

A Brief Introduction to Computer Architecture



BEng in Computer and Electronic Engineering

School of Electronic Engineering

Dublin City University

2024

Xiaojun Wang, xiaojun.wang@dcu.ie

Basic Computer Architecture

- A basic computer consists of
 - Input/Output,
 - Central Processing Unit (CPU) and
 - Memory

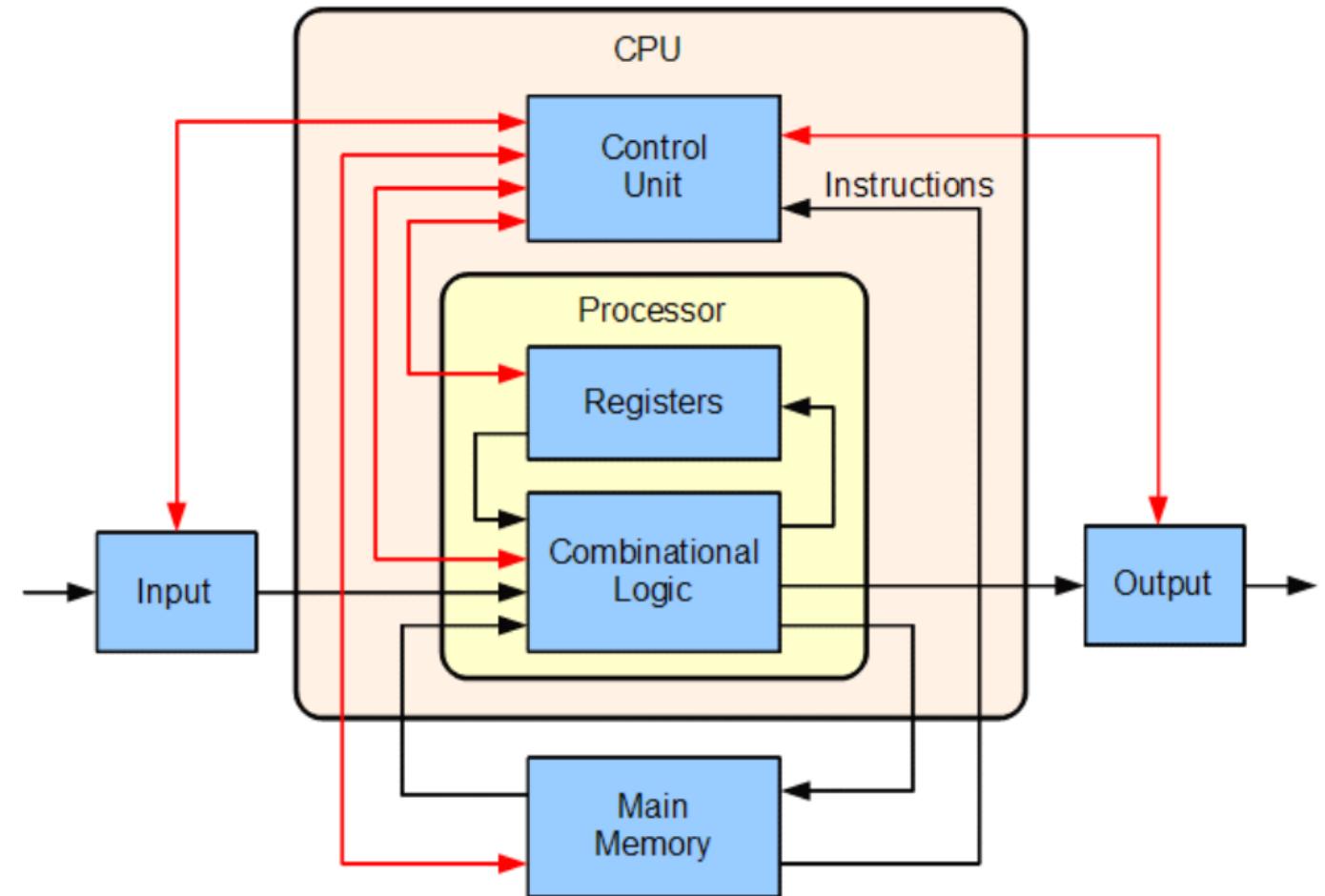


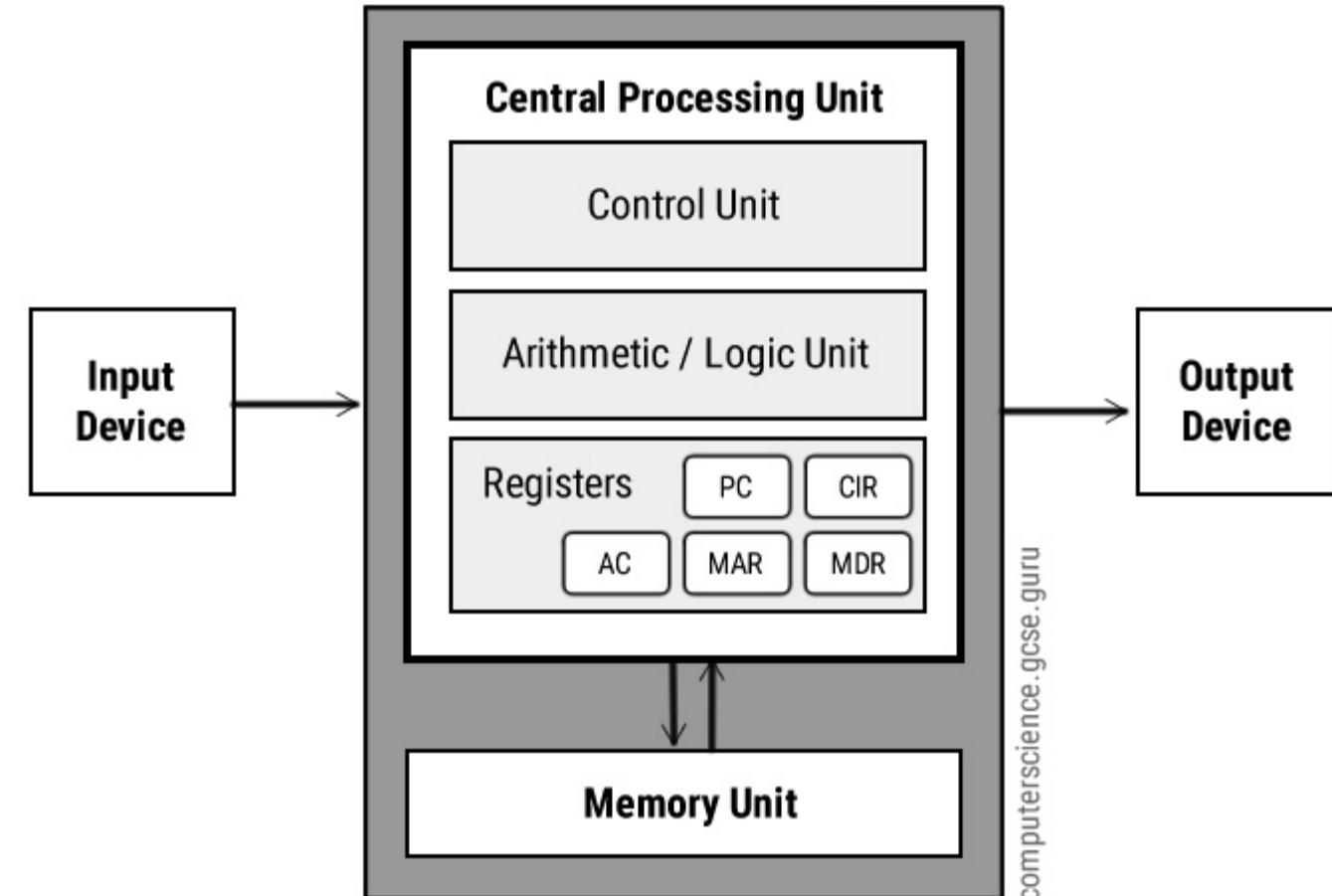
Diagram from Wikipedia

Black lines indicate data flow
Red lines indicate the control flow
Arrows indicate the flow direction

Von Neumann Architecture

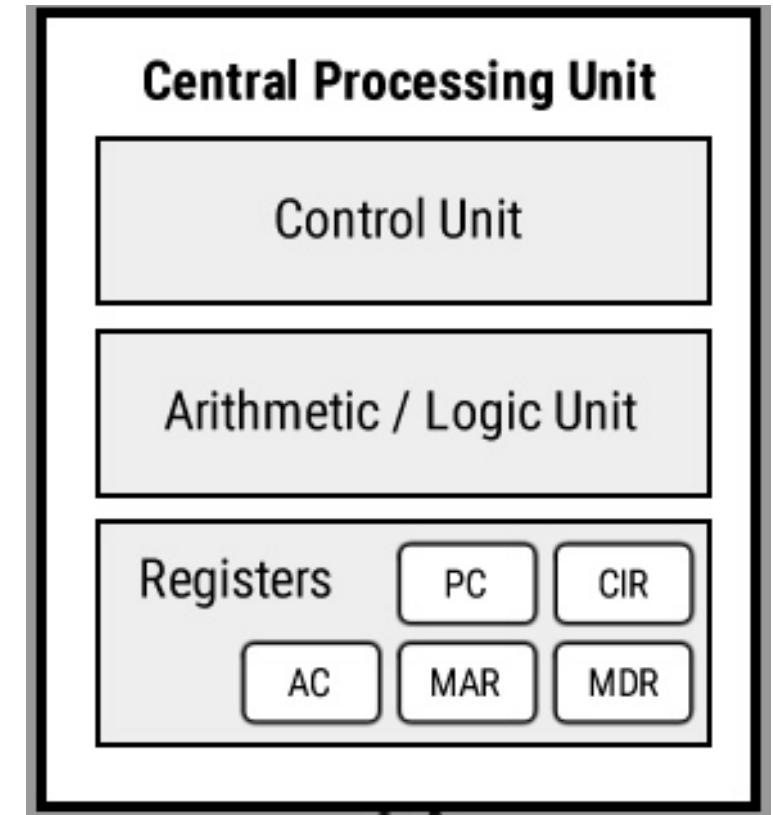
(also known as Princeton Architecture)

- John von Neumann published in 1945.
- Consists of
 - A Central Processing Unit (**CPU**)
 - Control Unit
 - An Arithmetic/Logic Unit (ALU))
 - Registers (PC, CIR, AC, MAR MDR)
 - **Memory Unit**
 - **Input/Output** Devices



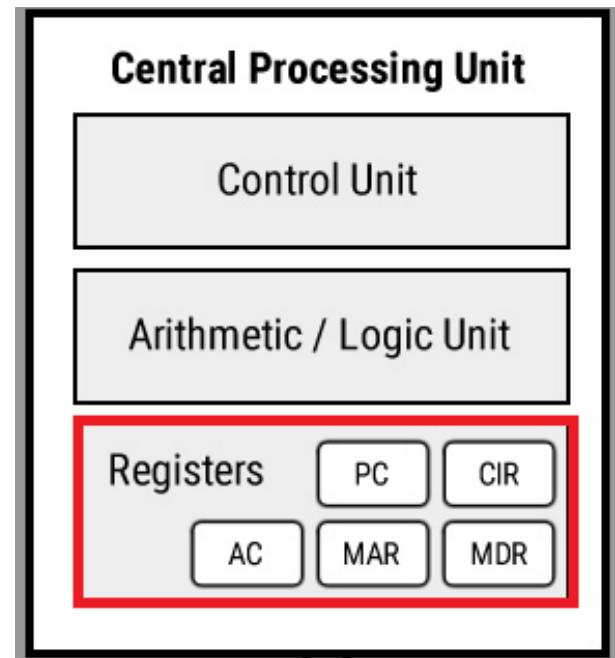
Central Processing Unit (CPU)

- CPU is the electronic circuit for executing the **instructions** of a computer program
- Sometimes referred to as *microprocessor* or *processor*
- CPU contains
 - Arithmetic/Logic Unit (ALU)
 - Control Unit
 - A set of Registers
- Different computer architectures have different CPUs executing different set of instructions



CPU Registers

- Registers are high speed storage areas in CPU
- All data must be stored in a register before it can be processed
- Common Registers examples

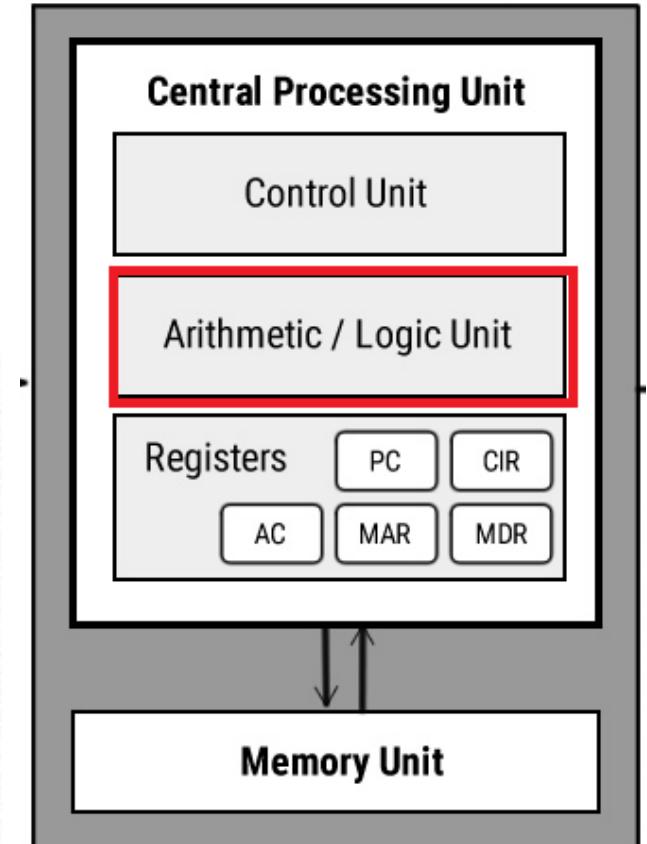
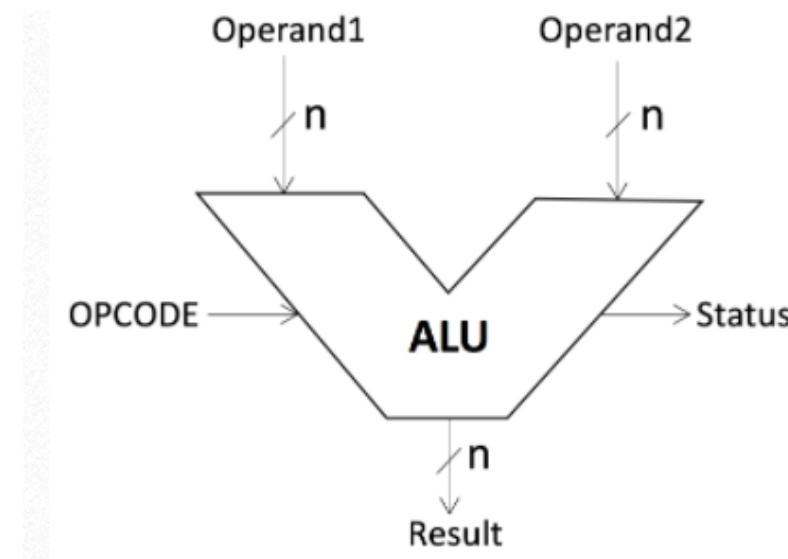


mnemonic	Register meaning	Typical usage
MAR	Memory Address Register	Holds the memory location of data that needs to be accessed
MDR	Memory Data Register	Holds data that is being transferred to or from memory
AC	Accumulator	Where intermediate arithmetic and logic results are stored
PC	Program Counter	Contains the address of the next instruction to be executed
CIR	Current Instruction Register	Contains the current instruction during processing

- Different **Instruction Set Architectures (ISA)** have different set of CPU registers

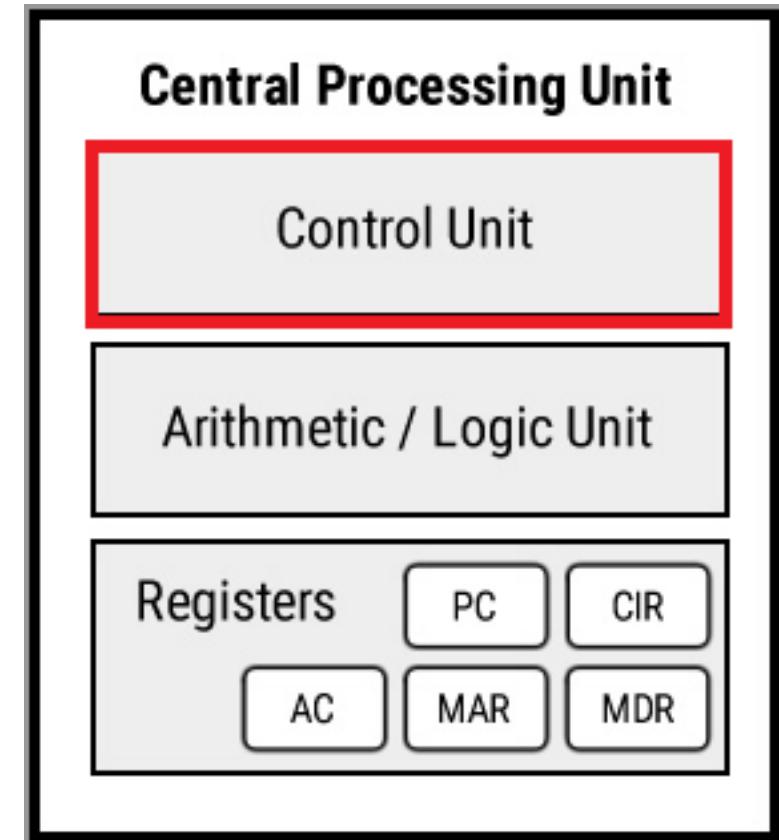
Arithmetic and Logic Unit (ALU)

- A combinational logic circuit performing different arithmetic and bitwise logic operations on integer binary numbers
- Inputs
 - Two data inputs (operands)
 - One control input (opcode)
- Outputs
 - Operation result
 - Status (flags):
 - Zero (result is zero)
 - Negative (result is negative)
 - Carry-out/borrow
 - Overflow
 - Parity



Control Unit (CU)

- The control unit controls the operation of the computer's ALU, memory and input/output devices
- Tells the components how to respond to the program instructions it has just *fetched* (read) and *decoded* (interpreted) from the memory unit
- The control unit also provides the timing and control signals to other computer components to *execute* the instruction
- The **fetch–decode–execute** cycle



Buses

- Buses are the means by which data is transmitted from one part of a computer to another
- Connect all major internal components to the CPU and memory
- CPU system buses consist of *control bus, data bus and address bus*

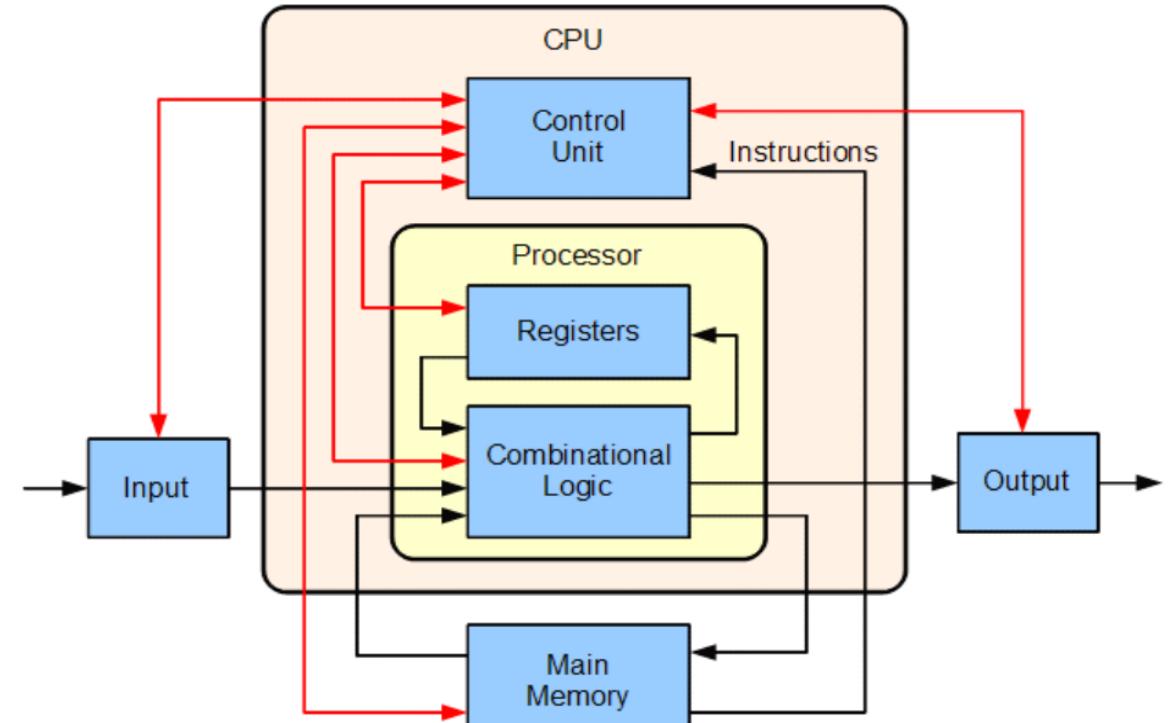


Diagram from Wikipedia

Black lines indicate data flow
Red lines indicate the control flow
Arrows indicate the flow direction

Bus name	Purpose
Address Bus	Carries the memory address of data (not the data itself) between the processor and memory
Data Bus	Carries data between the processor, the memory unit and the input/output devices
Control Bus	Carries the control signals/commands from the CPU (and status signals from other devices) to control and coordinate all the activities within the computer

