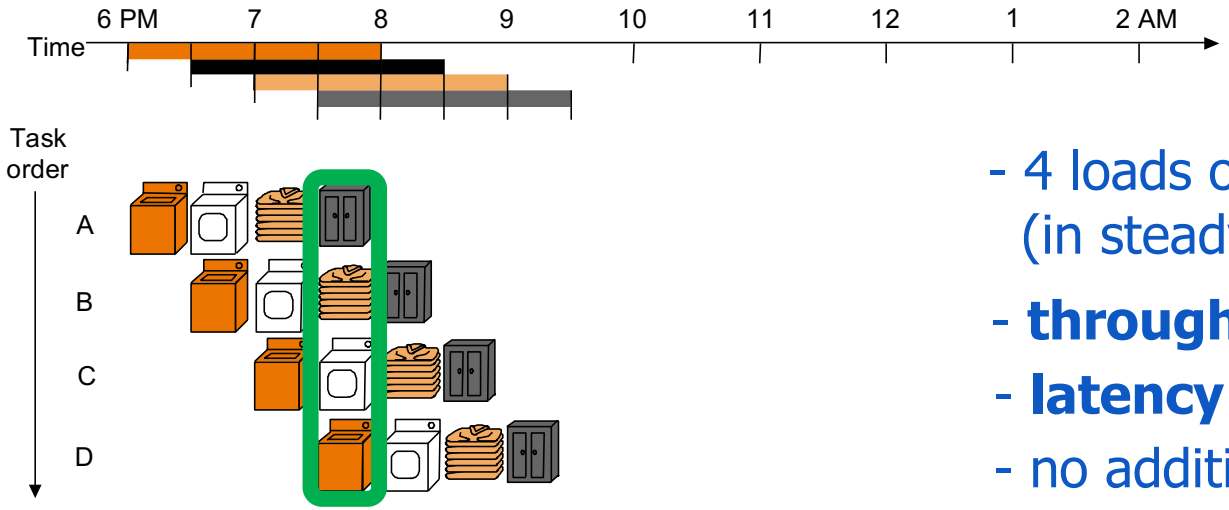
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not require the same resource(s)

Pipelining Multiple Loads of Laundry

1 load every 120 mins



1 load every 30 mins

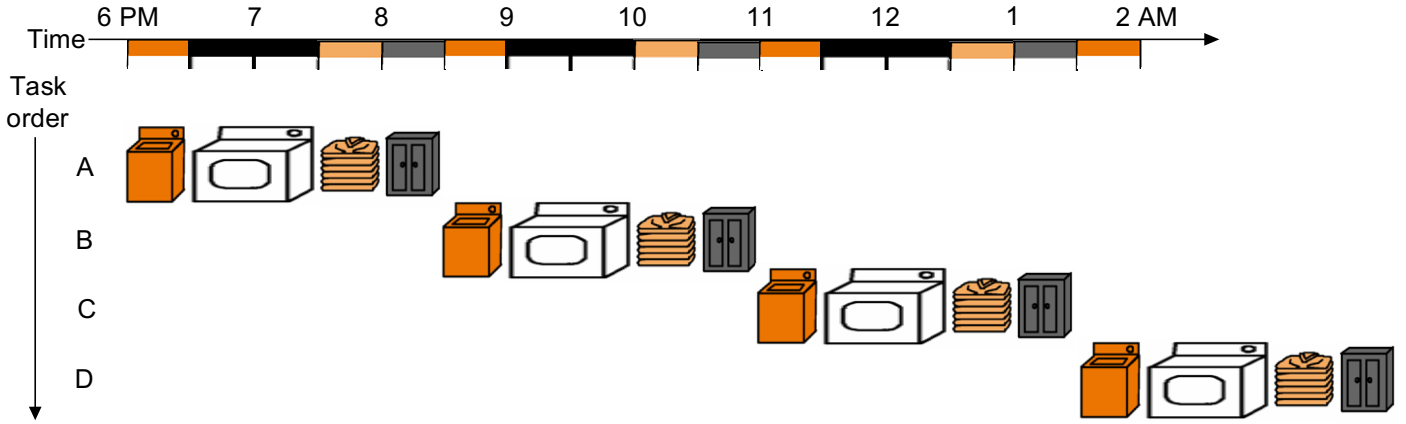


- 4 loads of laundry in parallel (in steady state)
- **throughput** increased by 4x
- **latency** per load is the same
- no additional resources

Pipelining Multiple Loads of Laundry: In Practice

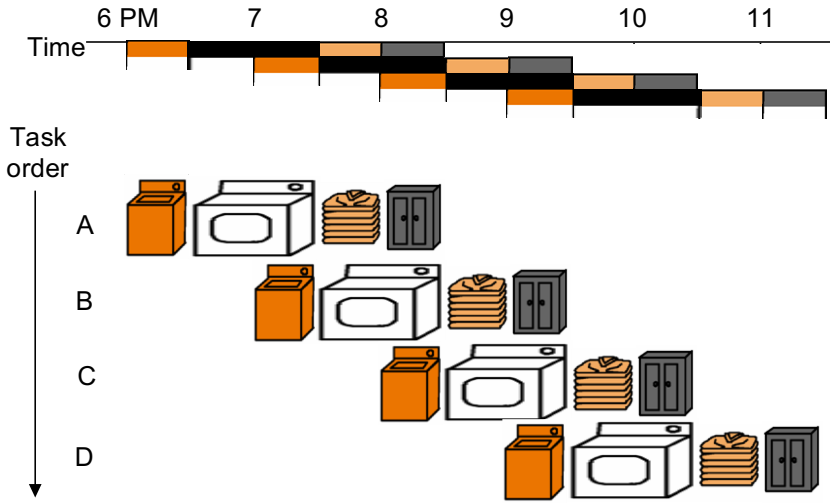
1 load every 150 mins

(slow dryer)



1 load every 60 mins

(slow dryer)

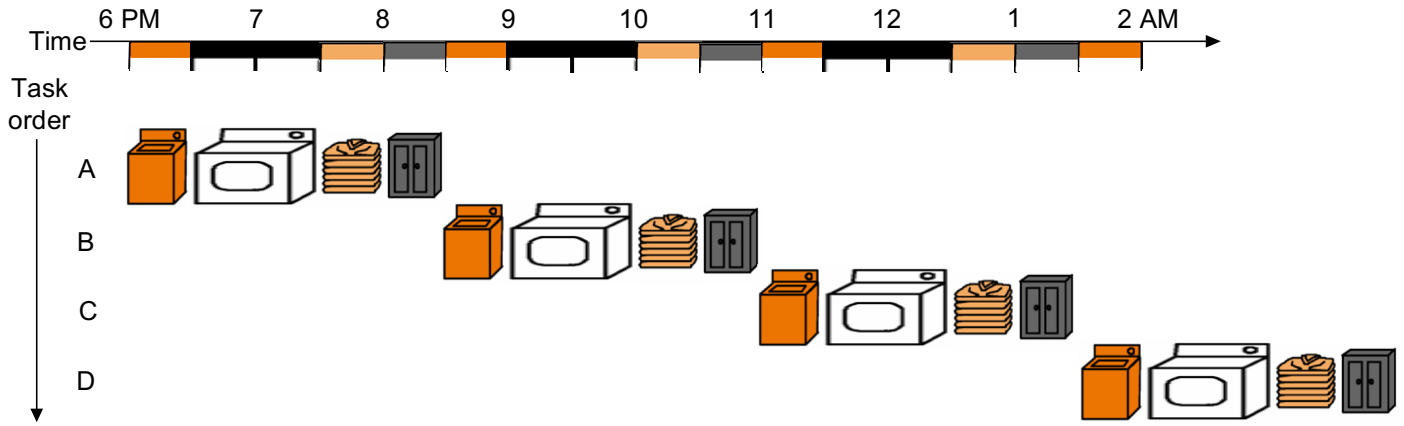


the slowest step (the dryer) decides throughput

Pipelining Multiple Loads of Laundry: In Practice

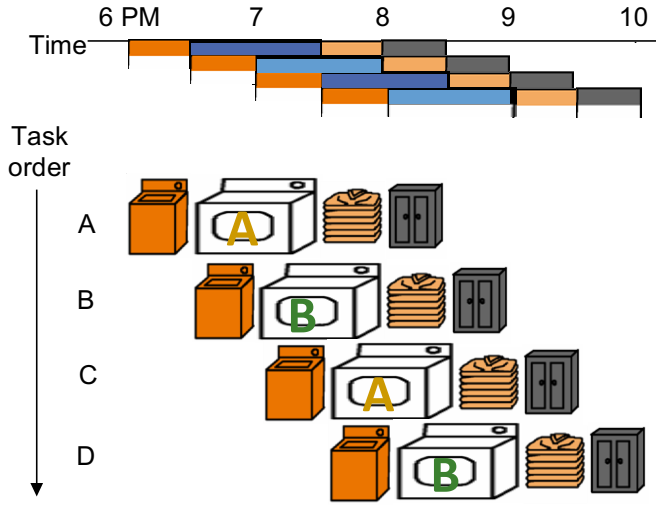
1 load every 150 mins

(slow dryer)