Memory Hierarchy

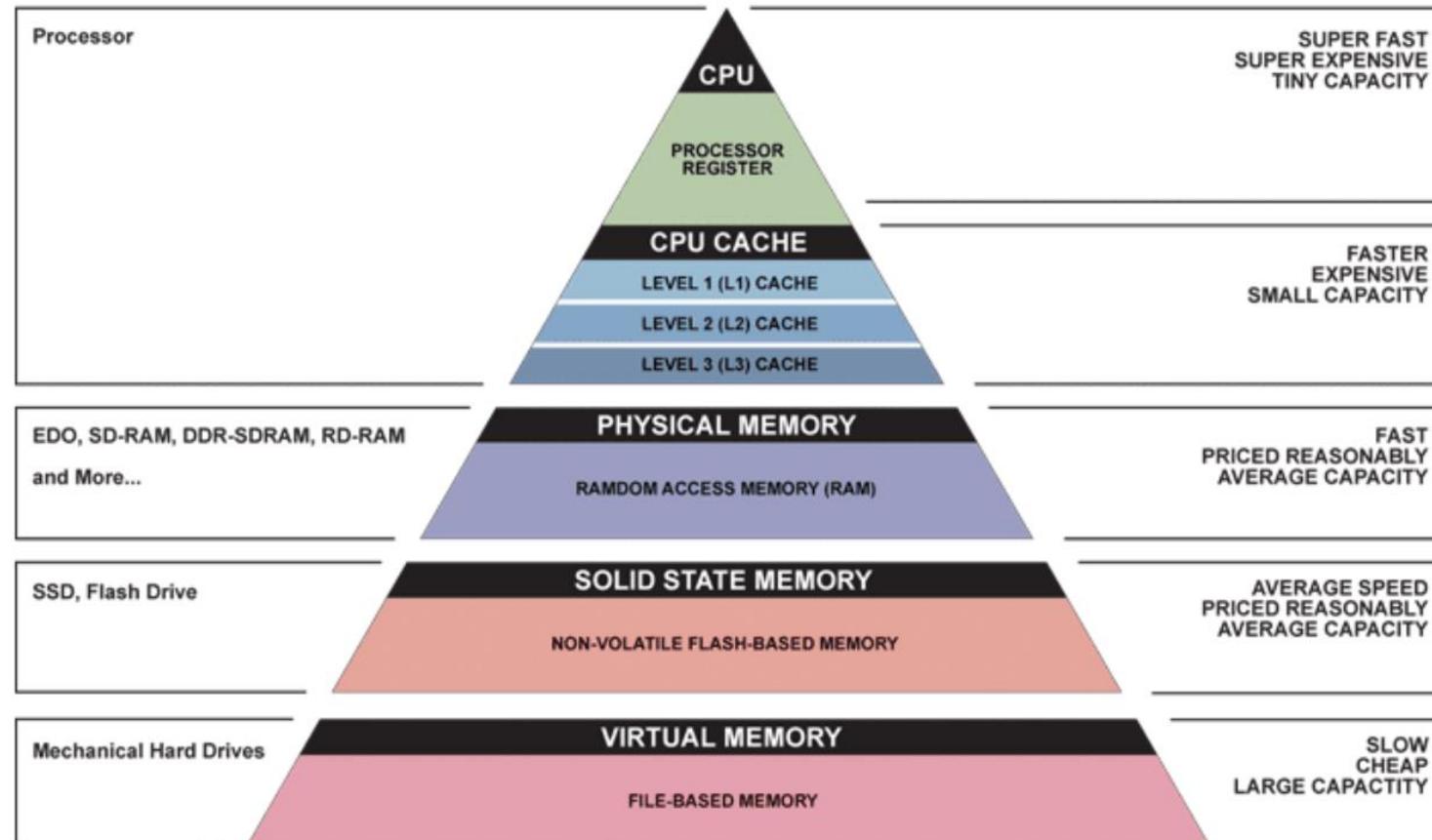


Fig. from <https://sites.google.com/site/cachememory2011/memory-hierarchy>

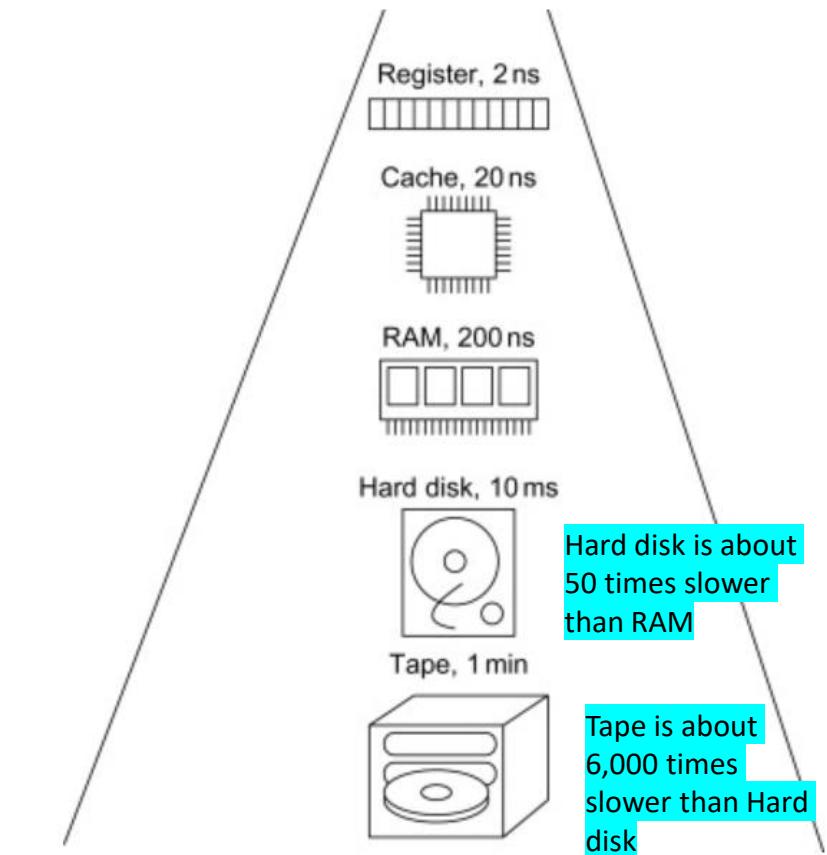
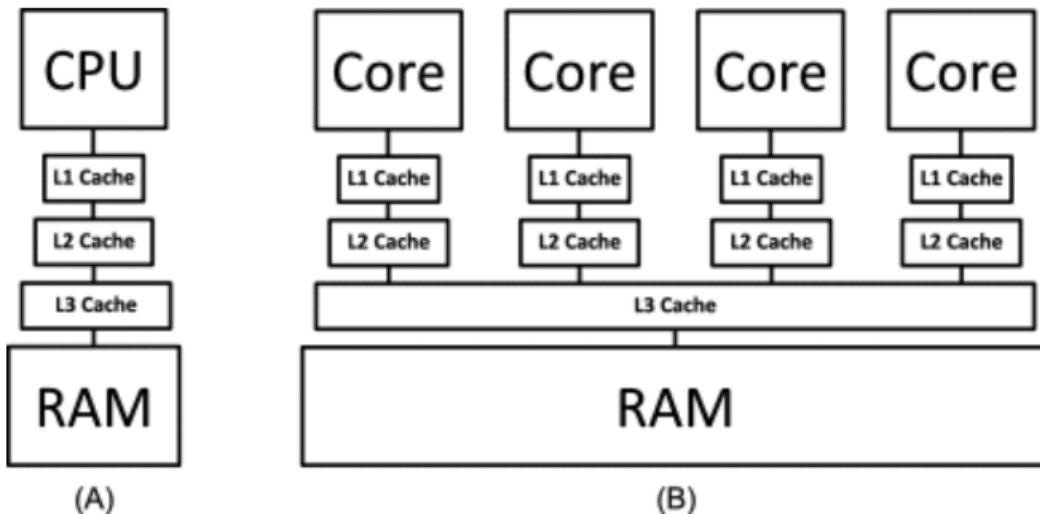
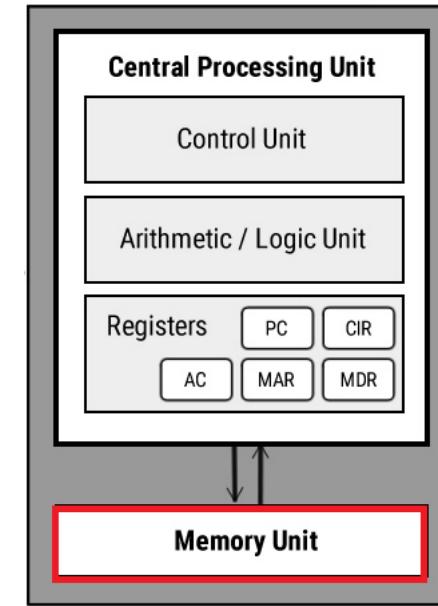


Fig. from Stefan Edelkamp, Stefan Schrödl, in [Heuristic Search](#), 2012

Memory Unit

- Cache, up to three levels
- RAM (*Random Access Memory*), sometimes referred to as *Primary* or *main memory*
- Hard drive is called *secondary memory*
 - Loading data from permanent memory (hard drive) into faster and directly accessible temporary memory (RAM), allows the CPU to operate much quicker



(A) Cache architecture (B) Multicore architecture with shared L3 cache
Fig. from "Philipp Neumann Prof, Dr, Julian Kunkel Dr, in Knowledge Discovery in Big Data from Astronomy and Earth Observation, 2020"

Cache Memory

- Cache memory temporarily stores frequently used instructions and data for quicker CPU access.
- Cache memory is made of *high-speed static RAM (SRAM)*.
- Main memory is made of *dynamic RAM (DRAM)*, slower and cheaper than SRAM.
- Cache memory is fast and expensive, additionally categorised as “levels” that describe its closeness and accessibility to the CPU.
 - L1 cache, or primary cache, is extremely fast but relatively small, usually embedded in the processor chip as CPU cache.
 - L2 cache, or secondary cache, is often more capacious than L1 cache.
 - L3 cache is specialised memory developed to improve the performance of L1 and L2. L1 and L2 are significantly faster than L3; L3 is usually double the speed of DRAM main memory)

Von Neumann Architecture *security issues*

- There are drawbacks to the Von Neumann design, especially regarding security, only conceived as a problem in the 1980s.
- Program modifications, either by accident or design, can be harmful.
- Since the processor executes the **memory word** the PC (program counter) points to, there is **no distinction between instructions and data**.
- This is the design flaw that attackers use to perform code injection attacks.

Code Injection Attack

- An attacker introduces malicious code into a vulnerable computer program, causing it to deviate from its intended execution path.
- It can have catastrophic consequences, including data loss, corruption, denial of access, and even complete host takeover.
- Commonly used in system hacking or cracking to gain unauthorised access, escalate privileges, or install malware on a server.

Harvard Architecture

- Separate Instruction memory and Data memory allow simultaneous access to instruction and data.
- In von Neumann architecture, program instructions and data share the same memory.
- **Modified Harvard architecture** allows the contents of the instruction memory to be accessed as data.
 - Separate caches for data and instruction
- *Real-world computer designs are based on modified Harvard architecture*, such as RISC-V, ARM, MIPS, SPARC, PowerPC, and x86 processors.

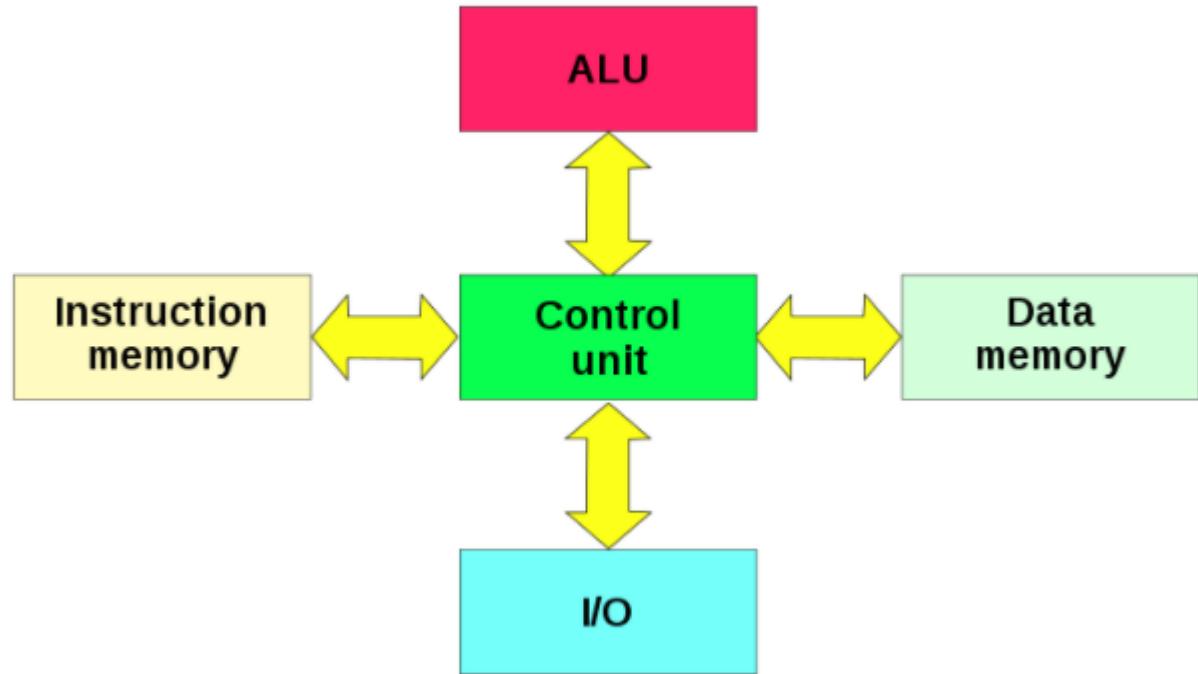
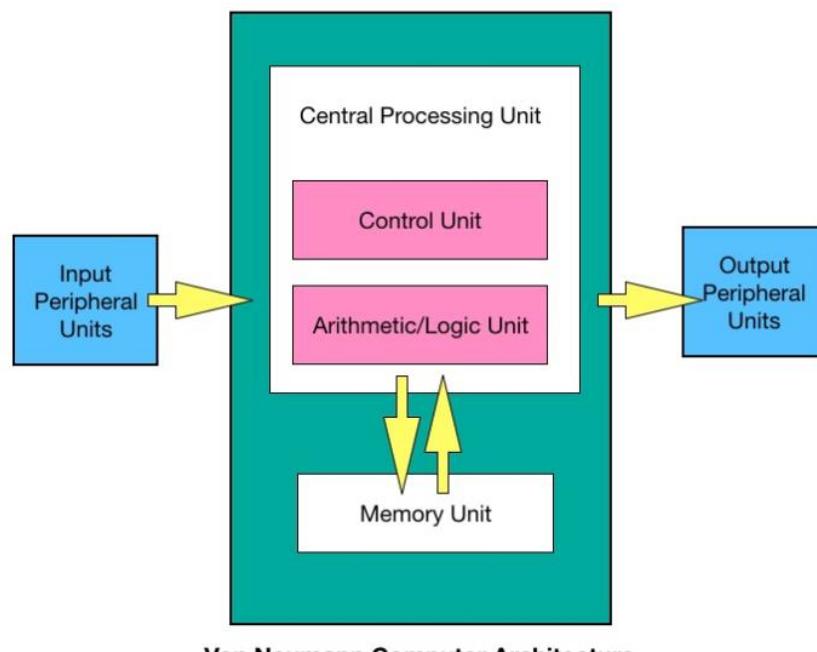


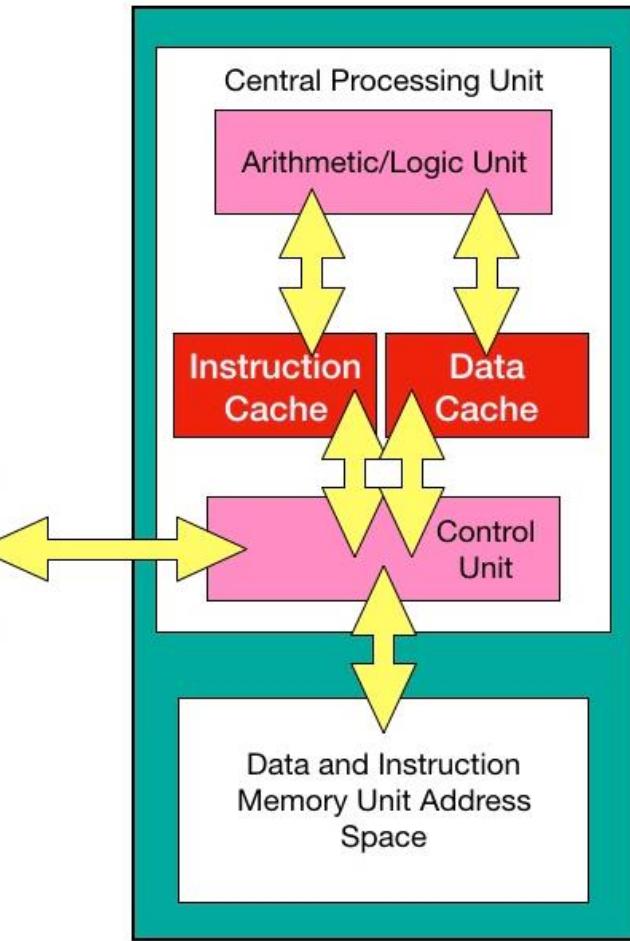
Fig. from Wikipedia

Modified Harvard Architecture

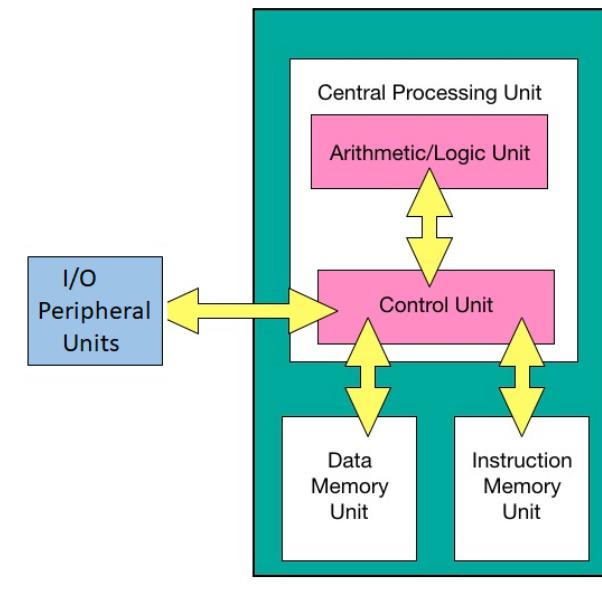
Modified Harvard Architecture is used by x86, ARM and PowerPC ISA, MIPS, RISC-V Processors



I/O
Peripheral
Units



Modified Harvard Computer Architecture



Harvard Computer Architecture

Instruction Set Architecture (ISA)

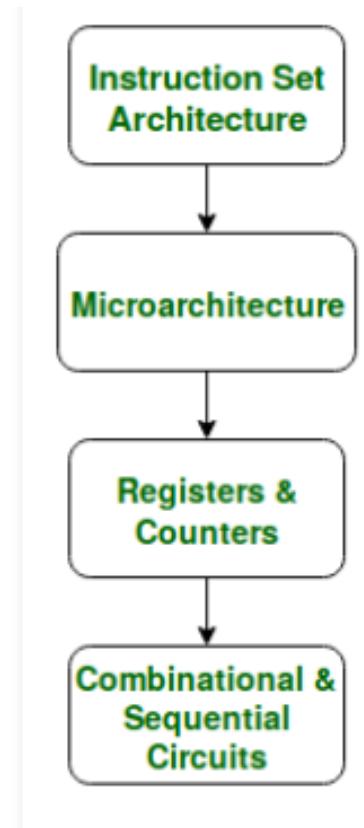
- An ISA defines the number and **types of instructions** to be supported by the processor; for example, RISC-V ISA has 47 instructions grouped into six types.
 1. R-type: register-register.
 2. I-type: short immediates and loads.
 3. S-type: stores.
 4. B-type: conditional branches.
 5. U-type: long immediates.
 6. J-type: unconditional jumps.
- An ISA defines the **maximum length** of each type of instruction.
 - Since the RISC-V is a 32-bit ISA, each instruction must be accommodated within 32 bits.
- The ISA defines the **Instruction Format** of each type of instruction.
- The **Instruction Format** determines how the entire instruction is encoded within 32-bits

Instruction Set Architecture (ISA)

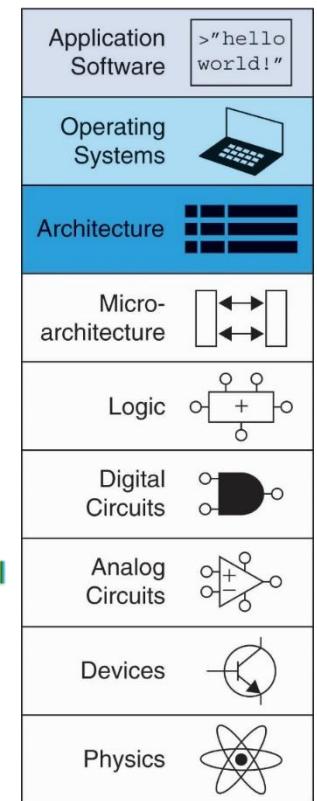
- The **ISA** is responsible for defining the set of instructions to be supported by the processor.
 - The ISA is *not* concerned with the implementation-specific details of a computer.
 - The same ISA can be implemented with different microarchitectures
- The Branch of **Computer Architecture** is more inclined towards the Analysis and Design of *Instruction Set Architecture*.
- For Example,
 - Intel developed the *x86* architecture,
 - ARM developed the *ARM* architecture,
 - AMD developed the *AMD64* architecture.
 - The **RISC-V** is an open-source ISA developed by UC Berkeley and widely accepted by industry.
 - We will look at RISC-V ISA in more detail.

Microarchitecture Level

- The **Microarchitecture** level lies just below the **ISA** level
- It is concerned with implementing the basic operations to be supported by the Computer as defined by the **ISA**.
- **The same ISA** may be implemented with **different microarchitectures**.
 - E.g. The AMD Athlon and Core 2 Duo processors are based on the *same ISA* but have *different microarchitectures* with different performance and efficiencies.



An ISA is the interface between a software program and the hardware processor.



(Fig. from Harris' book)

Microarchitecture Level

- The Microarchitecture is more concerned with the lower-level implementation of how the instructions will be executed on hardware.
- deals with concepts like
 - Instruction Pipelining,
 - Branch Prediction,
 - Out-of-Order Execution
 - etc.

Computer Organization

- The Branch of **Computer Organization** is concerned with implementing a particular ISA.
- Deals with various hardware implementation techniques, i.e. the *Microarchitecture level*.
- For Example, ARM licenses other companies like Qualcomm and Apple to use its ARM ISA.
 - Each company implements this ISA differently, with different performance and power efficiency.
 - The Qualcomm *Krait* cores have a different microarchitecture, and the Apple A-series processors have a different microarchitecture.
- RISC-V can be implemented using different micro-architectures

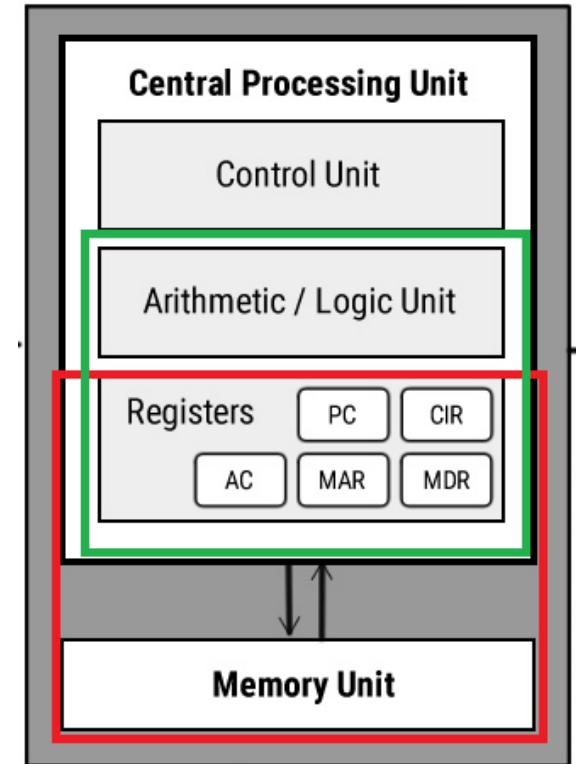
Instruction Set Architecture

- **Load-store architecture**

- Divide instructions into two categories
 - load and store instructions **between memory and registers**
 - ALU operations **between registers**, e.g. both operands and destination of an ADD operation, must be in registers
- Many **RISC** architectures, such as **ARM**, PowerPC, SPARC, **RISC-V** and MIPS, are load-store architectures
- Almost all vector processors (including many **GPUs**) use the load-store approach

- **Register-memory architecture**

- **CISC** (Complex Instruction Set Computer) instruction set architecture such as **x86**
- E.g. One of the operands for the ADD operation may be in memory, while the other is in a register



RISC vs CISC

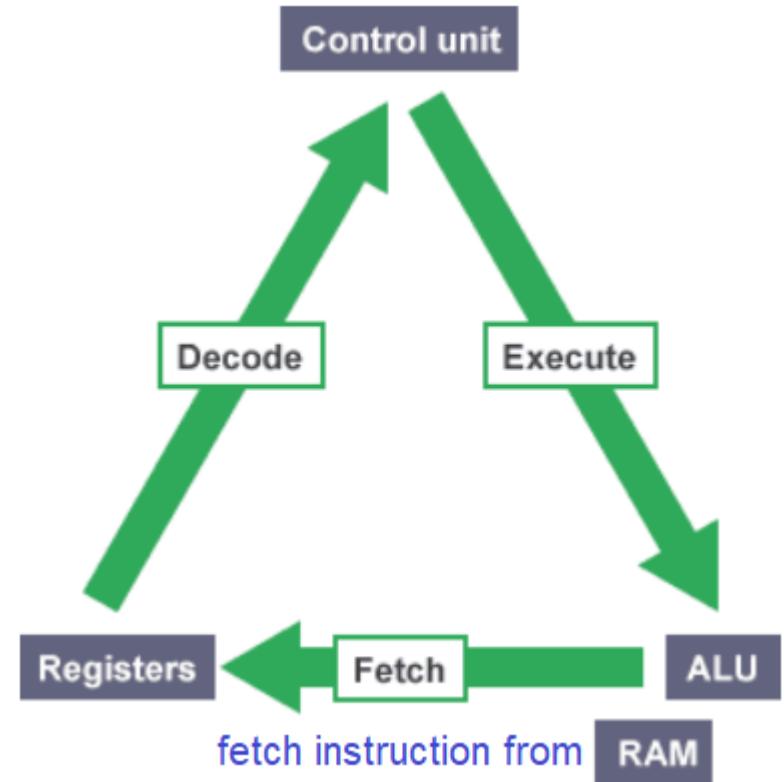
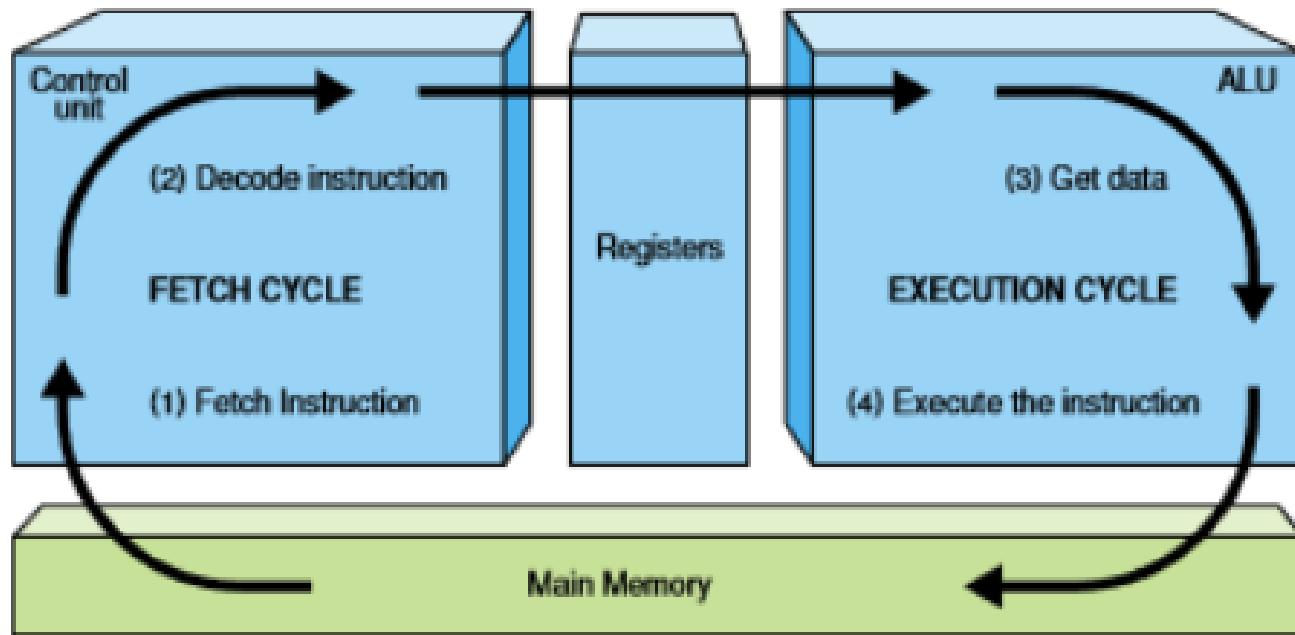
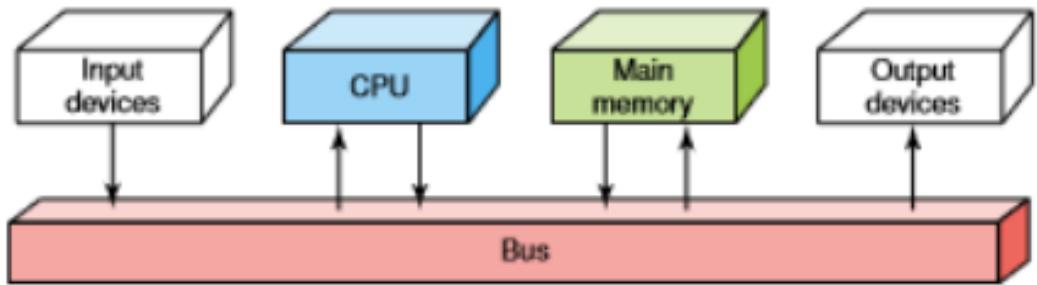
- Reduced Instruction Set Computer (RISC): *emphasises efficiency in cycles per instruction*
- CISC: Complex Instruction Set Computer: CISC *emphasises efficiency in instructions per program*

CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design <ul style="list-style-type: none">– the ISA does as much as possible using hardware circuitry	Software-centric design <ul style="list-style-type: none">– High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

Instruction Execution

- The CPU executes each instruction in a series of small steps.
- Roughly speaking, the steps are as follows:
 1. **Fetch** the next instruction from memory into the instruction register (IR).
 - Change the Program Counter (PC) to point to the next instruction.
 2. **Decode** the instruction (determine the operation, the operands).
 - If the instruction uses a word in memory, determine where it is.
 - Fetch the word, if needed, into a CPU register.
 3. **Execute** the instruction.
 - Go to step 1 to begin executing the next instruction.
- This sequence of steps is frequently referred to as the **fetch-decode-execute** cycle.

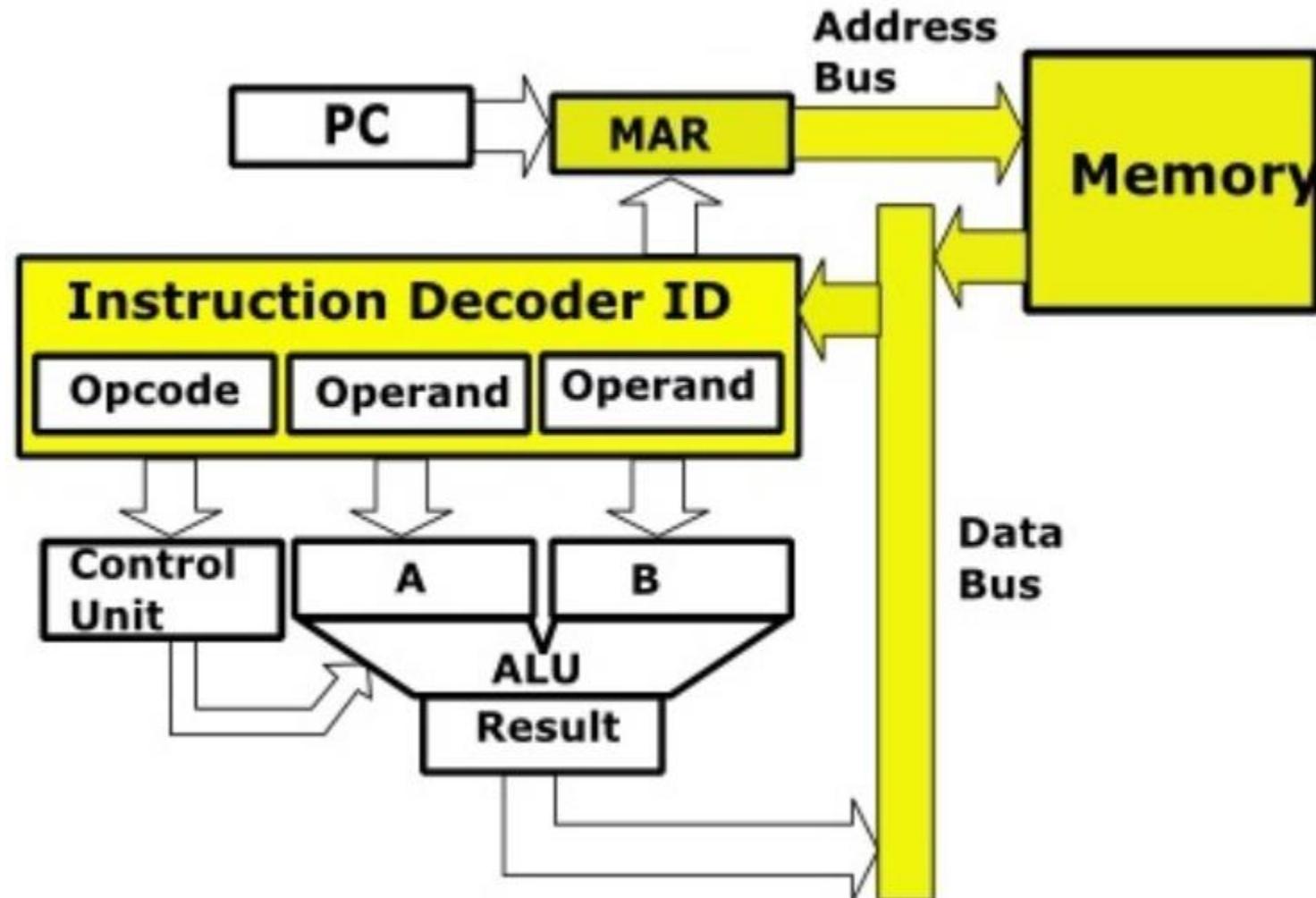
The Fetch-Decode-Execute Cycle



The Fetch-Decode-Execute Cycle illustration

Fetch

- Fetch instruction from memory at memory address in MAR
- The content of memory address in MAR is transferred over Data bus to the Current Instruction Register (CIR), to be decoded by the Instruction Decoder (ID)

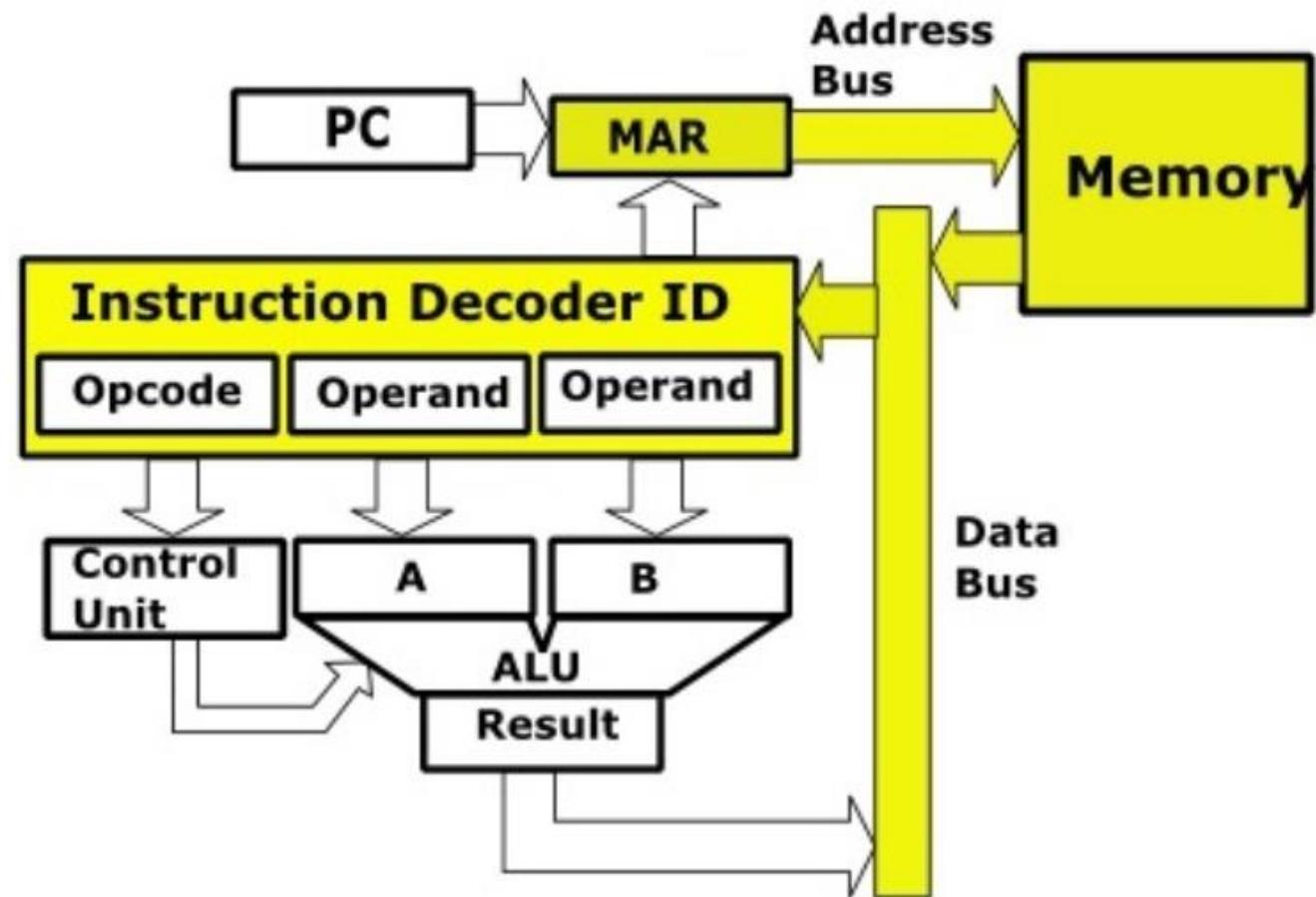


The Fetch-Decode-Execute Cycle illustration

Decode

The Instruction decoder determines:

1. The operation to be performed
2. Where to get the operands
3. Where the result should go

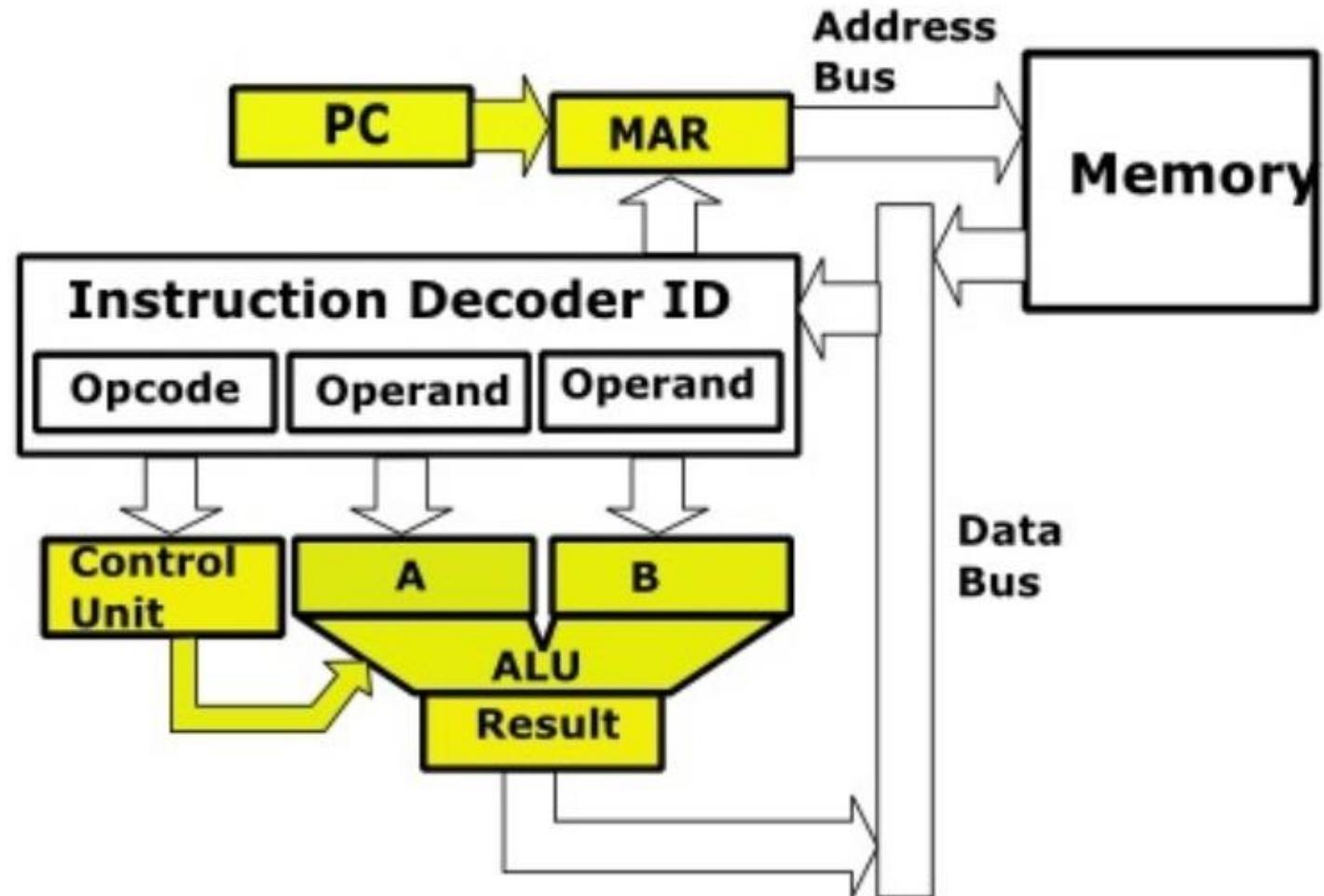


The Fetch-Decode-Execute Cycle illustration

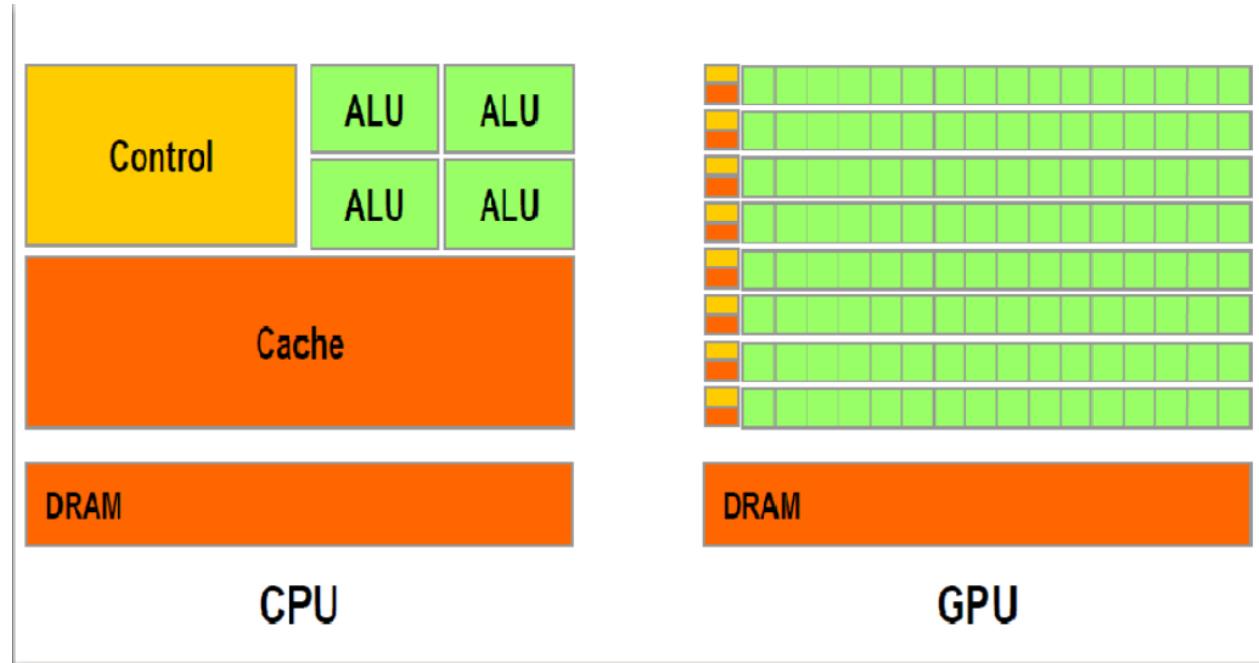
Execute

The control unit activates the control lines to:

1. Transfer the operands to the ALU input registers
2. Control the ALU to perform the operation specified in the Instruction Opcode
3. Place the result in the output register.
4. Increment the PC
5. Place the PC or Branch address in the MAR to fetch the next instruction



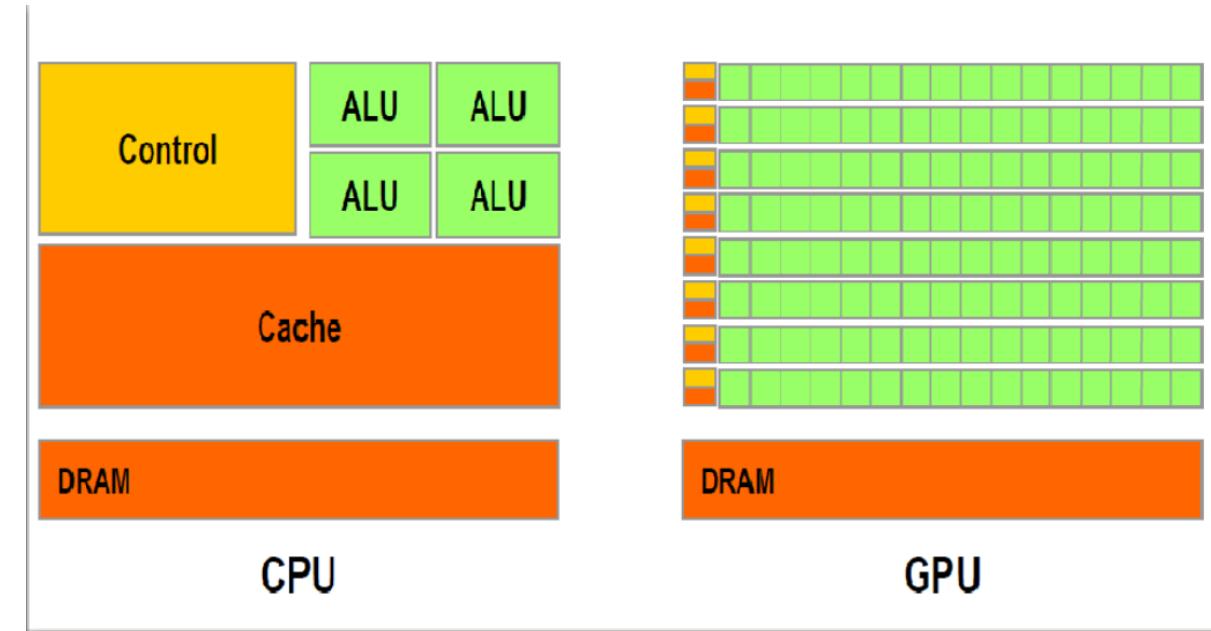
CPU vs GPU



A CPU (central processing unit) works together with a GPU (graphics processing unit) to increase the throughput of data and the number of concurrent calculations within an application.

CPU vs GPU

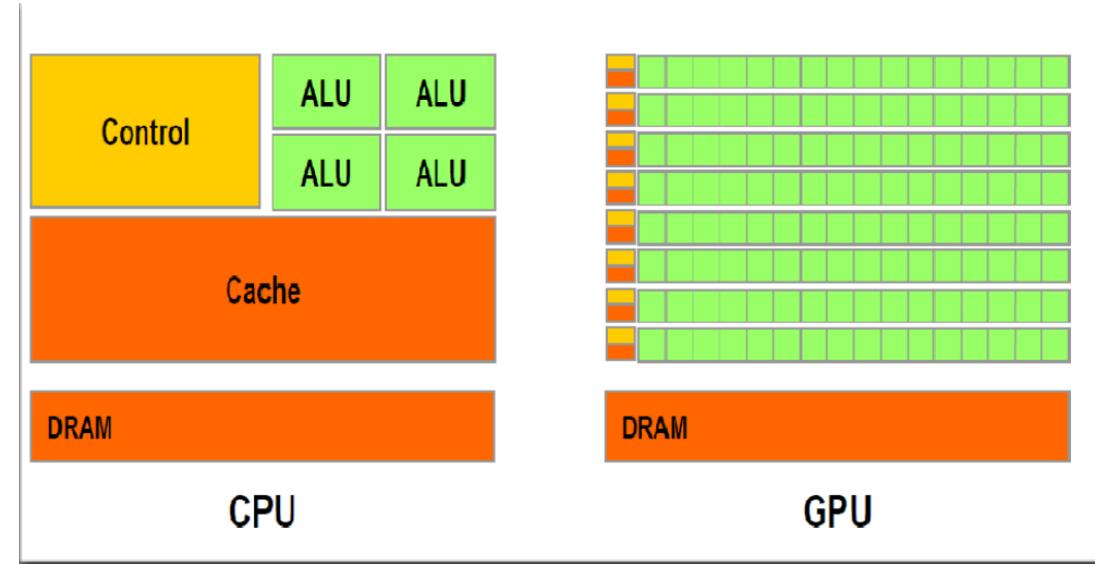
- **CPU:** The Central Processing Unit
 - has a large and broad instruction set, managing every input and output of a computer, which a GPU cannot do.
 - handles a wide range of tasks quickly (as measured by CPU clock speed)
 - is the taskmaster of the entire system, coordinating a wide range of general-purpose computing tasks.
 - Limited in the concurrency of tasks that can be running.
 - There might be 24 to 48 high-speed CPU cores in a server environment.



Ref: <https://www.omnisci.com/technical-glossary/cpu-vs-gpu>

CPU vs GPU

- **GPU:** Graphics Processing Unit
 - quickly renders high-resolution images and video concurrently.
 - accelerates calculations involving massive amounts of data, e.g. machine learning, financial simulations, risk modelling, and many other scientific computations.
 - *Can process data several orders of magnitude faster than a CPU due to massive parallelism.*
 - Less versatile than CPUs.
 - Adding 4 to 8 GPUs to a server can provide as many as 40,000 additional cores.
 - GPUs were used for mining cryptocurrencies such as Bitcoin or Ethereum.
 - Alternatives to GPU
 - Field-Programmable Gate Arrays (FPGA)
 - Application Specific Integrated Circuits (ASIC)



Ref: <https://www.omnisci.com/technical-glossary/cpu-vs-gpu>

End

A Brief Introduction to Hardware Description Languages (HDLs)



School of Electronic Engineering
Xiaojun Wang
Office: S431
xiaojun.wang@dcu.ie

Why use HDLs?

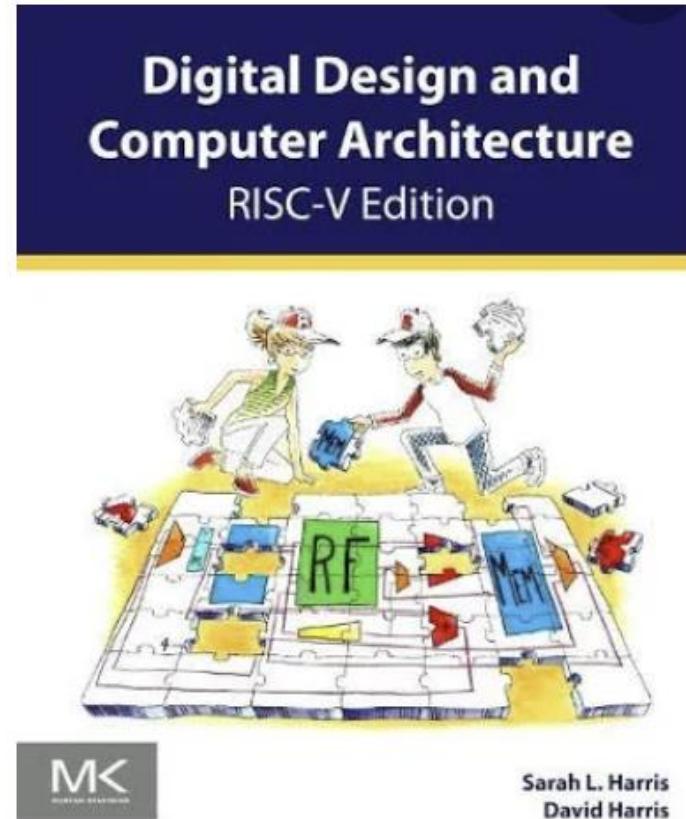
- Hardware Description Languages (HDL) are used in the electronic design process
 - CPU, GPU, SoC, ASIC, CPLD, FPGA designs
 - Soft IP development, technology-independent
- HDLs are used for Modelling/Simulation/Synthesis
- HDLs replace Gate-Level schematics just like high-level programming languages replace assembly code.
- HDLs and synthesis tools allow hardware designers the same high-level design compilation methods enjoyed by software designers.
- HDL increases designers' productivity
 - The Apple **A17 Pro** (12 September 2023) is a 64-bit ARM-based System On a Chip (SoC) used in **iPhone 15 Pro** and **iPhone 15 Pro Max** models. 6-core CPU and 6-core GPU, 8 GB RAM. 19 billion transistors. **3 nm** technology.