1 load every 30 mins

(2 slow dryers)



throughput restored (1 load per 30-min) using 2 dryers

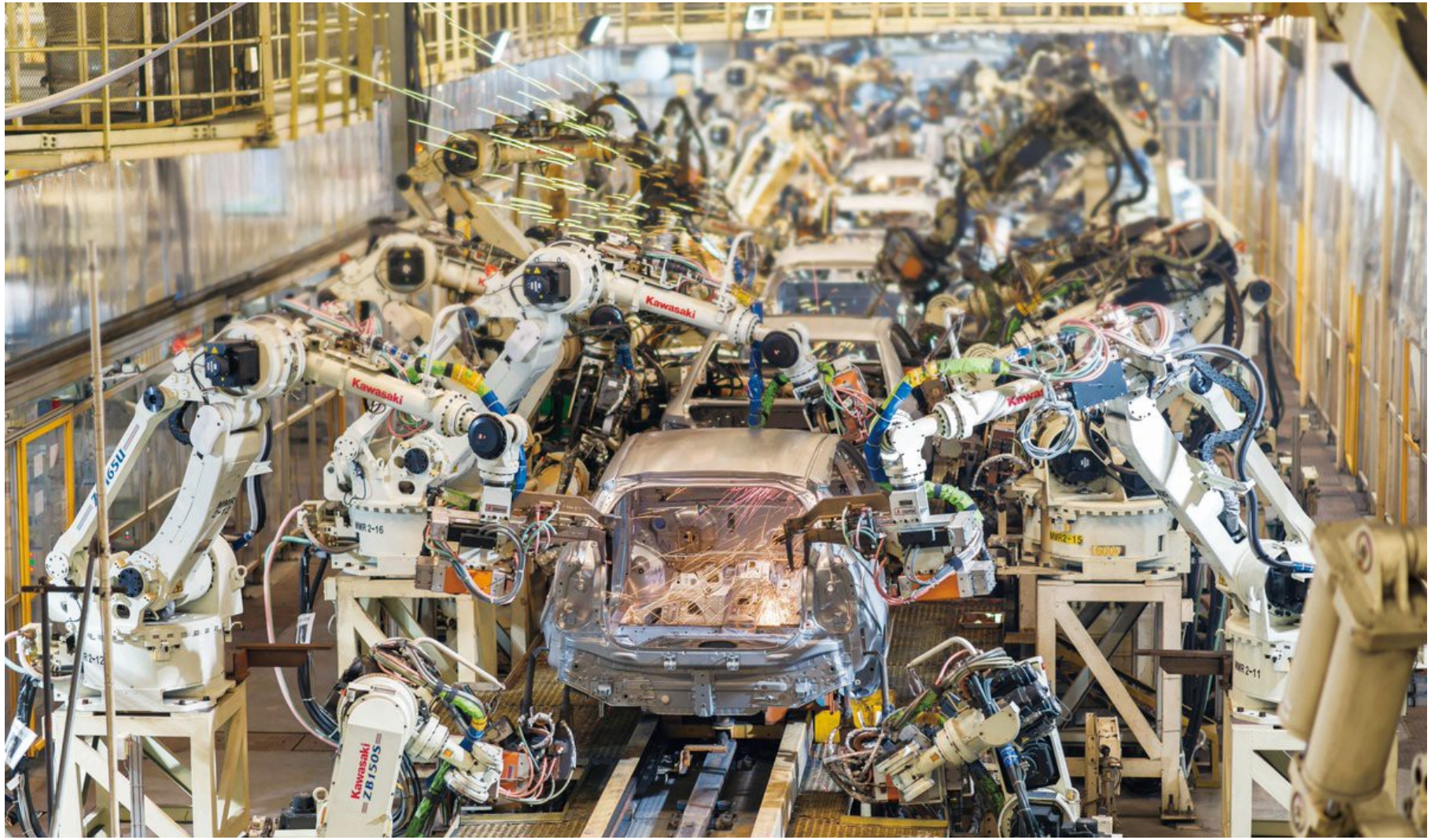
A Real-Life Pipeline: Automobile Assembly