Why use an HDL (Hardware Description Language)

- An HDL is similar to a high-level software programming language; it provides a means of:
 - Top-down approach for large hardware design projects
 - Functional simulation early in the design flow
 - Synthesis of HDL code to gates
 - Early testing of various design implementations
 - Reuse of RTL code
 - Incorporation of design changes and corresponding changes in documentation efficiently
 - Designer/user communication at the desired *abstraction level* of details

Abstraction

Hiding details when they aren't important



1st Edition - July 12, 2021

Paperback ISBN: 9780128200643

eBook ISBN: 9780128200650

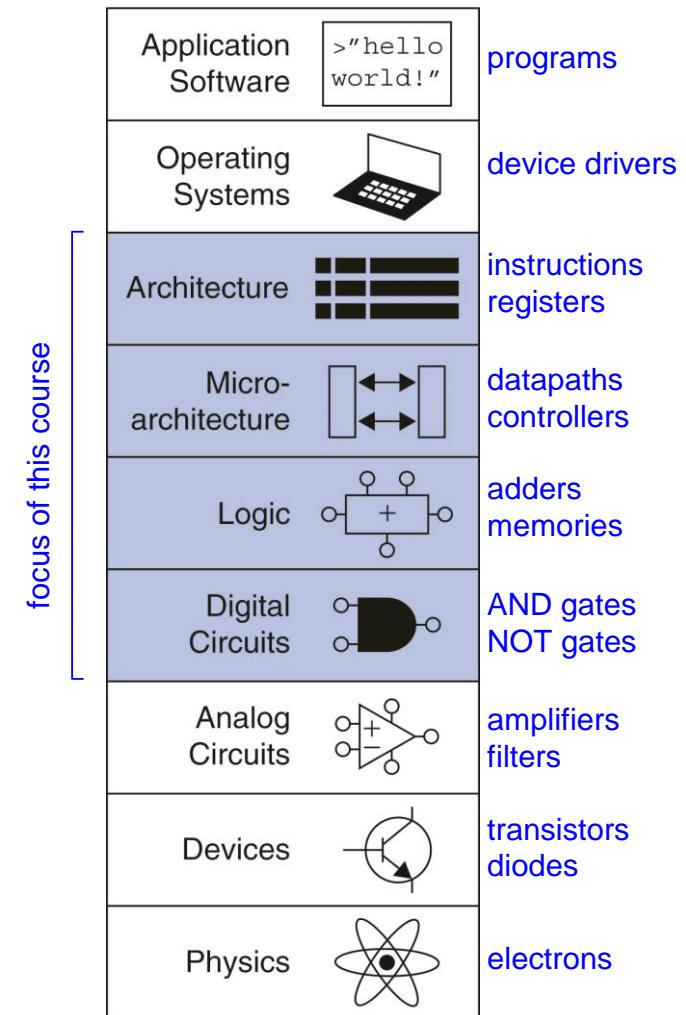
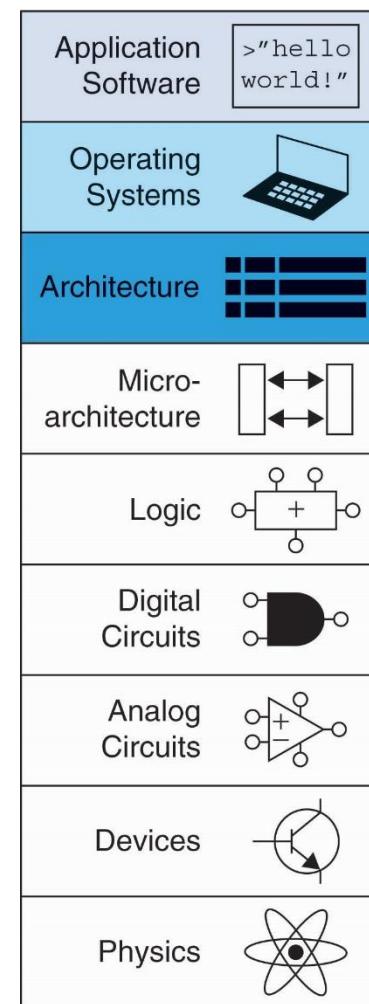


Diagram and slides from the book

Introduction

- **Architecture:** programmer's view of computer (Chapter 6)
 - Defined by instructions & operands locations
- **Microarchitecture:** how to implement an architecture in hardware (Chapter 7)



Assembly Language

- **Instructions:** commands in a computer's language
 - **Assembly language:** human-readable format of instructions
 - **Machine language:** computer-readable format (1's and 0's)
- **RISC-V architecture:**
 - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
 - First widely accepted open-source computer architecture

Once you've learned one architecture, it's easier to learn others

Design levels: Software vs Hardware

Abstraction Level	Software Design	Hardware Design	Hardware Synthesis
Algorithmic Level	Python, Java, C++, C, Ada	VHDL, SystemVerilog , Verilog, SystemC, ELLA, UDL/I, etc	High-Level-Synthesis (HLS)
Register-Transfer-Level (RTL)	Assembly language	Truth tables, state tables	RTL synthesis
Logic Level	Machine code	Boolean equations, gate-level circuit schematic	Logic synthesis

Typical Software Design Flow

High-level language program in C
(hardware-independent)

```
swap(int v[], int k)
{ int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

C compiler



Assembly Language program
(Hardware-dependent, this is for MIPS ISA)

```
swap:
    muli $2, $5,4
    add $2, $4, $2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler



Binary machine language program
(Hardware-dependent, this is for MIPS)

```
000000001010000100000000000011000
00000000100011100001100000100001
100011000110001000000000000000000
1000110011100100000000000000000000
1010110011100100000000000000000000
1010110001100010000100000000000000
000000111100000000000000000000000000
```

Typical Software Design Flow

- A Compiler converts hardware-independent high-level language code into a hardware-dependent lower-level Assembly language code.

**High-level language program in C
(Hardware-independent)**

```
swap(int v[], int k)
{ int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

C compiler

**RISC-V Assembly Language program
(Hardware-dependent)**

```
# Function: swap(int v[], int k)
# Arguments:
#   a0 -> base address of v[]
#   a1 -> index k

swap:
    slli t0, a1, 2          # t0 = k * 4 (multiply index by 4 to account for word size)
    add t1, a0, t0           # t1 = address of v[k] (base + offset)

    lw t2, 0, t1             # t2 = v[k] (load v[k] into t2)
    lw t3, 4(t1)              # t3 = v[k+1] (load v[k+1] into t3)

    sw t3, 0(t1)              # store v[k+1] into v[k] (store t3 at v[k])
    sw t2, 4(t1)              # store temp (v[k]) into v[k+1] (store t2 at v[k+1])

    ret                      # return from function
```

Typical Software Design Flow

- An **assembler** converts assembly language into machine language

RISC-V Assembly Instruction	RISC-V Machine Code (Hex)	RISC-V Machine Code (Binary)
slli t0, a1, 2	0x0005A513	0000000000000001011010010100010011
add t1, a0, t0	0x005302B3	0000000010100110000001010110011
lw t2, 0(t1)	0x00032283	00000000000000011001000101000011
lw t3, 4(t1)	0x00432303	0000000010000110010001100000011
sw t3, 0(t1)	0x01C30223	000000011000011001000000100011
sw t2, 4(t1)	0x007322A3	0000000011100110010001010100011
ret	0x00008067	0000000000000000100000001100111

HDL-Based Hardware Design Flow

SystemVerilog description
(Technology-independent)

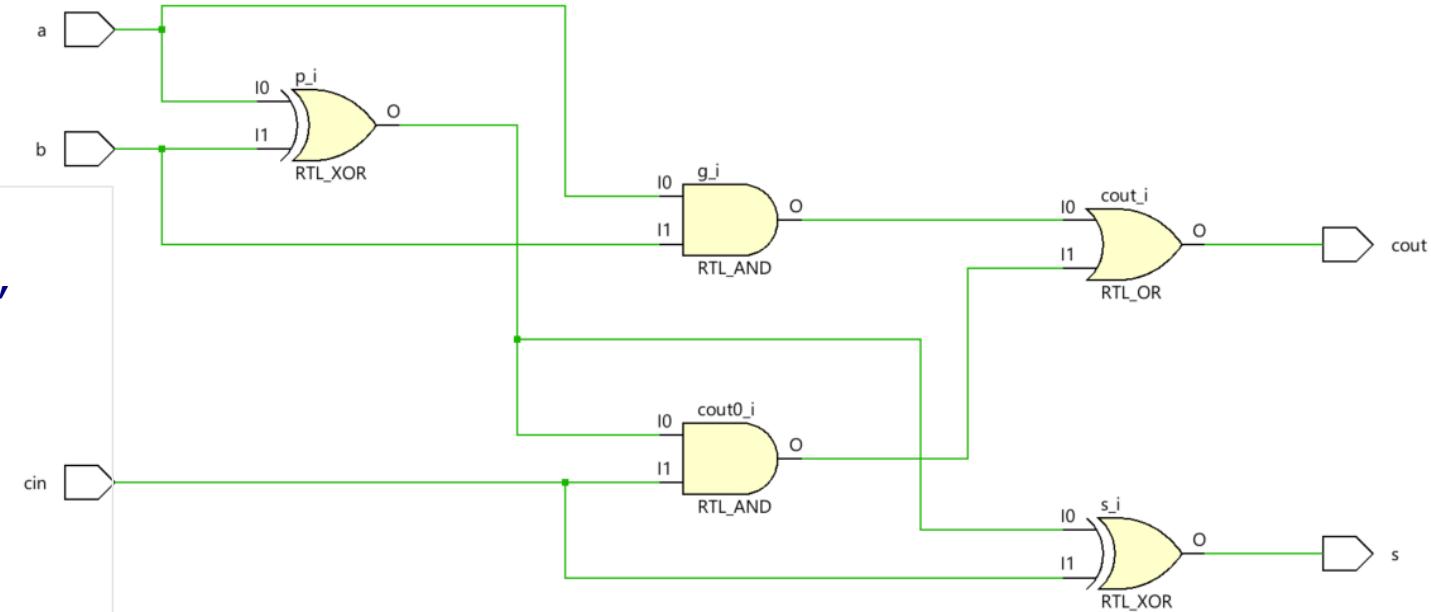
```
module fulladder(input logic a, b, cin,
                  output logic s, cout);
    logic p, g; // internal nodes

    assign p = a ^ b;
    assign g = a & b;

    assign s = p ^ cin;
    assign cout = g | (p & cin);

endmodule
```

Synthesis
&
Optimization



Technology-dependent netlist

Mapping &
Place & Route



A glimpse of various HDLs

- **VHDL:** VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, Ada-like
 - IEEE Std 1076-1987, **IEEE Std 1076-1993**, IEEE Std 1076-2008, IEEE Std 1076-2019
- **Verilog:** Mainly for the gate- and register-transfer-level design and modelling, C-like
 - IEEE Std 1364-1995, **IEEE Std 1364-2001**, IEEE Std 1364-2005, Verilog merged into SystemVerilog in 2009, *ceases to exist after 2009*
- **SystemVerilog:** extension of Verilog
 - IEEE Standard 1800-2005 (extension of Verilog for verification functionalities)
 - IEEE Standard 1800-2009 (Verilog merged into SystemVerilog)
 - IEEE Std 1800-2017
- **SystemC:** SystemC has semantic similarities to **VHDL** and **Verilog**
 - IEEE 1666-2011
- **UDL/I:** Unified Design Language for Integrated Circuit
 - Standard in Japan
- Proprietary languages
 - ELLA (an HDL originated from UK's Defence Research Agency in 1982)
 - BLM (Mentor Graphics Behavioural Language Modelling),
 - AHDL (Altera Hardware Description Language)
 - ABEL (Xilinx proprietary HDL)
 - etc.

Logic Simulation

- Simulation is a process of emulating real design behaviour in a software environment. It helps verify a design's functionality by injecting stimulus and observing the design outputs.
- The process of simulation includes:
 - Creating **test benches**, setting up libraries, and specifying the simulation settings for the Simulation
 - Generating a Netlist (if performing post-synthesis or post-implementation simulation)
 - Running a Simulation using a Vivado simulator or third-party simulators. See [Supported Simulators](#) for more information on supported simulators.

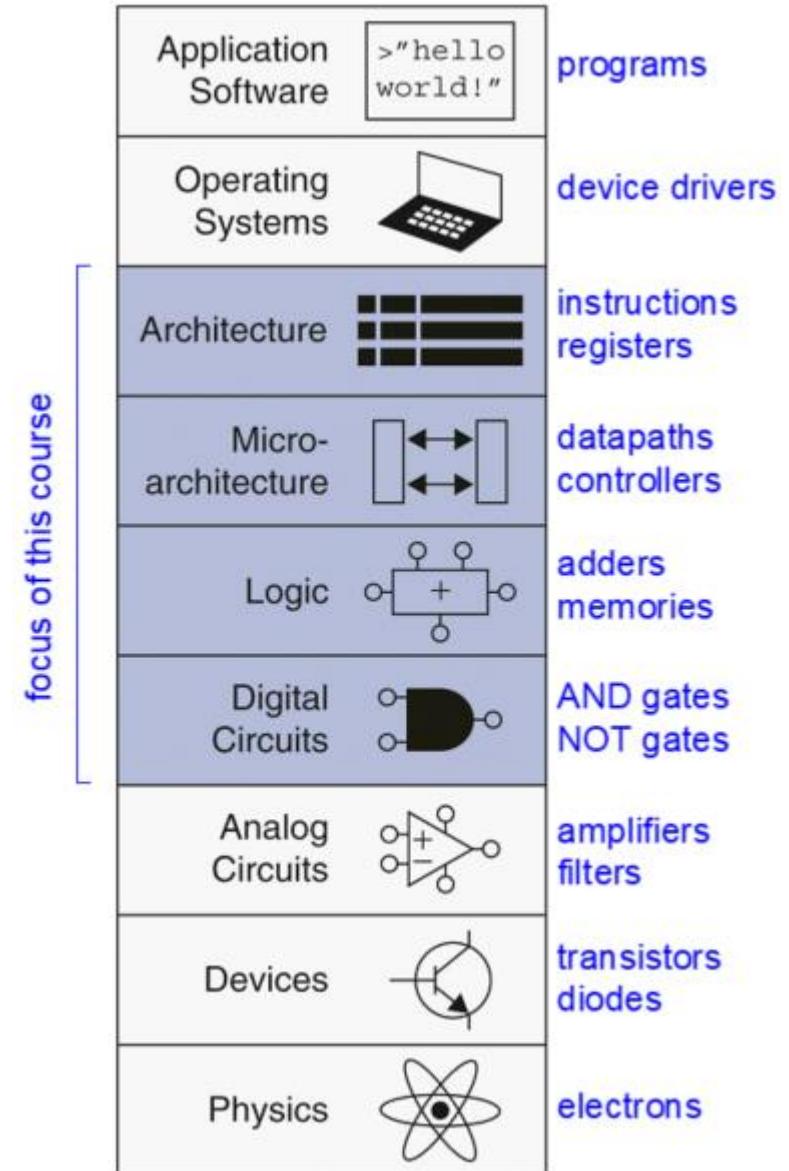
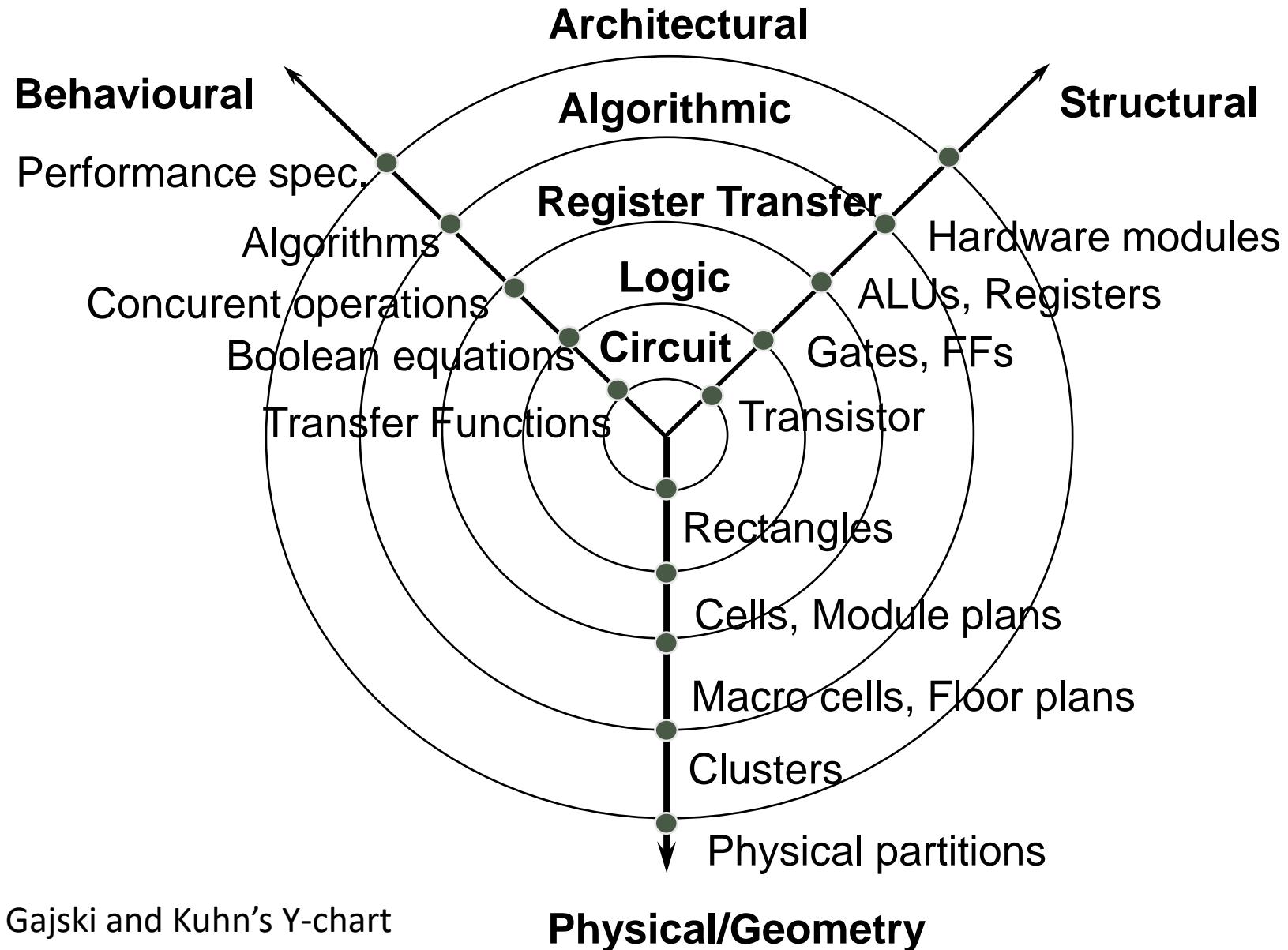
Synthesis

- Synthesis transforms a Register Transfer Level (RTL) specified design into a gate-level representation.
- Vivado synthesis supports a synthesisable subset of:
 - SystemVerilog: IEEE Standard for SystemVerilog-Unified Hardware Design Specification, and Verification Language (IEEE Std 1800-2012)
 - Verilog: IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2005)
 - VHDL: IEEE Standard for VHDL Language (IEEE Std 1076-2002)
 - VHDL 2008
 - Mixed languages: Vivado supports a mix of VHDL, Verilog, and SystemVerilog
- In most instances, the Vivado tools also support Xilinx design constraints (XDC), which is based on the industry-standard Synopsys design constraints (SDC).

AMD Xilinx Vivado Design Suite User Guide: Synthesis

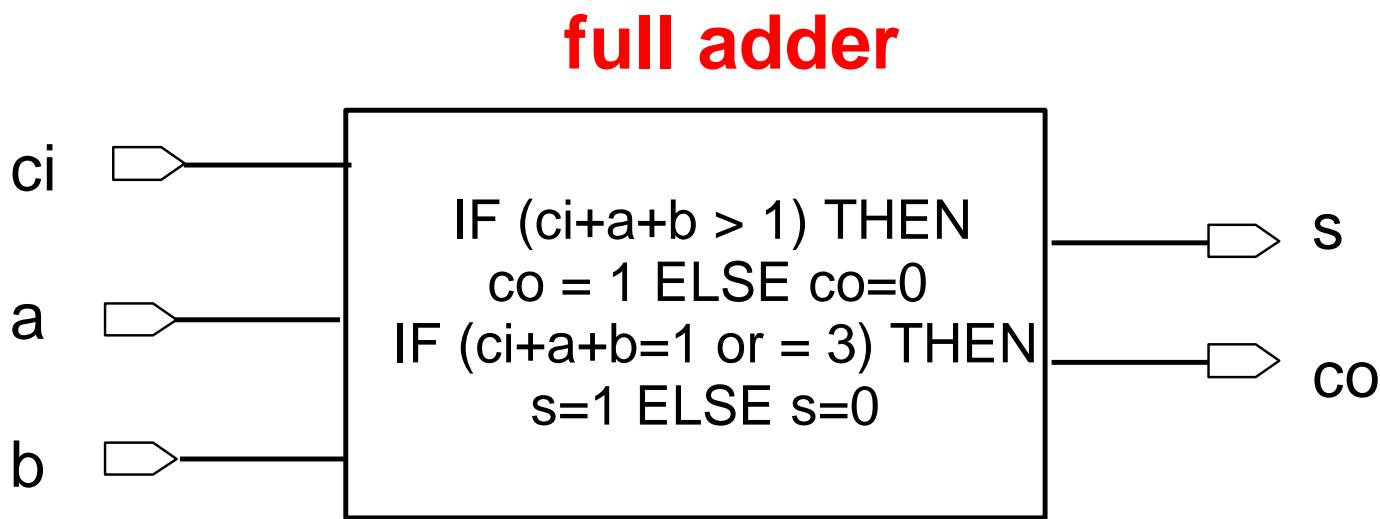
- https://loop.dcu.ie/pluginfile.php/5104321/mod_resource/content/8/ug901-vivado-synthesis.pdf
- Chapter 7: Verilog Language Support
- Chapter 8: SystemVerilog Support
- Chapter 9: Mixed Language Support
- Chapter 4: HDL Coding Techniques
- Chapter 5: VHDL Support
- Chapter 6: VHDL-2008 Language Support
- Xilinx Vivado ML 2024.1 defaults to Verilog

System Description Domains and Abstraction Levels



A SystemVerilog Design example:

- A one-bit full-adder



Truth table of a one-bit full-adder

a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The functionality of the one-bit full-adder is described in the next slide by counting the number of '1's in the input, to determine the output. (*This is NOT a good way to design in SystemVerilog*)

Behavioural Specification

```
module fa(input logic a, b, cin,
           output logic s, cout);
    logic [1:0] n; // count number of 1's in inputs

    always_comb // need begin/end because there is
    begin      // more than one statement in always
        if (a === 1'b1) n = n + 1;
        if (b === 1'b1) n = n + 1;
        if (cin === 1'b1) n = n + 1;
        cout = n[1];
        s    = n[0];
    end

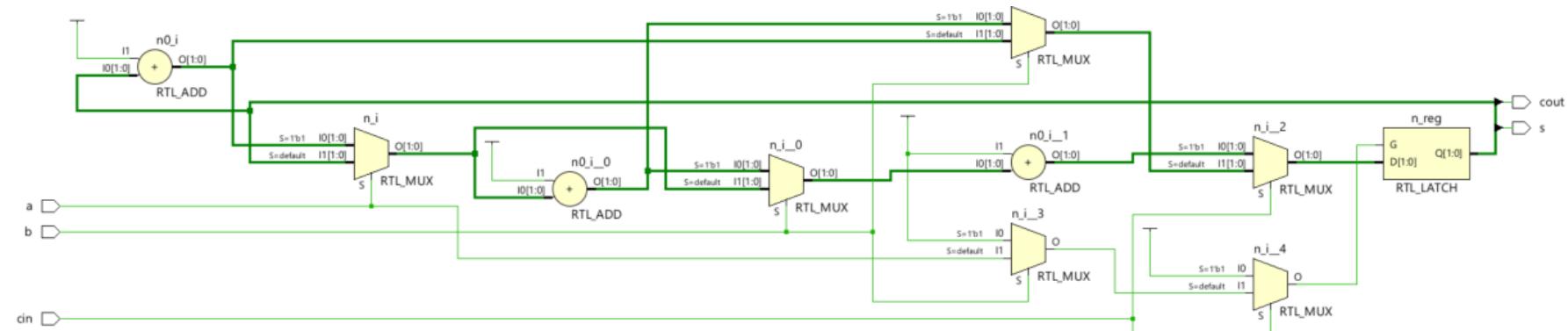
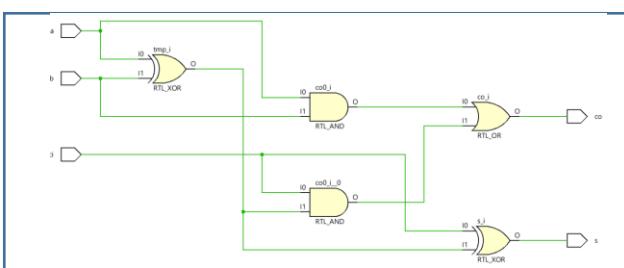
endmodule
```

- We can now simulate this high-level description model to verify the correct functionality of the design
- However, this is ***NOT a good description*** of the one-bit full-adder.
- This model shows that SystemVerilog is versatile and can describe a design at a very high abstract level.

Logic circuit produced from the above description

- As you can see the produced circuit is much larger than an optimized one-bit full-adder
- The circuit includes three two-bit adders, and other components, because the SystemVerilog description implies these operations
- The quality of the synthesized logic circuit depends on both the quality of the SystemVerilog code as well as the power of the synthesis tool*

Optimized circuit



The above code produced circuit

Use Karnaugh map for logic minimization

One-bit full-adder truth table

a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Karnaugh map for s

ab \ ci	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$\begin{aligned}s &= \bar{a}b\bar{c}_i + a\bar{b}\bar{c}_i + \bar{a}\bar{b}c_i + abc_i \\&= (\bar{a}b + a\bar{b})\bar{c}_i + (\bar{a}\bar{b} + ab)c_i \\&= (a \oplus b)\bar{c}_i + \overline{(a \oplus b)}c_i \\&= (a \oplus b) \oplus c_i\end{aligned}$$

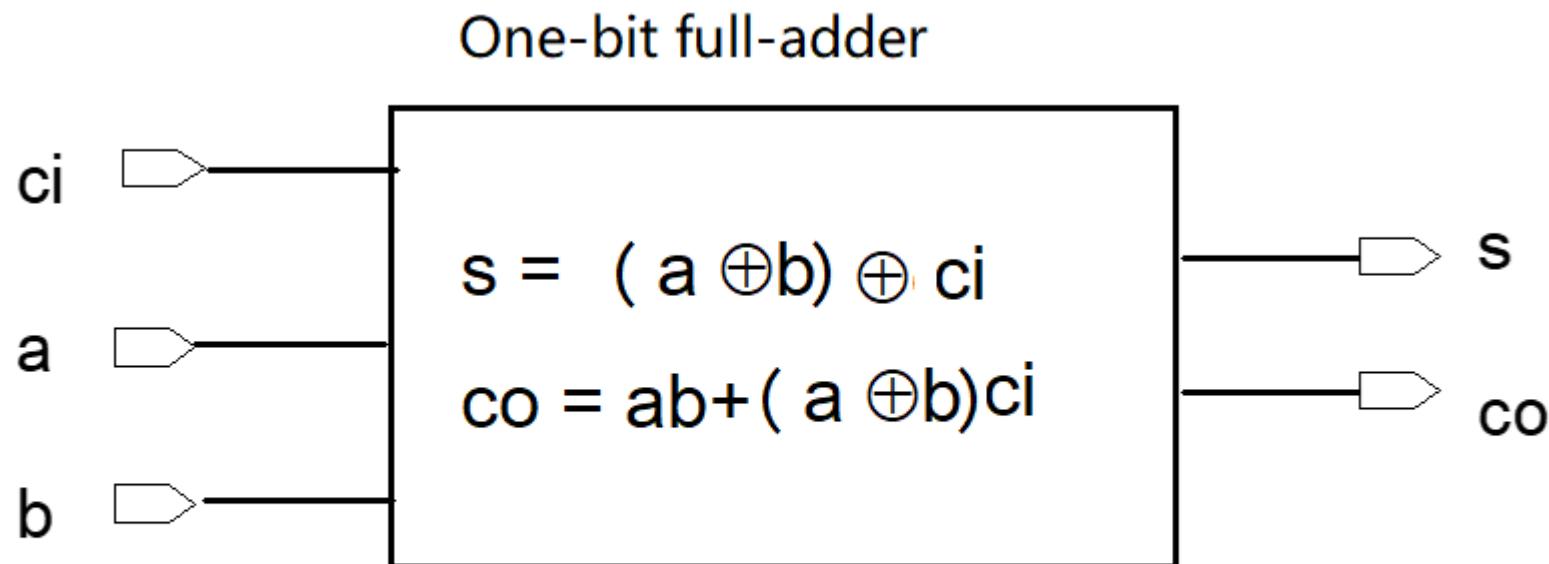
Karnaugh map for co

ab \ ci	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$\begin{aligned}co &= ab + \bar{a}bc_i + a\bar{b}c_i \\&= ab + (\bar{a}b + a\bar{b})c_i \\&= ab + (a \oplus b)c_i\end{aligned}$$

Data Flow Specification

- With the high-level description confirmed, logic equations describing the data flow are then created



Data Flow (RTL) Model of one-bit full-adder

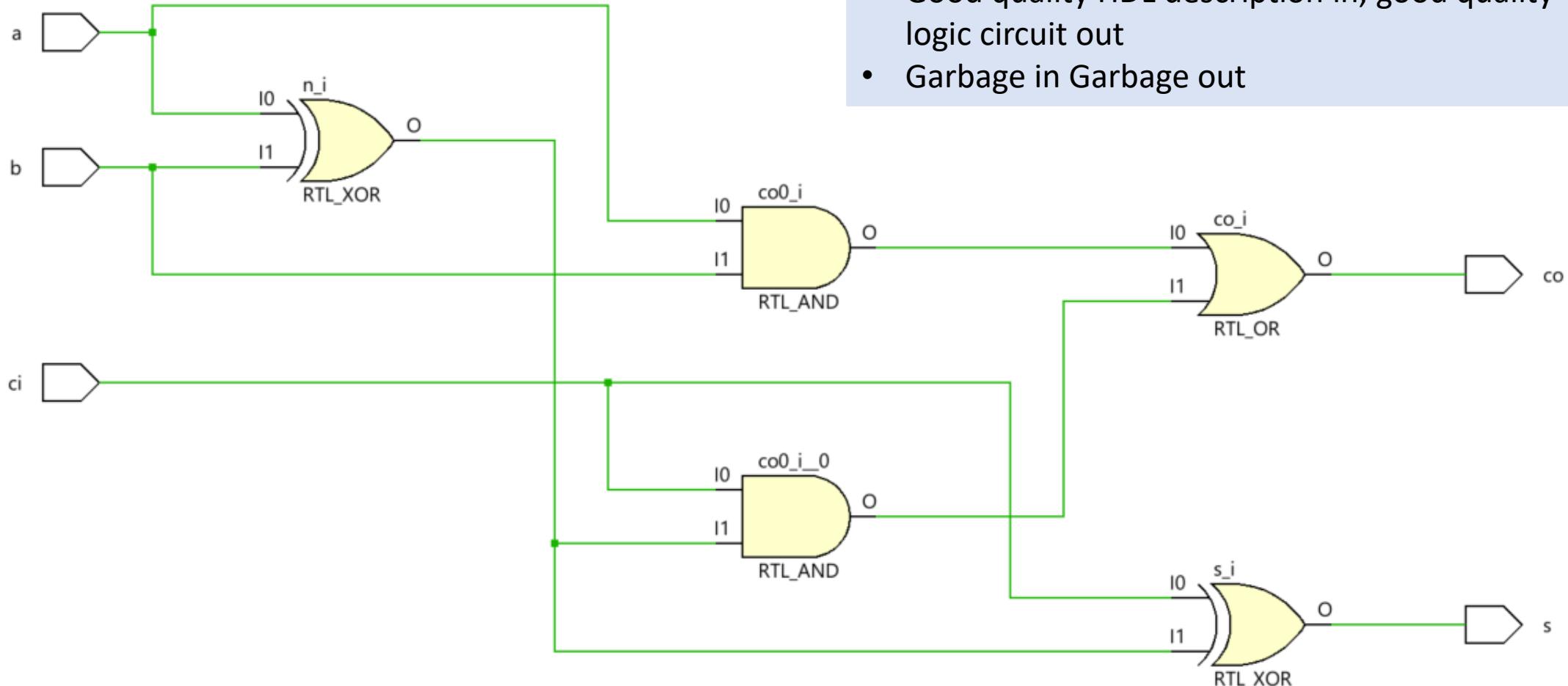
```
module fa_rtl(input logic a, b, ci,
               output logic s, co);
    logic n; // internal nodes

    assign n = a ^ b;

    assign s = ci ^ n;
    assign co = (a & b) | (ci & n);

endmodule
```

Logic circuit synthesized from the RTL code



- This is a much better implementation than the previous implementation
- Good quality HDL description in, good quality logic circuit out
- Garbage in Garbage out

SystemVerilog model based on truth-table

```
// Implies ROM lookup implementation

module fa_lut(input logic [2:0] a_b_ci,
               output logic [1:0] co_s);

  always_comb

    case (a_b_ci)

      3'b000 : co_s = 2'b00;
      3'b001 : co_s = 2'b01;
      3'b010 : co_s = 2'b10;
      3'b011 : co_s = 2'b10;
      3'b100 : co_s = 2'b01;
      3'b101 : co_s = 2'b10;
      3'b110 : co_s = 2'b10;
      3'b111 : co_s = 2'b11;

    default: co_s = 2'b00; // default (co, s) = 00

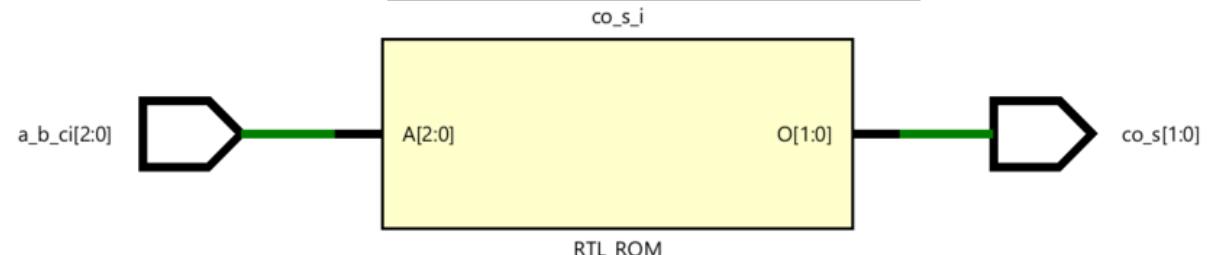
  endcase

endmodule
```

- SystemVerilog can describe the one-bit full-adder directly based on its truth-table
- Avoid Manual Karnaugh map minimization
- Will produce ROM-based look-up-table implementation.

One-bit full-adder truth table

a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



VHDL vs SystemVerilog

- Industry use is split pretty evenly between the two main HDLs
- Both languages have similar capabilities
- VHDL inherited some syntax from the software language Ada
 - **The first standardised language specifically for Hardware Design (IEEE 1076 - 1987)**
- Verilog inherited some syntax from the software language C
- Which language to use depends on
 - What your tools support
 - Company/Personal preference
- Vivado supports mixed VHDL and SystemVerilog design

End

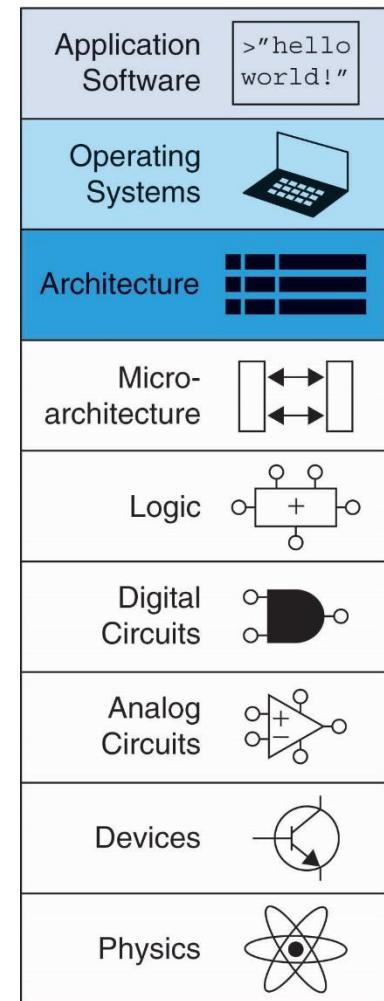
Digital Design & Computer Architecture

Sarah Harris & David Harris

Chapter 6: Architecture

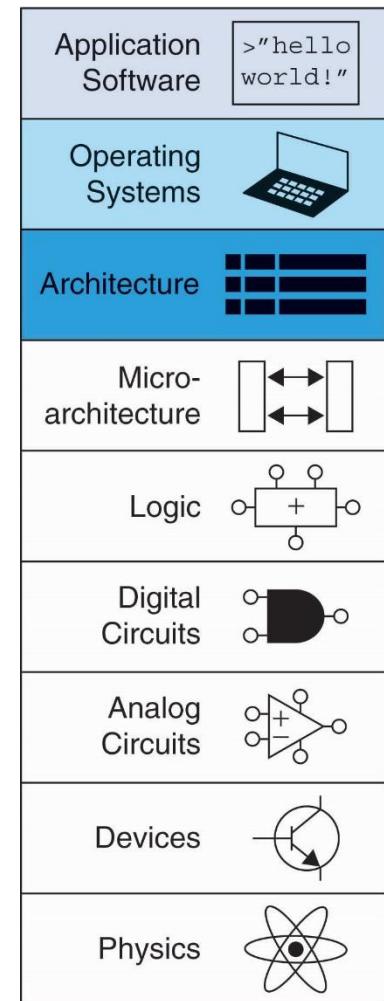
Chapter 6 :: Topics

- **Introduction**
- **Assembly Language**
- **Programming**
- **Machine Language**
- **Addressing Modes**
- **Lights, Camera, Action:
Compiling, Assembly, & Loading**
- **Odds & Ends**



Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
 - Defined by instructions & operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



Assembly Language

- **Instructions:** commands in a computer's language
 - **Assembly language:** human-readable format of instructions
 - **Machine language:** computer-readable format (1's and 0's)
- **RISC-V architecture:**
 - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
 - First widely accepted open-source computer architecture

Once you've learned one architecture, it's easier to learn others

Kriste Asanovic

- Professor of Computer Science at the University of California, Berkeley
- Developed RISC-V during one summer
- Chairman of the Board of the RISC-V Foundation
- Co-Founder of SiFive, a company that commercializes and develops supporting tools for RISC-V



Andrew Waterman

- Co-founded SiFive with Krste Asanovic
- Weary of existing instruction set architectures (ISAs), he co-designed the RISC-V architecture and the first RISC-V cores
- Earned his PhD in computer science from UC Berkeley in 2016



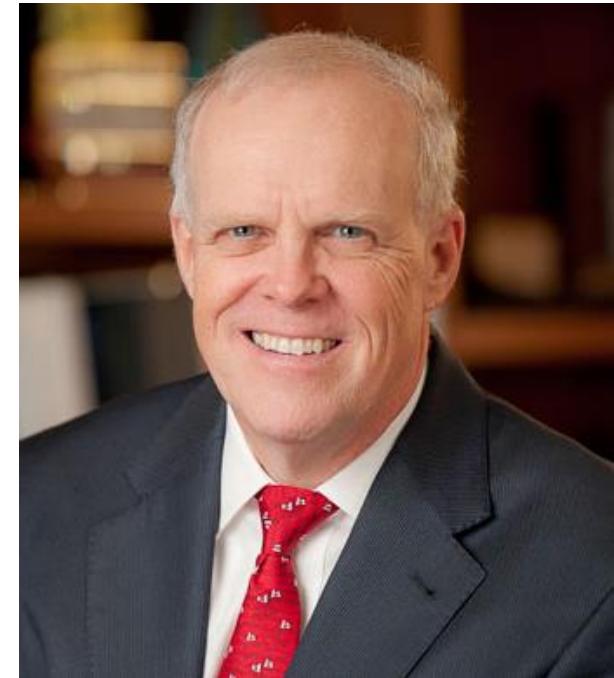
David Patterson

- Professor of Computer Science at the University of California, Berkeley since 1976
- Coinvented the Reduced Instruction Set Computer (**RISC**) with John Hennessy in the 1980's
- Founding member of RISC-V team.
- Was given the Turing Award (with John Hennessy) for pioneering a quantitative approach to the design and evaluation of computer architectures.



John Hennessy

- President of Stanford University from 2000 - 2016.
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (**RISC**) with David Patterson in the 1980's
- Was given the Turing Award (with David Patterson) for pioneering a quantitative approach to the design and evaluation of computer architectures.



Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

- 1. Simplicity favors regularity**
- 2. Make the common case fast**
- 3. Smaller is faster**
- 4. Good design demands good compromises**

Chapter 6: Architecture

Instructions

Instructions: Addition

C Code

```
a = b + c;
```

RISC-V assembly code

```
add a, b, c
```

- **add:** mnemonic indicates operation to perform
- **b, c:** source operands (on which the operation is performed)
- **a:** destination operand (to which the result is written)

Instructions: Subtraction

Similar to addition - only **mnemonic** changes

C Code

```
a = b - c;
```

RISC-V assembly code

```
sub a, b, c
```

- **sub**: mnemonic
- **b, c**: source operands
- **a**: destination operand

Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Multiple Instructions

More complex code is handled by multiple RISC-V instructions.

C Code

```
a = b + c - d;
```

RISC-V assembly code

```
add t, b, c    # t = b + c  
sub a, t, d    # a = t - d
```

Design Principle 2

Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer (RISC)***, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers (CISC)***

Chapter 6: Architecture

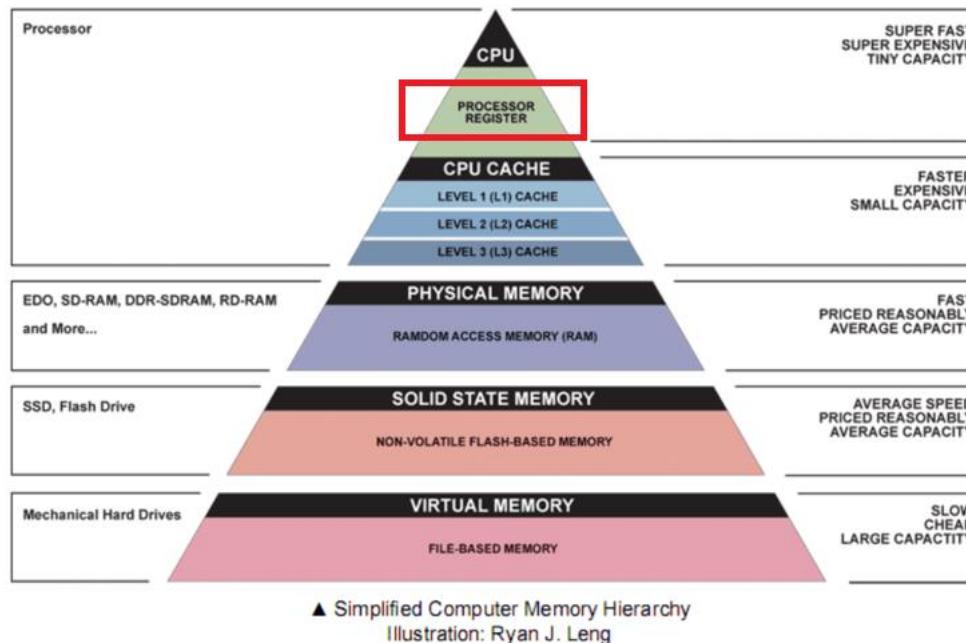
Operands

Operands

- **Operand location:** physical location in computer
 - Registers
 - Memory
 - Constants (also called *immediates*)

Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data



Design Principle 3

Smaller is Faster

- RISC-V includes only a small number of registers
 - RISC-V base ISA consists of 31 general-purpose registers x1-x31, which hold integer values.
 - The register x0 is hardwired to the constant 0.
 - There is an additional user-visible program counter **pc register** which holds the address of the current instruction.

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Hardwired to Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0 / fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers (to hold variables)
t3-6	x28-31	Temporaries

Operands: Registers

- **Registers:**
 - Can use either name (i.e., `ra`, `zero`) or `x0`, `x1`, etc.
 - Using name is preferred
- Registers used for **specific purposes**:
 - `zero` always holds the **constant value 0**.
 - the ***saved registers***, `s0`–`s11`, used to hold variables
 - the ***temporary registers***, `t0`–`t6`, used to hold intermediate values during a larger computation
 - Discuss others later

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c  
add s0, s1, s2
```

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c  
add s0, s1, s2
```

indicates a single-line comment

Instructions with Constants

- addi instruction

C Code

```
a = b + 6;
```

RISC-V assembly code

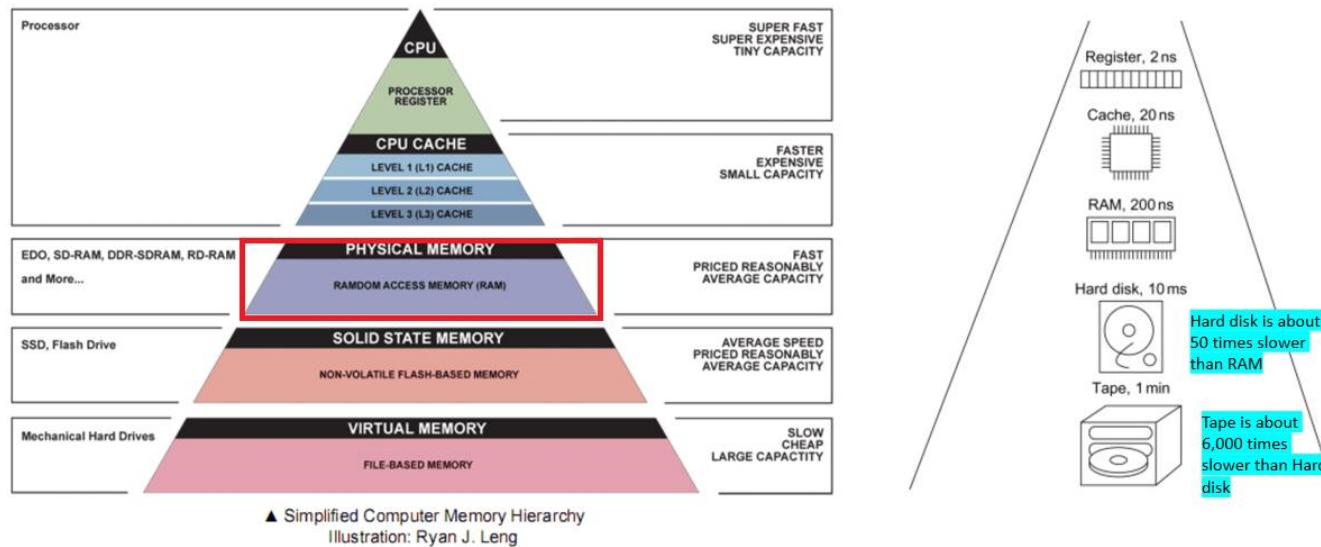
```
# s0 = a, s1 = b  
addi s0, s1, 6
```

Chapter 6: Architecture

Memory Operands

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers (or Cache)



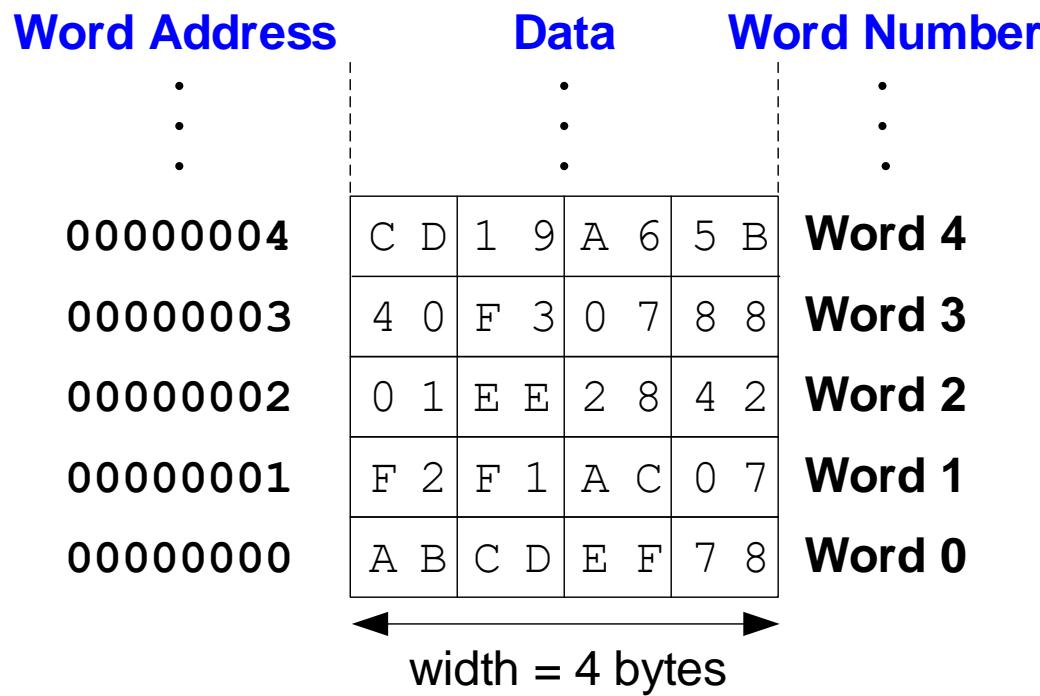
Memory

- First, we'll discuss **word-addressable** memory
- Then we'll discuss **byte-addressable** memory

RISC-V is **byte-addressable**

Word-Addressable Memory

- Each 32-bit data word has a unique address



RISC-V uses **byte-addressable** memory, which we'll talk about next.

Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (**lw**)
- **Format:**

lw t1, 5(s0) # t1 is x6

lw destination, offset(base)

- **Address calculation:**
 - add *base address* (*s0*) to the *offset* (5)
 - address = (*s0* + 5)
- **Result:**
 - *t1* holds the data value at address (*s0* + 5)

Any register may be used as base address

Name	Register Number	Usage
zero	x0	Hardwired to Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers (to hold variables)
t3-6	x28-31	Temporaries

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into s3
 - address = $(0 + 1) = 1$
 - s3 = 0xF2F1AC07 after load

Assembly code

```
lw s3, 1(zero) # read memory word 1 into s3 (x19)
```

Name	Register Number	Usage
zero	x0	Hardwired to Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers (to hold variables)
t3-6	x28-31	Temporaries

Word Address	Data				Word Number
...
00000004	C D	1 9	A 6	5 B	Word 4
00000003	4 0	F 3	0 7	8 8	Word 3
00000002	0 1	E E	2 8	4 2	Word 2
00000001	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

Writing Word-Addressable Memory

- Memory write is called a *store*
- **Mnemonic:** *store word* (`sw`)

Writing Word-Addressable

- **Example:** Write (store) the value in t4 into memory address 3
 - add the base address (zero) to the offset (0x3)
 - address: $(0 + 0x3) = 3$
 - for example, if t4 holds the value **0xFEEDCABB**, then after this instruction completes, word 3 in memory will contain that value

Offset can be written in **decimal** (default) or **hexadecimal**

Assembly code

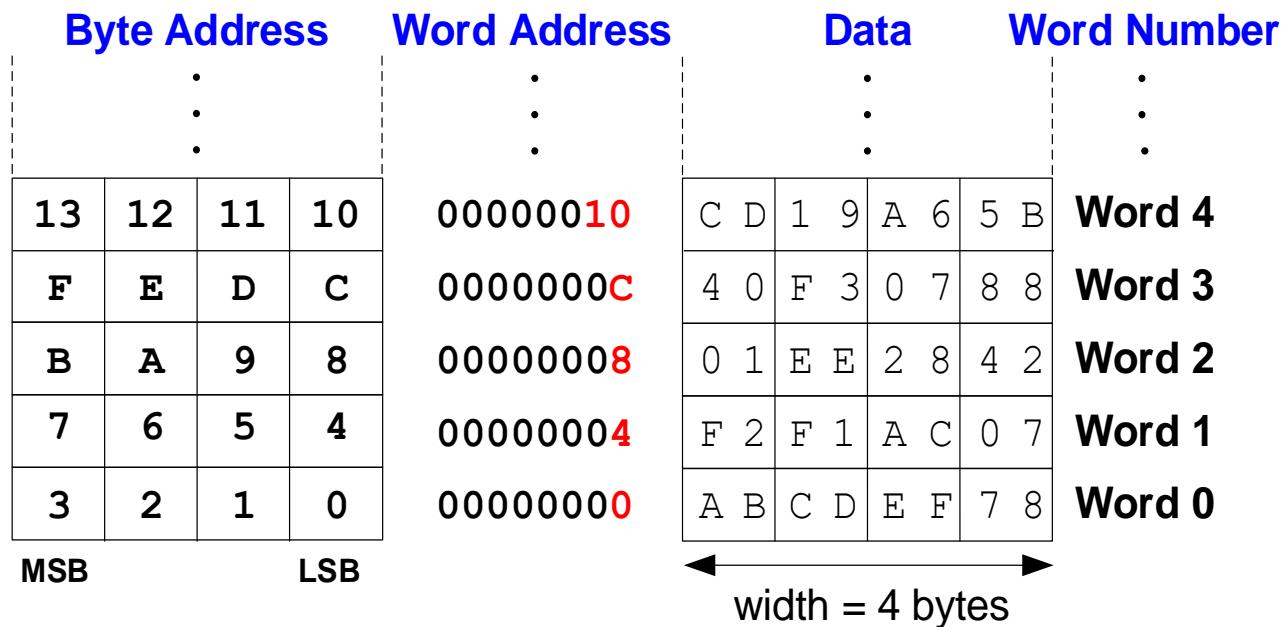
```
sw t4, 0x3(zero) # write the value in t4 (x29)
                   # to memory word 3
```

Word Address	Data				Word Number
:	:	:	:	:	:
00000004	C D	1 9	A 6	5 B	Word 4
00000003	4 0	F 3	0 7	8 8	Word 3
00000002	0 1	E E	2 8	4 2	Word 2
00000001	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

Name	Register Number	Usage
zero	x0	Hardwired to Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers (to hold variables)
t3-6	x28-31	Temporaries

Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address **increments by 4**



Reading Byte-Addressable Memory

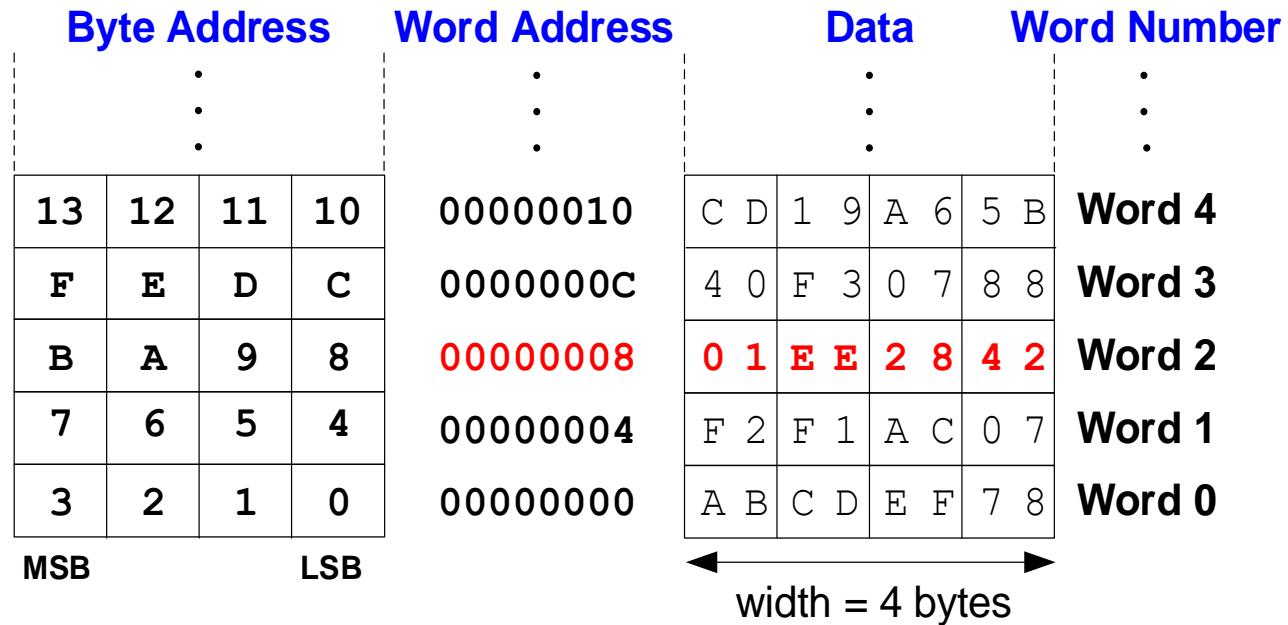
- The address of a memory word must now be multiplied by 4. For example,
 - the address of memory word 2 is $2 \times 4 = 8$
 - the address of memory word 10 is $10 \times 4 = 40$ (0x28)
- RISC-V is **byte-addressed**, not word-addressed

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into s3.
- s3 holds the value **0x1EE2842** after load

RISC-V assembly code