The guinea pig was the T's magneto, a component that supplied ignition energy to the engine before generators became common. A complex and innovative component that was one of the early Model T's technological advantages, Ford's magneto was integrated with the engine's flywheel and involved many pieces. Under the old system, each magneto was assembled by one worker. On average, that worker could assemble 35 of them in a nine-hour shift, or roughly one every 15 minutes.

Ford's transition to moving assembly lines began in April 1913 with the integrated (and complex) flywheel/magneto. With each worker assigned to complete a few specific tasks rather than build the entire unit, Ford reduced magneto assembly time from about 15 minutes to 5, and the required workforce decreased from 29 to 14.

https://www.caranddriver.com/features/a15115930/fords-assembly-line-turns-100-how-it-really-put-the-world-on-wheels-feature/

A Real-Life Pipeline: Automobile Assembly



An Old Pipelined Computer: IBM Stretch

Design	
Manufacturer	IBM
Designer	Gene Amdahl
Release date	May 1961
Units sold	9
Price	US$7,780,000 (equivalent to $67,380,000 in 2020)
Casing	
Weight	70,000 pounds (35 short tons; 32 t)[1]
Power	100 kW[1] @ 110 V
System	
Operating system	MCP
CPU	64-bit processor
Memory	2048 kilobytes (262144 x 64bits)[1]
MIPS	1.2 MIPS

V • T • E

https://en.wikipedia.org/wiki/IBM_7030_Stretch

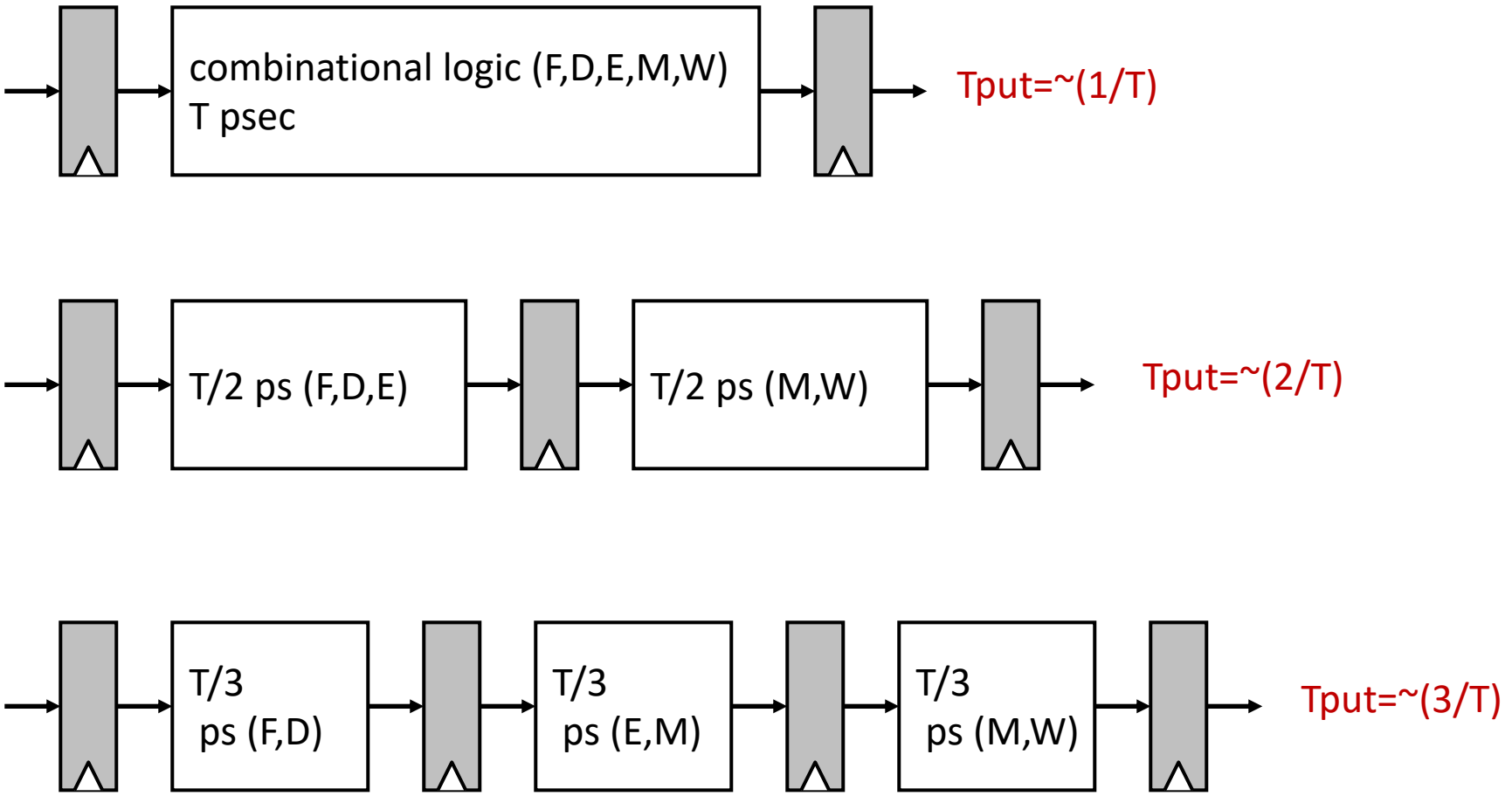
An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing "cycle"?