```
lw s3, 8(zero) # read word at address 8 into s3
```

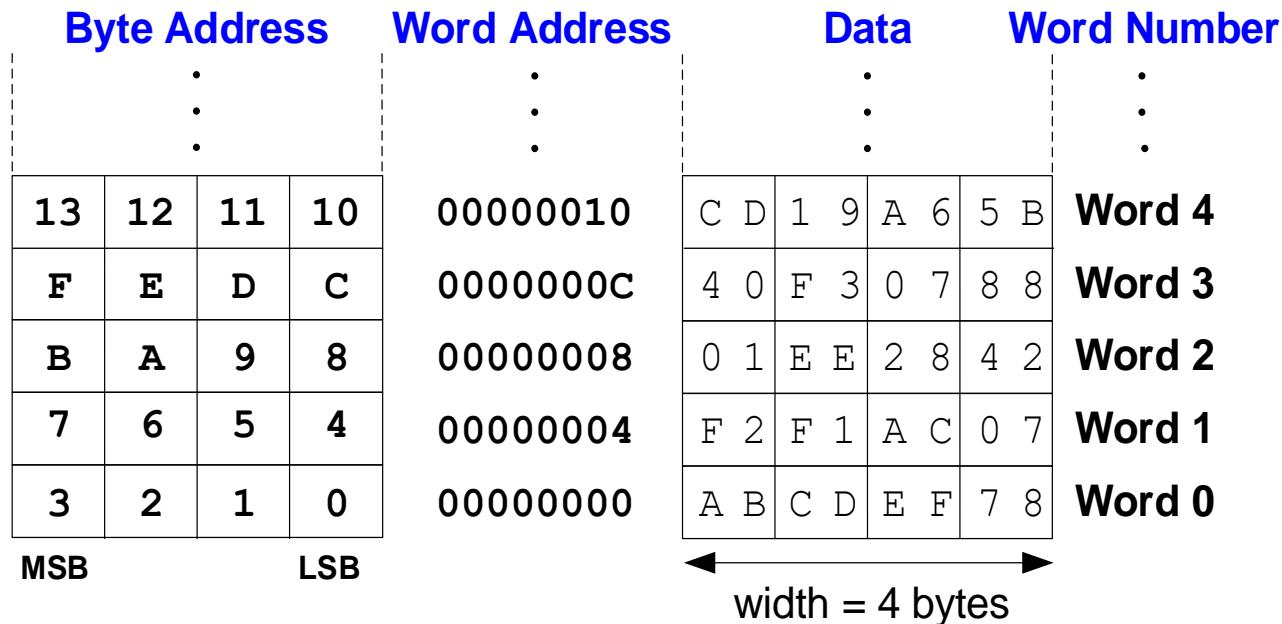


Writing Byte-Addressable Memory

- **Example:** store the value held in `t7` into memory address `0x10` (16)
 - if `t7` holds the value `0xAABBCCDD`, then after the `sw` completes, word 4 (at address `0x10`) in memory will contain that value

RISC-V assembly code

```
sw t7, 0x10(zero) # write t7 into address 16
```

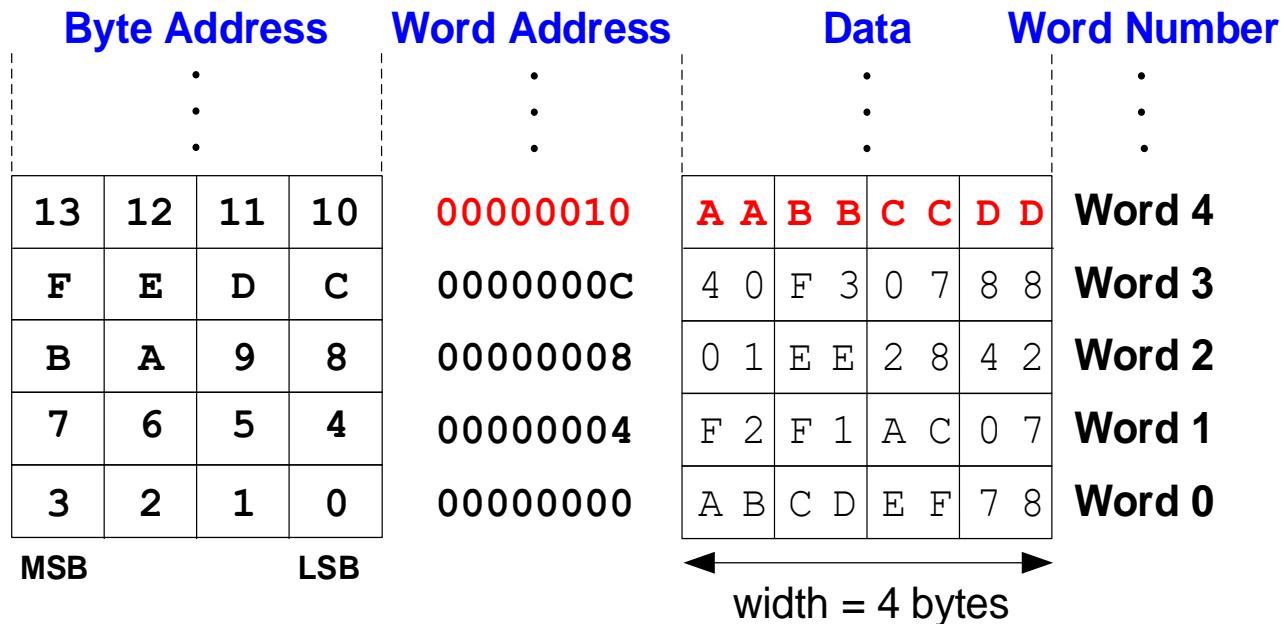


Writing Byte-Addressable Memory

- **Example:** store the value held in `t7` into memory address `0x10` (16)
 - if `t7` holds the value `0xAABBCCDD`, then after the `sw` completes, word 4 (at address `0x10`) in memory will contain that value

RISC-V assembly code

```
sw t7, 0x10(zero) # write t7 into address 16
```



Chapter 6: Architecture

Generating Constants

Generating 12-Bit Constants

- 12-bit signed constants (immediates) using addi:

C Code

```
// int is a 32-bit signed word  
int a = -372;  
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b  
addi s0, zero, -372  
addi s1, s0, 6
```

Generating 12-Bit Constants

- 12-bit signed constants (immediates) using addi:

C Code

```
// int is a 32-bit signed word  
int a = -372;  
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b  
addi s0, zero, -372  
addi s1, s0, 6
```

Any immediate that needs **more than 12 bits** cannot use this method. (This will become clear when we learn the RISC-V instruction encoding)

Generating 32-bit Constants

- Use load upper immediate (lui) and addi
- lui: puts an immediate in the upper 20 bits of the destination register and 0's in the lower 12 bits

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui s0, 0xFEDC8
addi s0, s0, 0x765
```

Generating 32-bit Constants

- Use load upper immediate (lui) and addi
- lui: puts an immediate in the upper 20 bits of the destination register and 0's in the lower 12 bits

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that addi **sign-extends** its 12-bit immediate

Generating 32-bit Constants

- If **bit 11** of 32-bit constant is **1**, increment upper 20 bits by **1** in lui

C Code

```
int a = 0xFEDC8EAB;
```

RISC-V assembly code

```
# s0 = a  
lui s0, 0xFEDC9      # s0 = 0xFEDC9000  
addi s0, s0, -341    # s0 = 0xFEDC9000 + 0xFFFFFEAB  
                      #     = 0xFEDC8EAB
```

Note: $-341 = 0xEAB$

Binary representation of
0x8EAB is **1000111010101011**

↑
Increment upper 20 bits by 1

↑

bit 11

↑

bit 0

↑
Sign extension

Generating 32-bit Constants

- If **bit 11** of 32-bit constant is **1**, increment upper 20 bits by **1** in lui (WHY?)

C Code

```
int a = 0xFEDC8EAB;
```

RISC-V assembly code

```
# s0 = a  
lui s0, 0xFEDC9          # s0 = 0xFEDC9000  
addi s0, s0, -341        # s0 = 0xFEDC9000 + 0xFFFFFEAB  
                          #     = 0xFEDC8EAB
```

Note: $-341 = 0xEAB$

Binary representation of

$0x8EAB$ is **1000111010101011**

↑↑
Increment upper 20 bits by 1 bit 11 bit 0

↑
Sign extension

Generating 32-bit Constants

- If **bit 11** of 32-bit constant is **1**, increment upper 20 bits by **1** in lui **(WHY?)**

C Code

```
int a = 0xFEDC8EAB;
```

RISC-V assembly code

s0 = a

lui s0, 0xFEDC9

addi s0, s0, -341

= 0xFEDC8EAЕ

Note: $-341 = 0xEAB$

Binary representation of

0x8EAB is 1000111010101011

Increment upper 20 bits by 1

bit 11

bit 0

- To cancel the effect of the sign extension of the addition instruction.

$$\begin{array}{r} \text{FFFFF} \\ + \quad \quad \quad 1 \\ \hline =\text{100000} \end{array}$$

- The upper 20 bits are not affected
 - The overflow bit '1' is ignored

This isn't very easy.
Are there simpler
ways?

Generating 32-bit Constants

- Use `ori` (OR immediate) instead of `addi`

C Code

```
int a = 0xFEDC8EAB;
```

RISC-V assembly code

```
# s0 = a
lui s0, 0xFEDC8          # s0 = 0xFEDC8000
ori s0, s0, 0xEAB        # s0 = 0xFEDC8000 or 0xEAB
                          #      = 0xFEDC8000 | 0x00000EAB
                          #      = 0xFEDC8EAB
```

Generating 32-bit Constants

- Use the RISC-V pseudo instruction `li`_(load immediate)
 - [RISC-V Reference Card](#) (pseudo instruction on Page 4)

RISC-V assembly code

```
# s0 = a
li s0, 0xFEDC8EAB; # s0 = 0xFEDC8EAB;
```

- The `li` pseudo-instruction is a convenience provided by the assembler
- Make it easier to load constants without manually writing multiple instructions.
 - For small immediate values (12-bit or less), `li` is translated into a single `addi` instruction.
 - For larger immediate values, it is broken down into multiple instructions.

Chapter 6: Architecture

End of week 2 slides

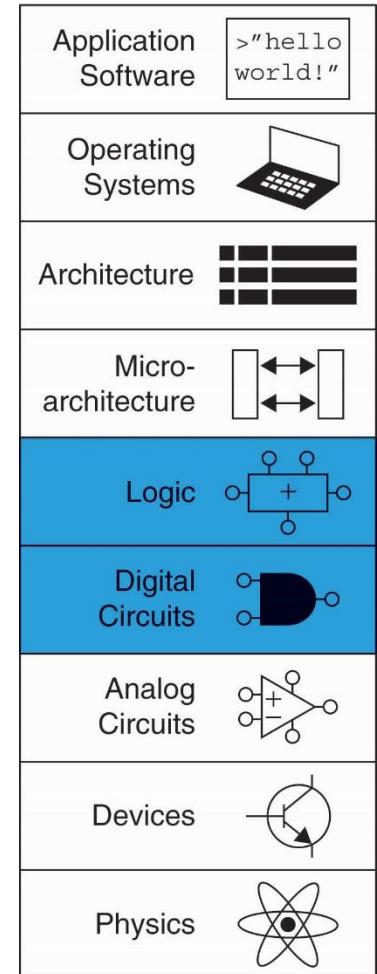
Digital Design & Computer Architecture

Sarah Harris & David Harris

Chapter 4: Hardware Description Languages

Chapter 4 :: Topics

- **Introduction**
- **Combinational Logic**
- **Delays**
- **Sequential Logic**
- **Combinational Logic w/ Always**
- **Blocking & Nonblocking Assignments**
- **Finite State Machines**
- **Parameterized Modules**
- **Testbenches**



Chapter 4: Hardware Description Languages

Introduction

Introduction

- **Hardware description language (HDL):**
 - Specifies logic function only
 - Computer-aided design (CAD) tool produces or *synthesises* the optimised gates
- Most commercial designs built using HDLs
- Two leading HDLs:
 - **SystemVerilog**
 - Developed in 1984 by Gateway Design Automation
 - IEEE Standard (1364) in 1995
 - Extended in 2005 (IEEE STD 1800-2009)
 - **VHDL 2008**
 - Developed in 1981 by the US Department of Defense
 - IEEE Standard (1076) in 1987
 - Updated in 2008 (IEEE STD 1076-2008)
 - VHDL-2019

HDL to Gates

- **Simulation**

- Inputs applied to circuit
- Outputs checked for correctness
- Millions of dollars saved by debugging in simulation instead of hardware

- **Synthesis**

- Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

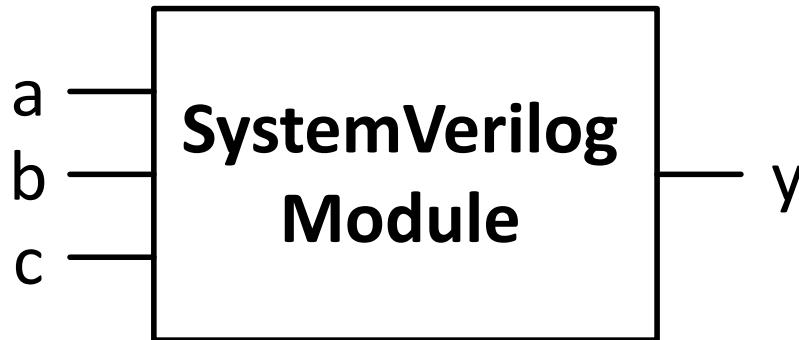
HDL: *Hardware* Description Language

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

Beware of treating HDL like software and coding without thinking of the hardware.

SystemVerilog Modules



Two types of Modules:

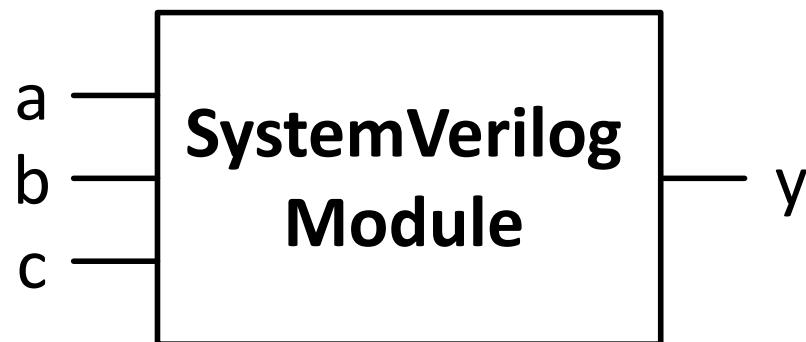
- **Behavioral:** describe what a module does
- **Structural:** describe how it is built from simpler modules

Module Declaration

SystemVerilog:

```
module example(input logic a, b, c,  
               output logic y);  
    // module body goes here  
endmodule
```

- **module/endmodule**: required to begin/end module
- **example**: name of the module



Behavioral SystemVerilog

SystemVerilog:

```
module example(input logic a, b, c,  
                output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

- **module/endmodule**: required to begin/end module
- **example**: name of the module
- **Operators**:

~: NOT

&: AND

|: OR

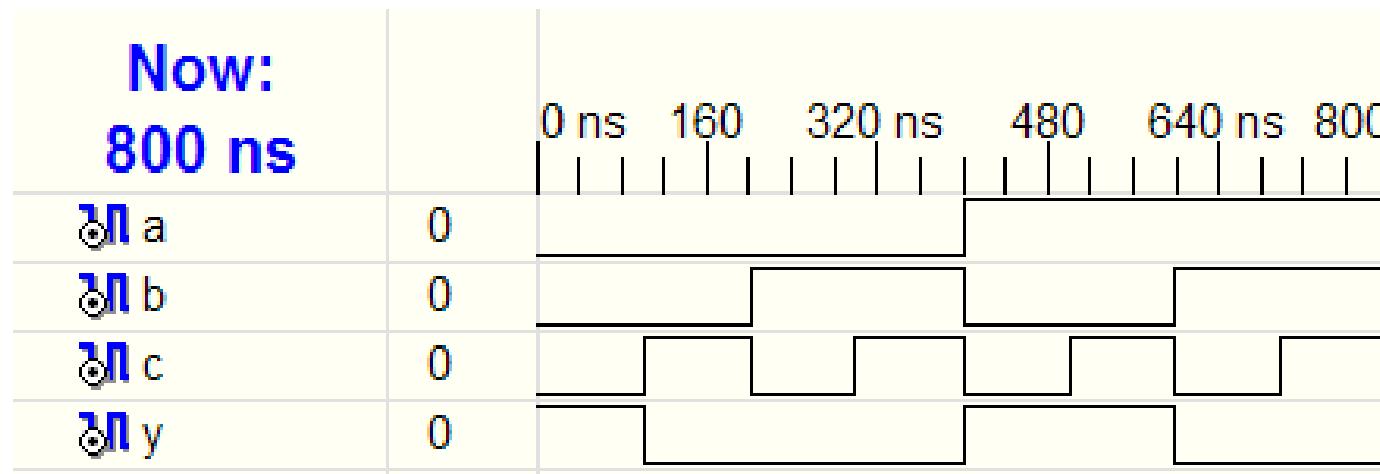
$$y = \bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot c$$

The **logic** is a 4-state data type that, in addition to 0 and 1, allows capturing the Z (high impedance) or X (unknown) behaviour of the design.

HDL Simulation

SystemVerilog:

```
module example(input logic a, b, c,  
                output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```



$$y = \bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot c$$

HDL Synthesis

SystemVerilog:

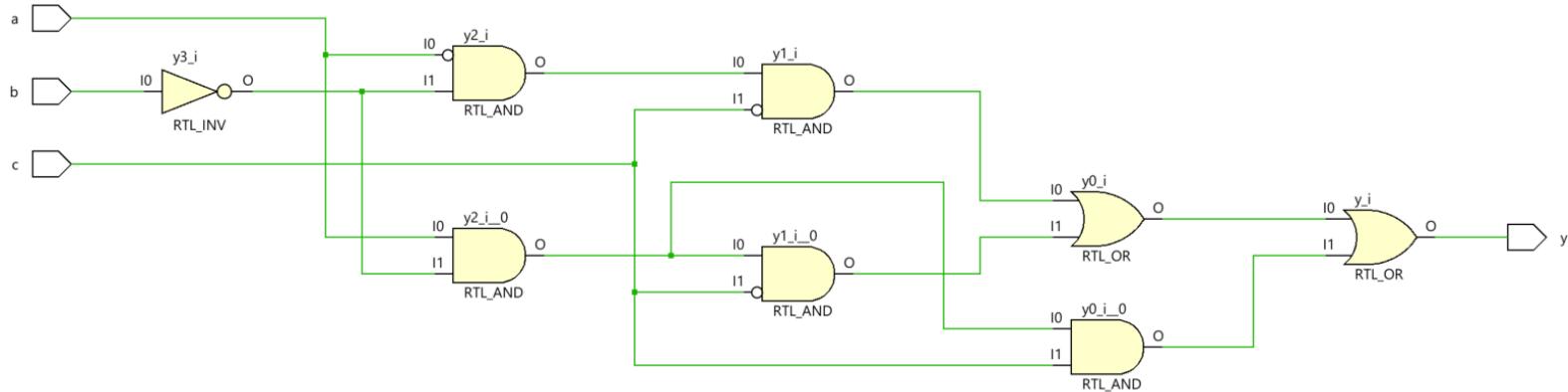
```
module example(input logic a, b, c,  
                output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

$$y = \bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot c$$

Synthesis:

Can be optimised to

$$y = \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot c$$



HDL: *Hardware* Description Language

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

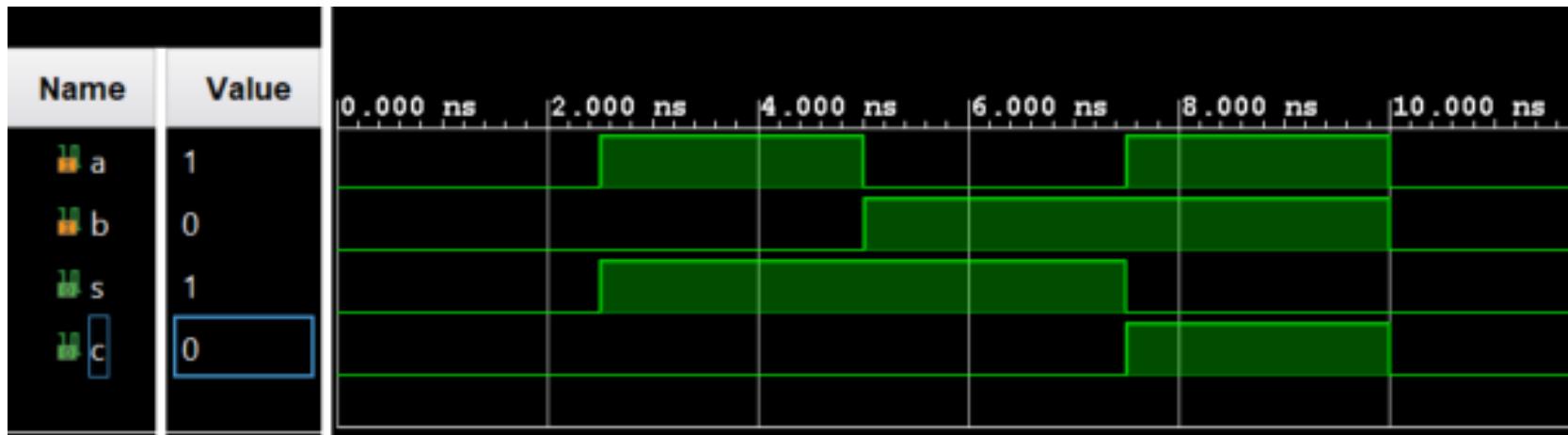
Beware of treating HDL like software and coding without thinking of the hardware.

Half Adder – Simulation

SystemVerilog:

```
module half_adder(input logic a, b,  
                    output logic s, c);  
    assign s = a ^ b; // a xor b  
    assign c = a & b; // a and b  
endmodule
```

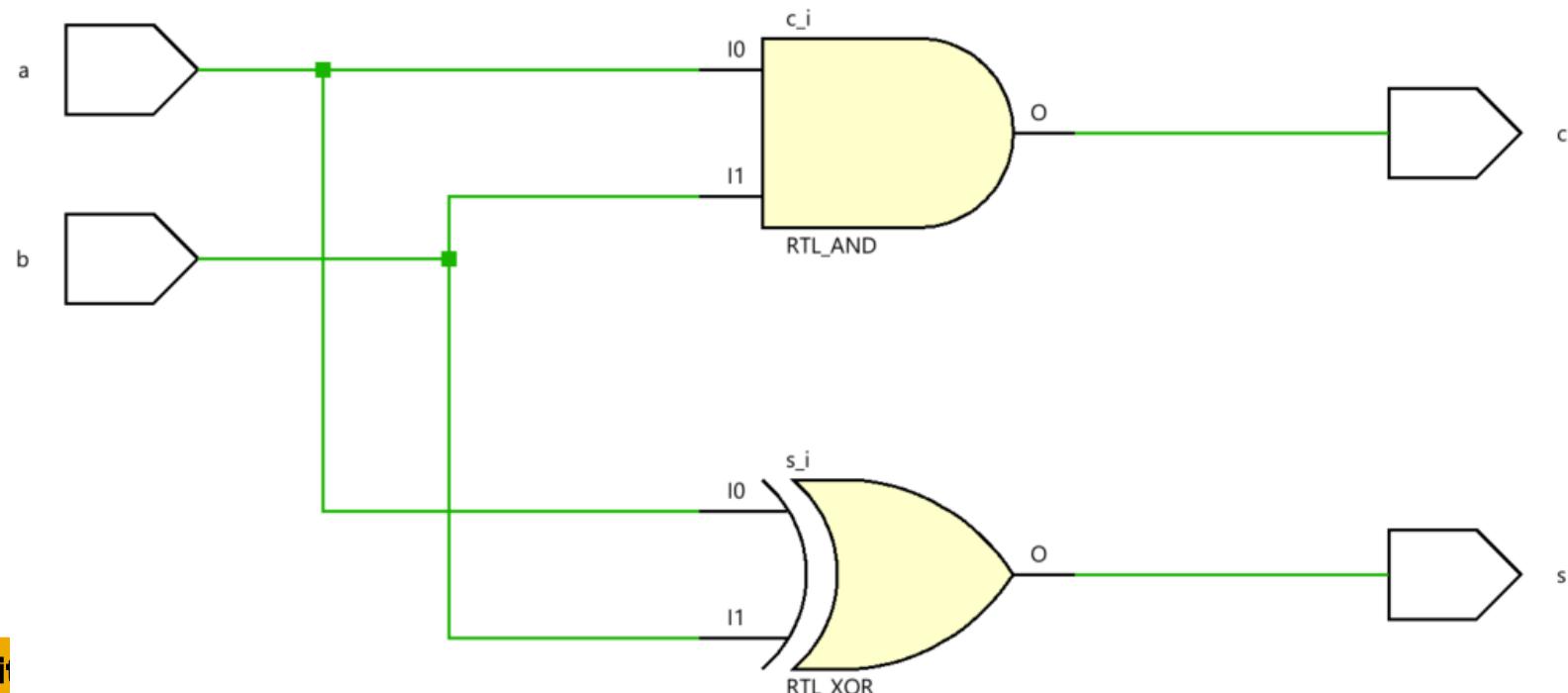
Half Adder Truth Table			
Inputs		Outputs	
a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Half Adder – Synthesis

SystemVerilog:

```
module half_adder(input logic a, b,  
                    output logic s, c);  
  
    assign s = a ^ b; // a xor b  
    assign c = a & b; // a and b  
  
endmodule
```



SystemVerilog logic type

- In SystemVerilog, the logic type is a 4-state data type. The four possible values for a logic variable are:
 - 0:** Logic low (Represents a logical false or a low voltage level).
 - 1:** Logic high (Represents a logical true or a high voltage level).
 - X:** Unknown value (Represents an unknown value. This can occur if the value cannot be determined, such as in the case of contention (multiple drivers driving different values to the same net) or uninitialised variables).
 - Z:** High impedance (This represents high impedance or tristate. This typically means the signal is not actively driven by any source and is floating).
- These four states are particularly useful for hardware modelling, as they provide more information than just binary (0 and 1), allowing for accurate simulation of circuits with uninitialised or floating signals.

SystemVerilog Syntax

- **Case sensitive**
 - **Example:** reset and Reset are not the same signal.
- **No names that start with numbers**
 - **Example:** 2mux is an **invalid** name
- **Whitespace ignored**
- **Comments:**
 - // single line comment
 - /* multiline
comment */

Structural SystemVerilog

```
module and3(input logic a, b, c,  
            output logic y);  
    assign y = a & b & c;  
endmodule
```

```
module inv(input logic a,  
           output logic y);  
    assign y = ~a;  
endmodule
```

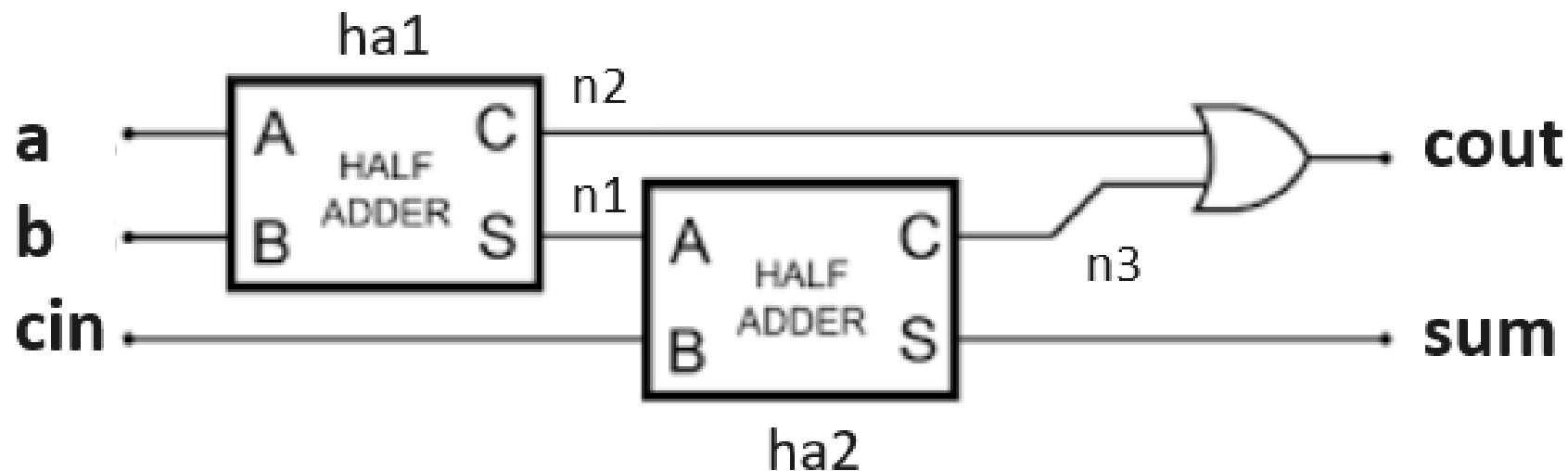
```
module nand3(input logic a, b, c  
             output logic y);  
    logic n1;                      // internal signal  
    and3 andgate(a, b, c, n1);      // instance of and3  
    inv inverter(n1, y);           // instance of inv  
endmodule
```

Structural SystemVerilog

```
module full_adder(input logic a, b, cin,
                   output logic sum, cout);
    logic n1, n2, n3; // internal connection signals

    half_adder ha1(a, b, n1, n2); // half adder instance, ha1
    half_adder ha2(n1, cin, sum, n3); // half adder instance, ha2
    assign cout = n2 | n3;

endmodule
```

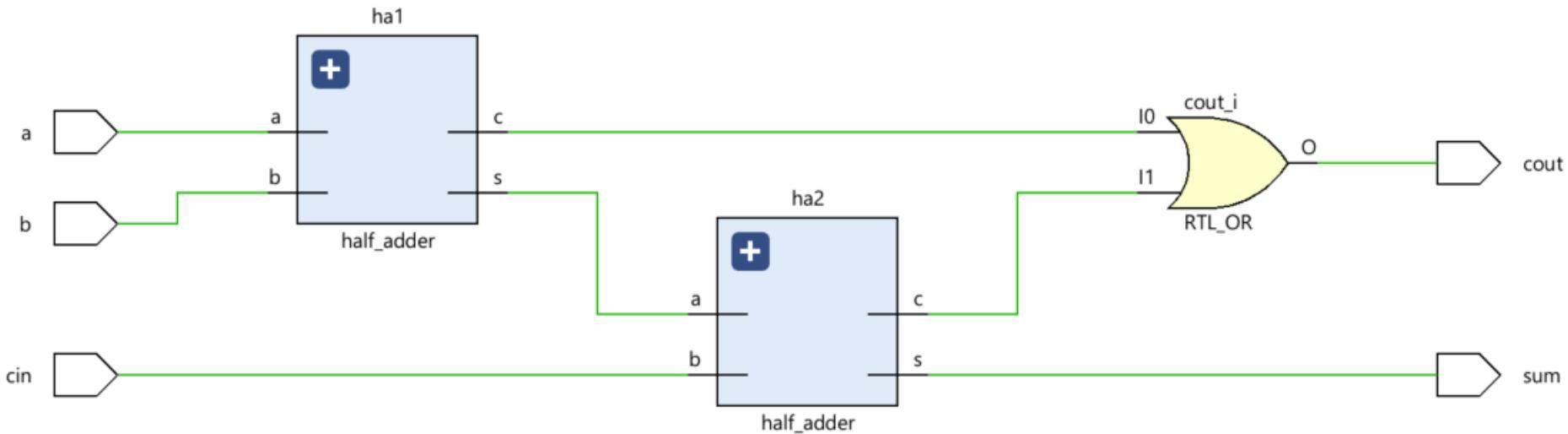


Structural SystemVerilog

```
// named port association for connection of components
module full_adder(input logic a, b, cin,
                    output logic sum, cout);
    logic n1, n2, n3; // internal connection signals

    half_adder ha1(.a(a), .b(b), .s(n1), .c(n2));
    half_adder ha2(.a(n1), .b(cin), .s(sum), .c(n3));
    assign cout = n2 | n3;

endmodule
```



Chapter 4: Hardware Description Languages

Combinational Logic

Bitwise Operators

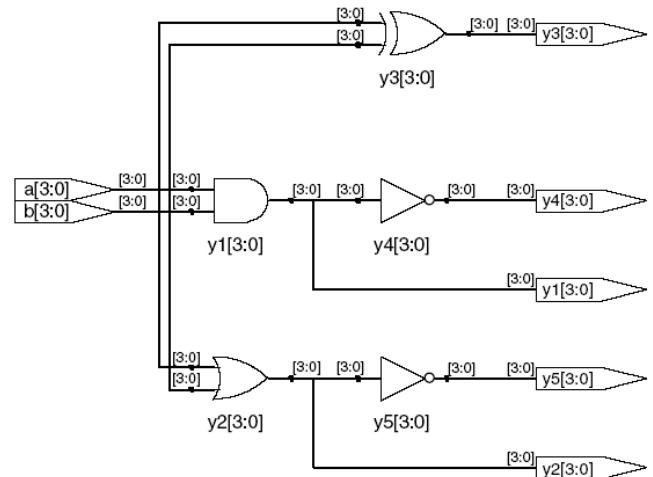
SystemVerilog:

```
module gates(input logic [3:0] a,  
             output logic [3:0] y1, y2, y3, y4, y5);  
    /* Five different two-input logic  
       gates acting on 4 bit busses */  
    assign y1 = a & b;      // AND  
    assign y2 = a | b;      // OR  
    assign y3 = a ^ b;      // XOR  
    assign y4 = ~(a & b);   // NAND  
    assign y5 = ~(a | b);   // NOR  
  
endmodule
```

// single line comment

/*...*/ multiline comment

Synthesis:

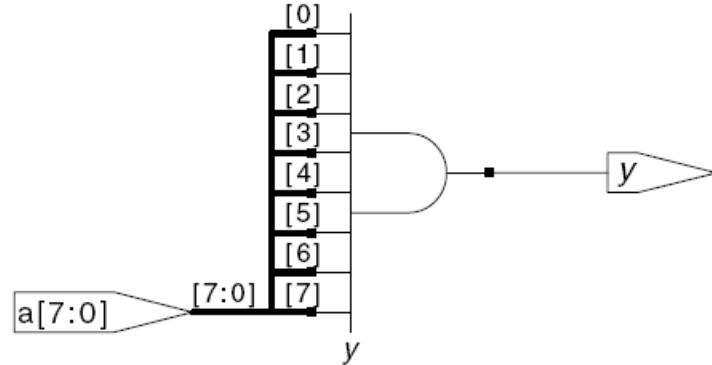


Reduction Operators

SystemVerilog:

```
module and8(input logic [7:0] a,
             output logic      y);
    assign y = &a;
    // &a is much easier to write than
    // assign y = a[7] & a[6] & a[5] & a[4] &
    //           a[3] & a[2] & a[1] & a[0];
endmodule
```

Synthesis:



Six Reduction Operators

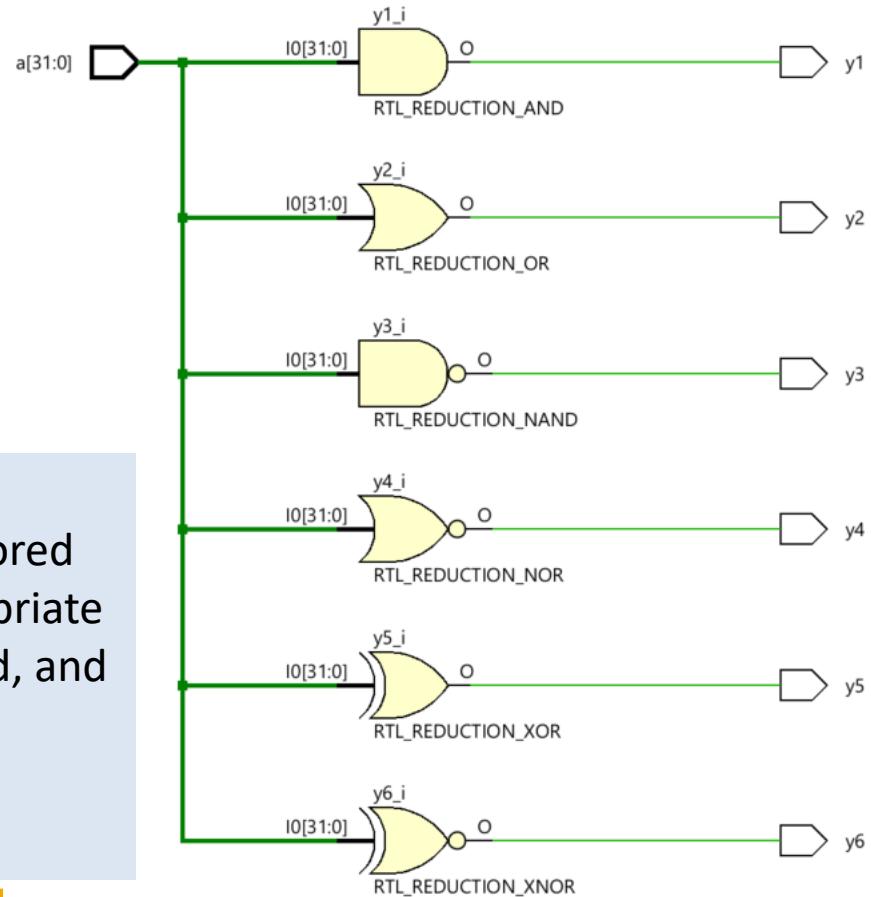
SystemVerilog:

```
module reduction(input logic [31:0] a,
    output logic y1, y2, y3, y4, y5, y6);

    assign y1 = &a; // reduction AND
    assign y2 = |a; // reduction OR
    assign y3 = ~&a; // reduction NAND
    assign y4 = ~|a; // reduction NOR
    assign y5 = ^a; // reduction XOR
    assign y6 = ~^a; // reduction XNOR

endmodule
```

Synthesis:



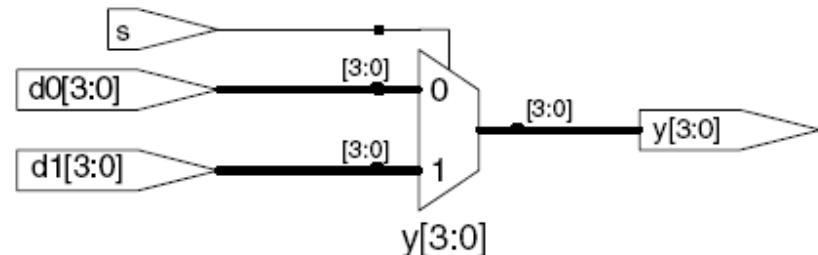
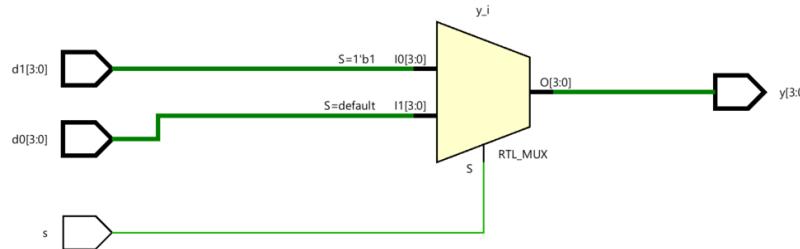
- SystemVerilog has six reduction operators
- A reduction operator accepts a single vectored (multiple bit) operand, performs the appropriate bit-wise reduction on all bits of the operand, and returns a single bit result
- For example, the 32-bits of **a** are **ANDed** together to produce a single bit result **y1**.

Conditional Assignment

SystemVerilog:

```
module mux2(input logic [3:0] d0, d1,  
            input logic s,  
            output logic [3:0] y);  
    assign y = s ? d1 : d0;  
endmodule
```

Synthesis:



? : is also called a *ternary operator* because it operates on 3 inputs: s , d_1 , and d_0 .

Internal Variables

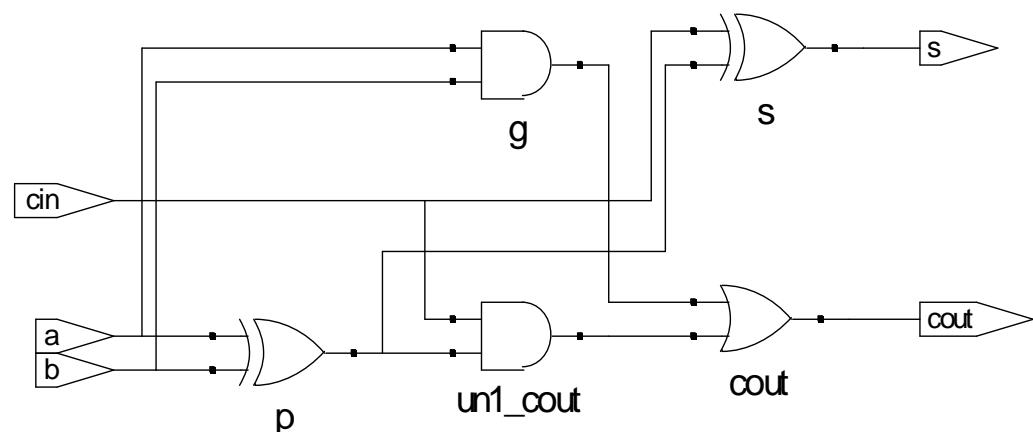
SystemVerilog:

```
module fulladder(input logic a, b, cin,
                  output logic s, cout);
    logic p, g; // internal nodes

    assign p = a ^ b; # propagate carry
    assign g = a & b; # generate carry

    assign s = p ^ cin;
    assign cout = g | (p & cin);
endmodule
```

Synthesis:



Operator Precedence

Highest

\sim	NOT
$*$, $/$, $\%$	mult, div, mod
$+$, $-$	add, sub
$<<$, $>>$	shift
$<<<$, $>>>$	arithmetic shift
$<$, $<=$, $>$, $>=$	comparison
$==$, $!=$	equal, not equal
$\&$, $\sim \&$	AND, NAND
\wedge , $\sim \wedge$	XOR, XNOR
\mid , $\sim \mid$	OR, NOR
$? :$	ternary operator

Lowest

Numbers

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00...0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00...0101010

Bit Manipulations: Example 1

SystemVerilog:

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

If **y** is a 12-bit signal, the above statement produces:

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

Underscores (_) are used for formatting only to make it easier to read. SystemVerilog ignores them.

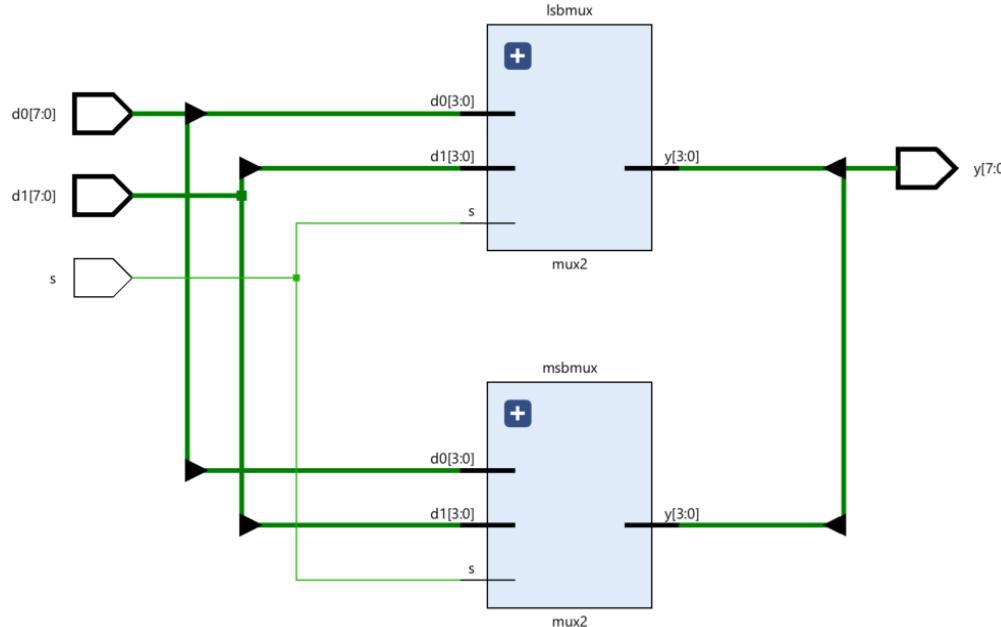
Bit Manipulations: Example 2

SystemVerilog:

```
module mux2_8(input logic [7:0] d0, d1,  
                input logic s,  
                output logic [7:0] y);
```

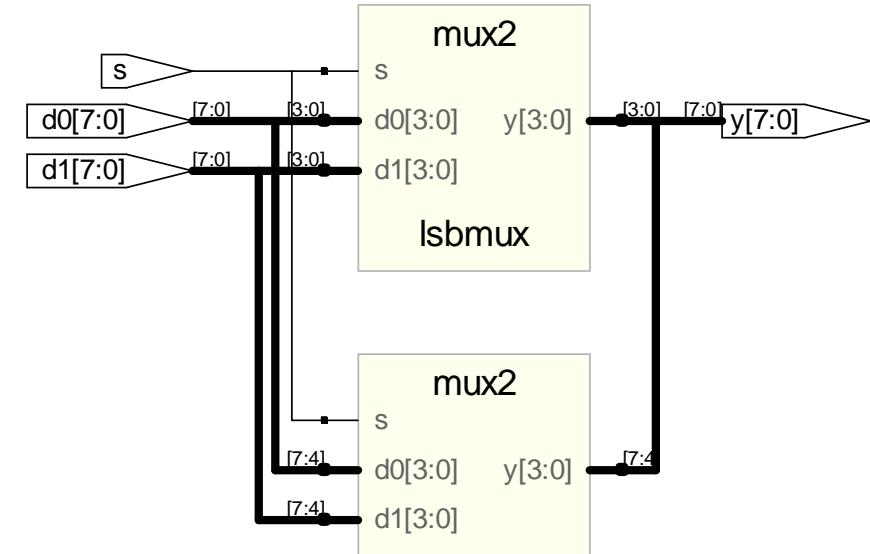
```
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
```

```
endmodule
```



This example shows building an 8-bit 2-to-1 multiplexer using two 4-bit 2-to-1 multiplexers.

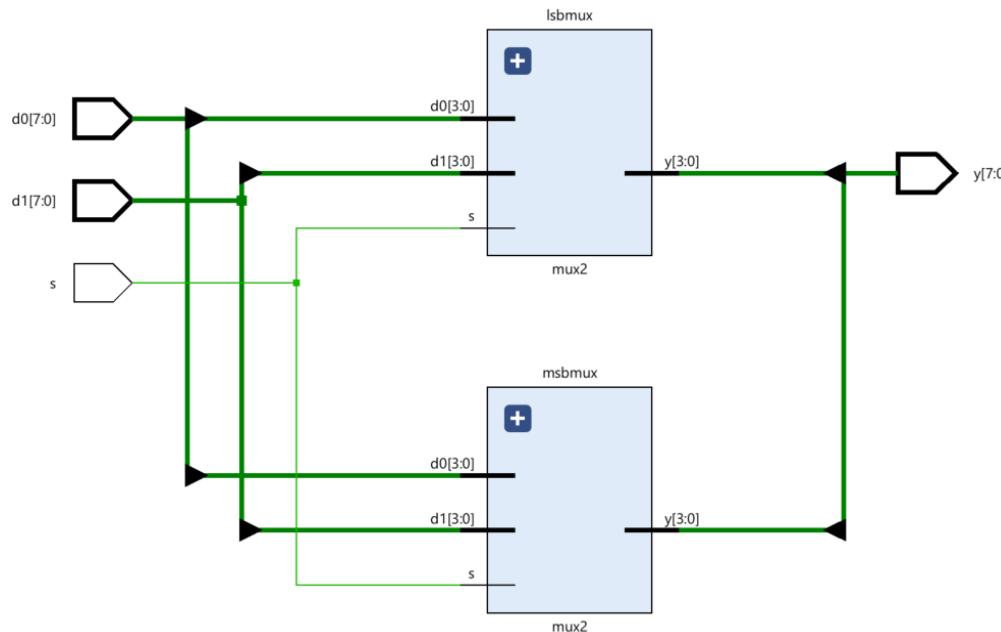
Synthesis:



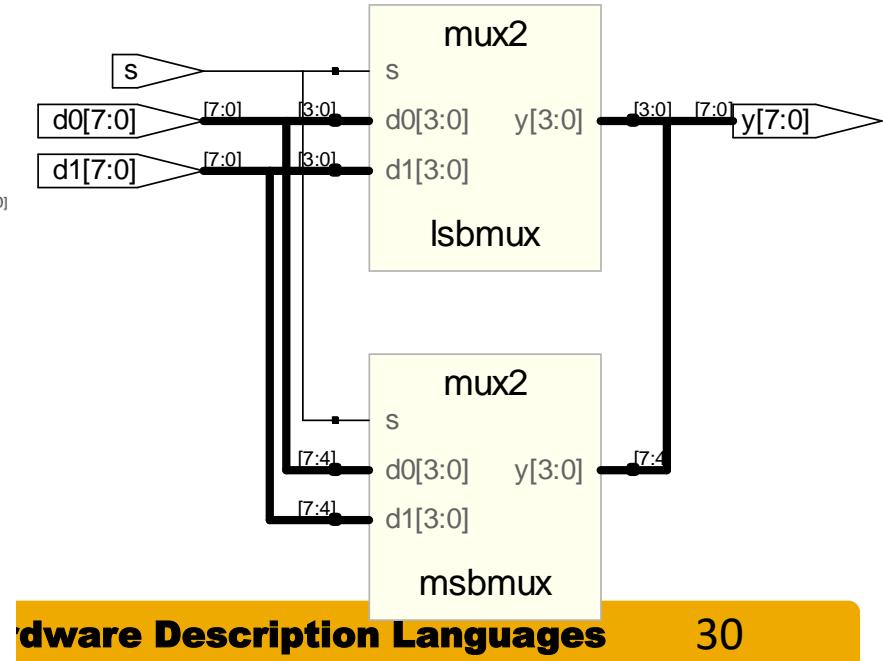
Bit Manipulations: Example 2

SystemVerilog:

```
module mux2_8(input logic [7:0] d0, d1,  
                input logic s,  
                output logic [7:0] y);  
  