Ideal Pipelining

Tput = Throughput



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

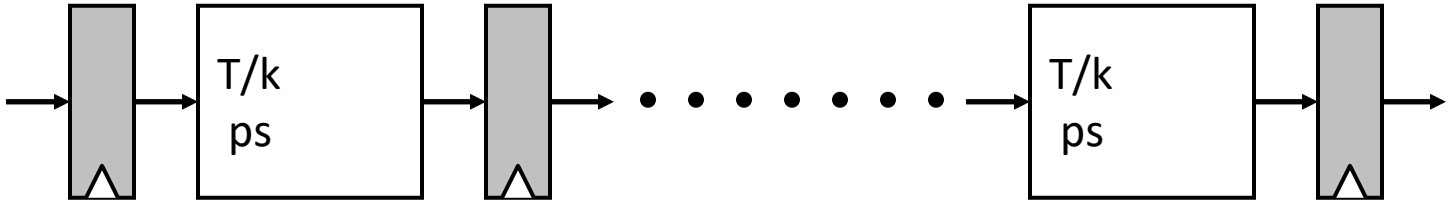
  Tput = 1 / (T+S) where S = register (sequential logic) delay

- k-stage pipelined version

  $Tput_{k\text{-stage}} = 1 / (T/k + S)$

  $Tput_{max} = 1 / (1 \text{ gate delay} + S)$

Register delay reduces throughput (sequencing overhead b/w stages)



This picture assumes "perfect division of work between stages (T/k)"

More Realistic Pipeline: Cost

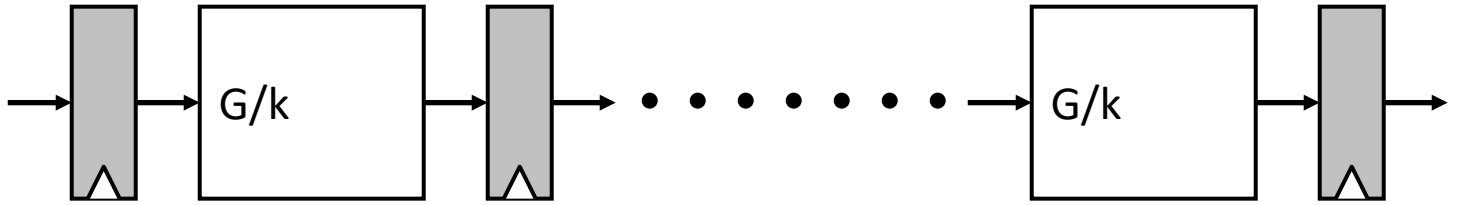
- Nonpipelined version with combinational cost G

  Cost = G+R where R = register cost

- k-stage pipelined version

  $Cost_{k\text{-}stage} = G + Rk$

Registers increase hardware cost



This picture ignores resource and register replication that is likely needed (G/k and Rk)

KING'S College LONDON

Hardware Design

Lecture 5: Multi-Cycle Microarchitecture and Pipelining

Dr. Haiyu Mao

26.02.2026