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);  
  
endmodule
```



Synthesis:



Chapter 4: Hardware Description Languages

Delays

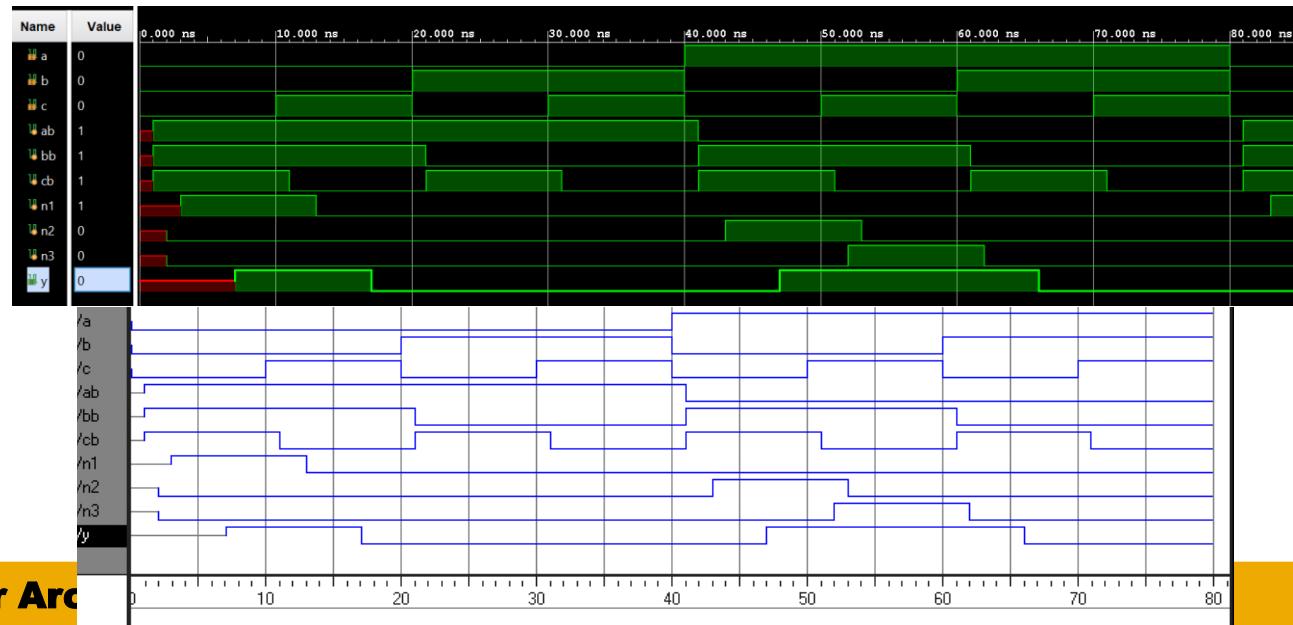
Delays – for simulation only

SystemVerilog:

```
module example(input logic a, b, c,  
               output logic y);  
  
    logic ab, bb, cb, n1, n2, n3;  
  
    assign #1 {ab, bb, cb} = ~{a, b, c}; // 1 ns delay for inverter  
    assign #2 n1 = ab & bb & cb; // 2ns delay for 3-input AND gate  
    assign #2 n2 = a & bb & cb;  
    assign #2 n3 = a & bb & c;  
  
    assign #4 y = n1 | n2 | n3; // 4ns delay for 3-input OR ga  
  
endmodule
```

Simulation

Delays are for
simulation only! They
do not determine the
delay of your
hardware.

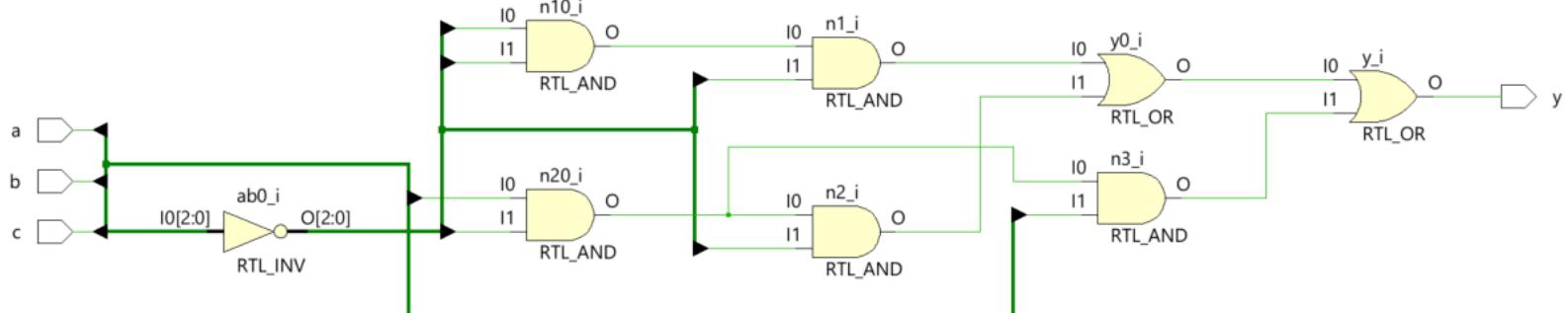


Delays – ignored for synthesis

SystemVerilog:

```
module example(input logic a, b, c,
                output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c}; // 1 ns delay for inverter
    assign #2 n1 = ab & bb & cb; // 2ns delay for 3-input AND gate
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3; // 4ns delay for 3-input OR ga
endmodule
```

Synthesis



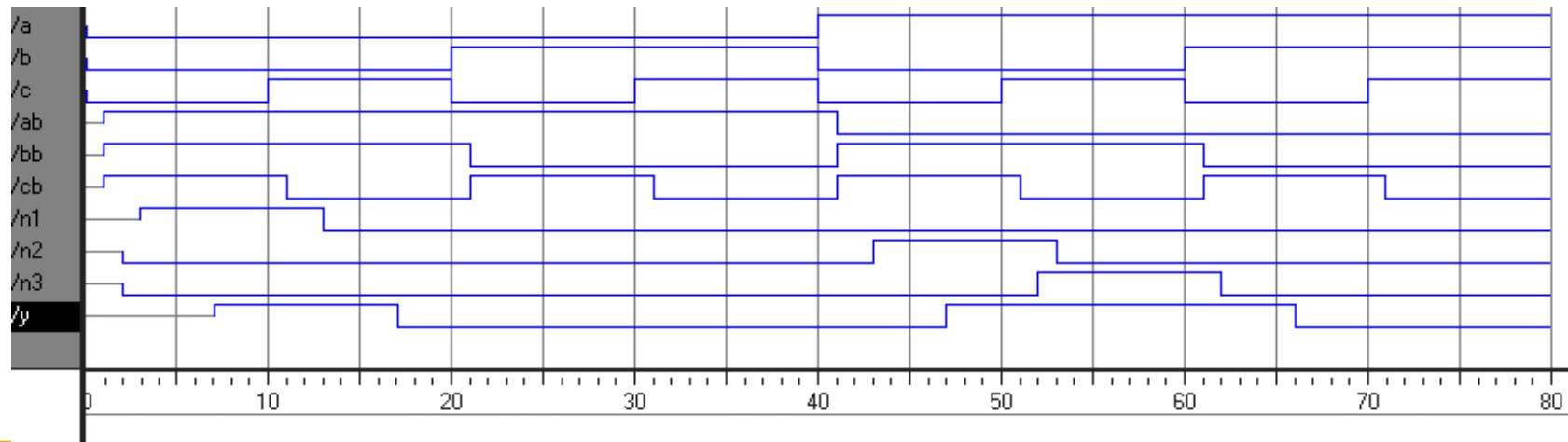
Delays

SystemVerilog:

```
module example(input logic a, b, c,
                output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

Delays are for simulation only! They do not determine the delay of your hardware.

Simulation

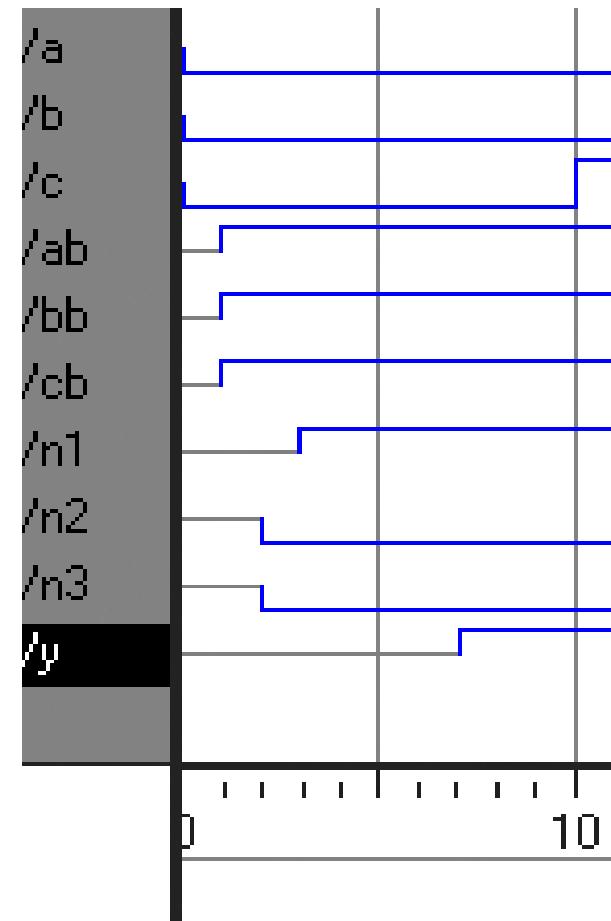


Delays

SystemVerilog:

```
module example(input logic a, b, c,
               output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

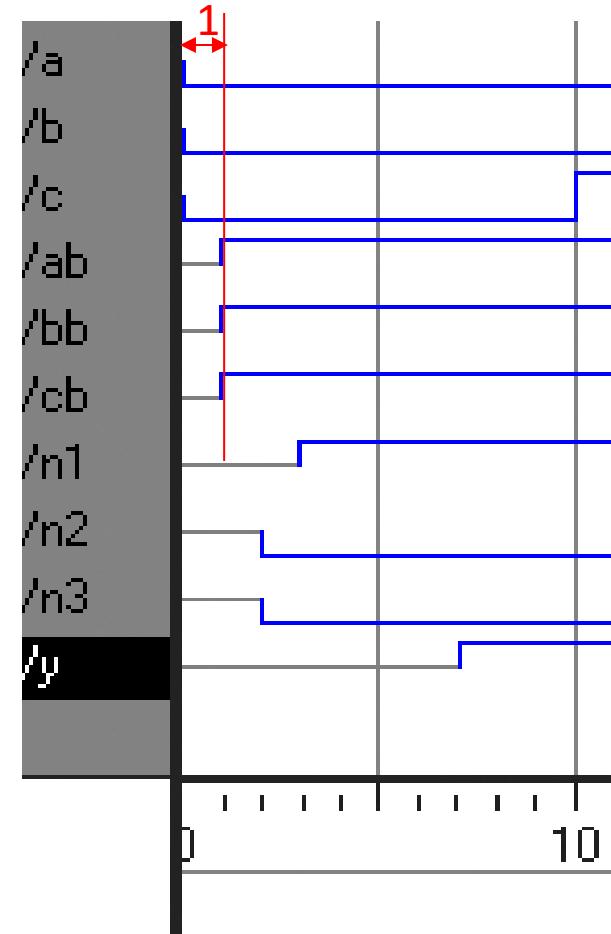


Delays

SystemVerilog:

```
module example(input logic a, b, c,
               output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

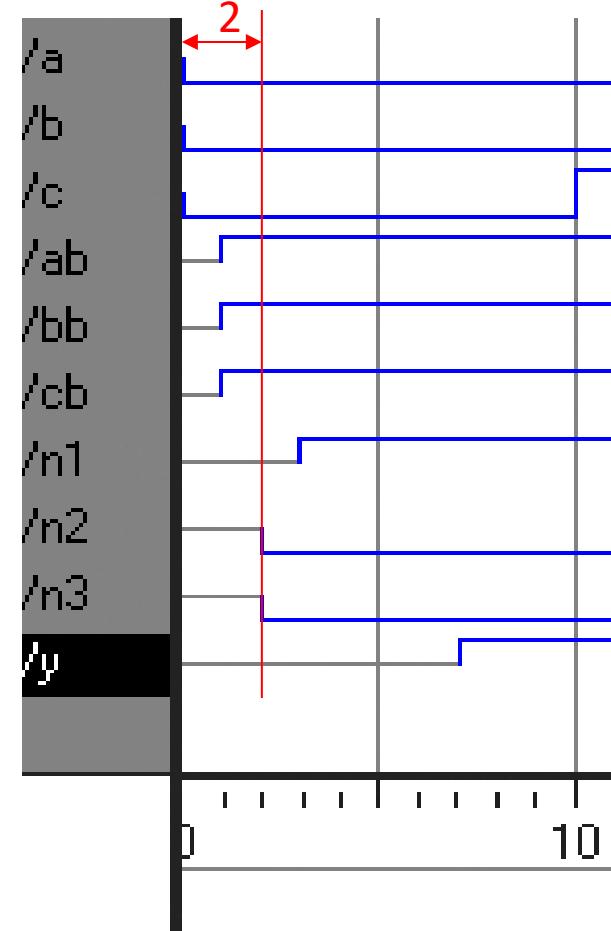


Delays

SystemVerilog:

```
module example(input logic a, b, c,
               output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

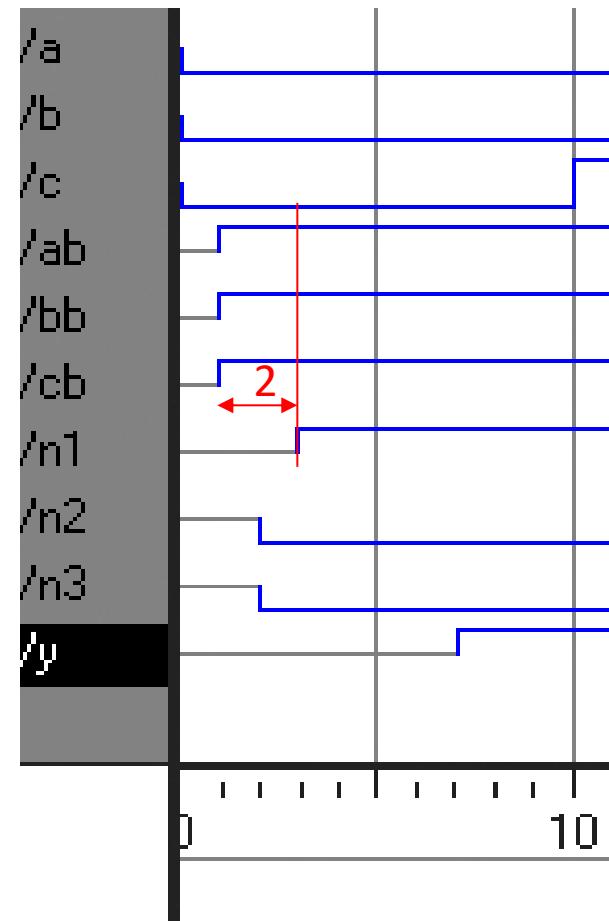


Delays

SystemVerilog:

```
module example(input logic a, b, c,
               output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation

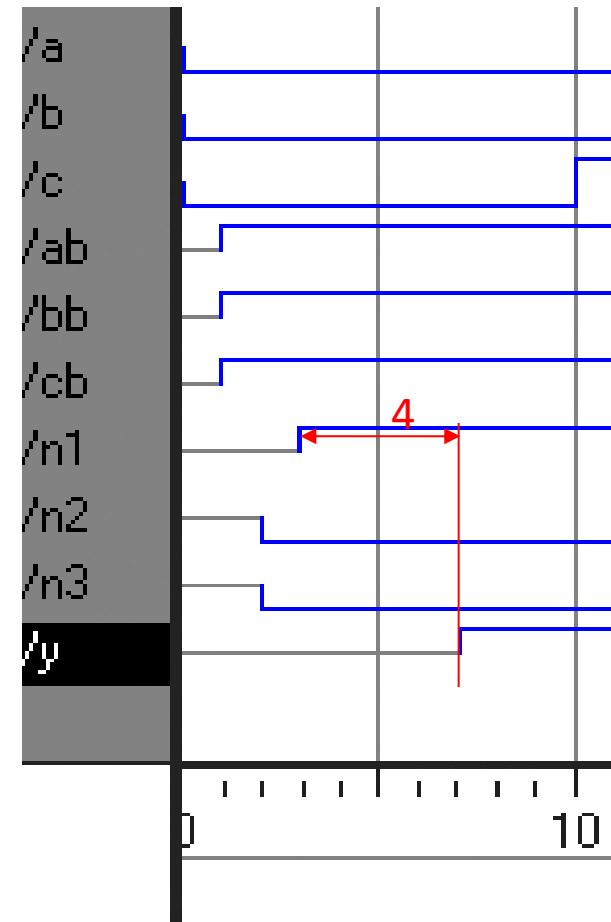


Delays

SystemVerilog:

```
module example(input logic a, b, c,
               output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

Simulation



End of Week2 Slides

Chapter 4: Hardware Description Languages

Sequential Logic

Sequential Logic

- SystemVerilog uses **idioms** to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware
- Use the **idioms**, so the synthesis tool understands your design intention in terms of hardware implementation

always Statement

General Structure:

```
always @ (sensitivity list)  
statement;
```

Whenever an **event** in the sensitivity list occurs, the statement is executed

- An **event** refers to a change in the value of a signal (e.g., a voltage level or a logical state) at a specific point in time.
- Events are used to model and simulate how a digital circuit behaves over time as its inputs and internal states evolve.
- In an event-driven simulation, the simulator tracks and processes events chronologically, updating the values of signals and propagating changes through the circuit whenever an event occurs.
- Event-driven simulation is more efficient, especially for large-scale circuits, because it focuses only on changes (events) rather than simulating the entire system at every time step.

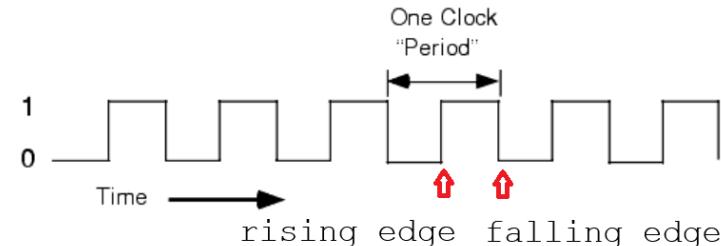
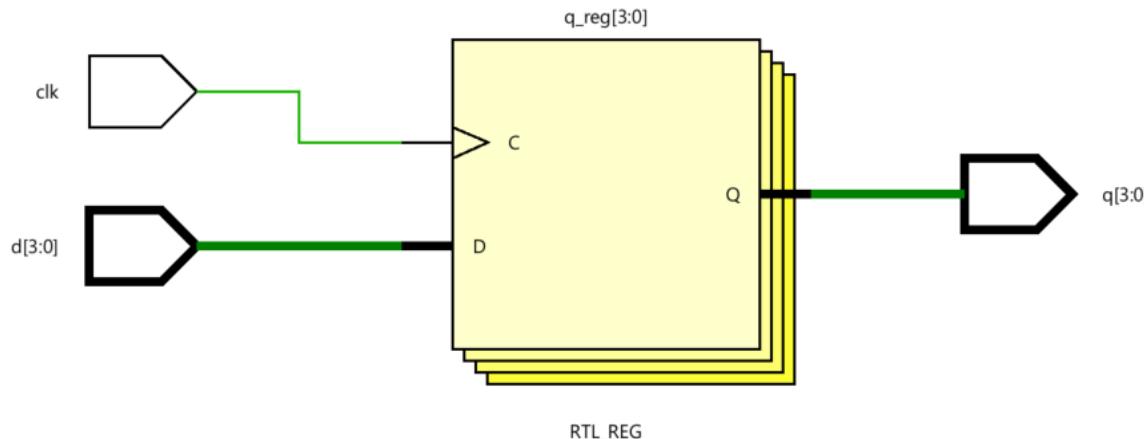
D Flip-Flop

```
module flop(input logic      clk,
             input logic [3:0] d,
             output logic [3:0] q);

    always_ff @ (posedge clk)
        q <= d;                                // pronounced "q gets d"

endmodule
```

Synthesis:



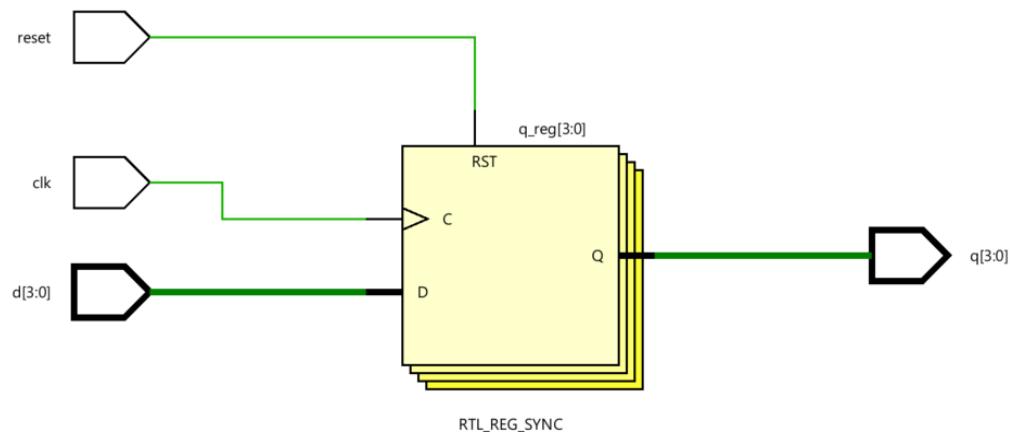
D Flip-Flop – with synchronous reset

```
module flopr(input logic      clk,
              input logic      reset,
              input logic [3:0] d,
              output logic [3:0] q);

    // synchronous reset
    always_ff @(posedge clk)
        if (reset) q <= 4'b0; // At clk rising edge check reset
        else         q <= d;

endmodule
```

Synthesis:



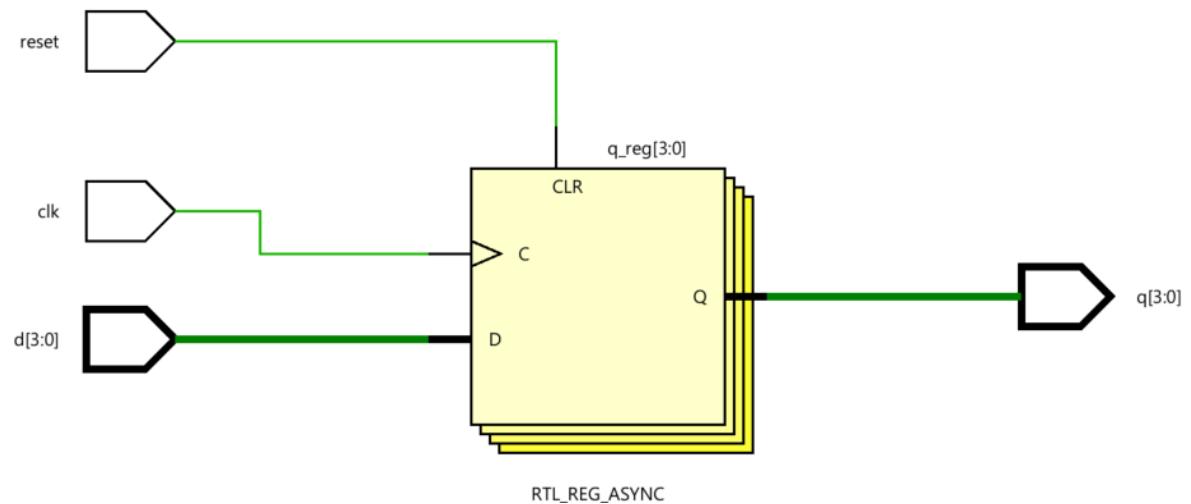
D Flip-Flop – with asynchronous reset

```
module flop_ar(input logic      clk,
                input logic      reset,
                input logic [3:0] d,
                output logic [3:0] q);

    // asynchronous reset
    always_ff @ (posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else         q <= d;

endmodule
```

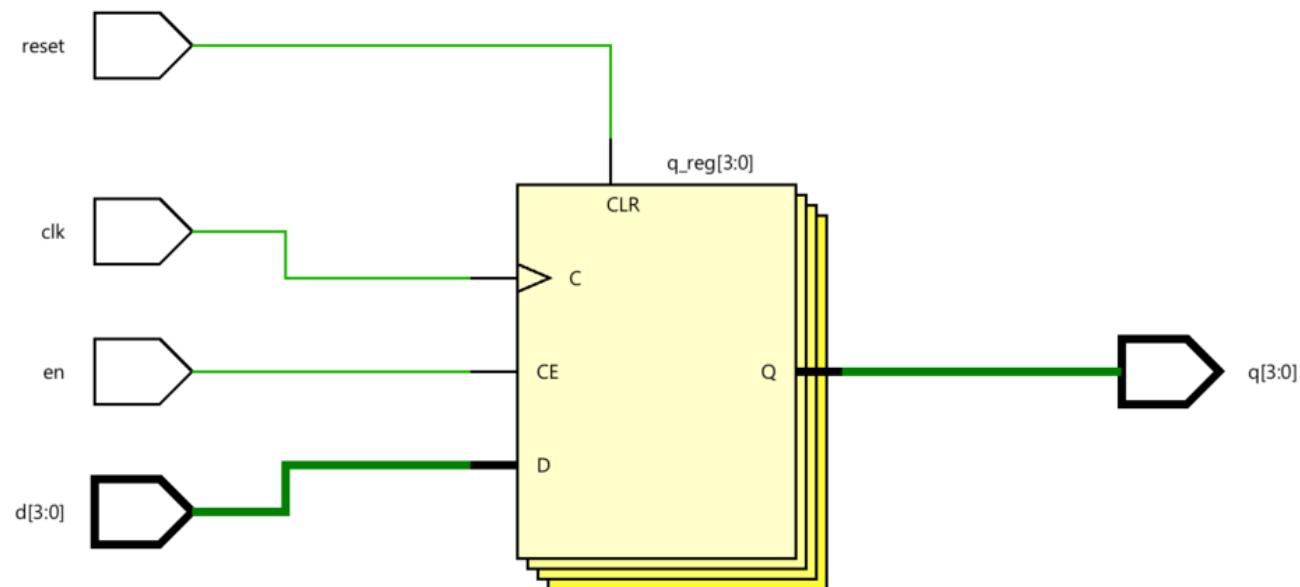
Synthesis:



D Flip-Flop with Enable

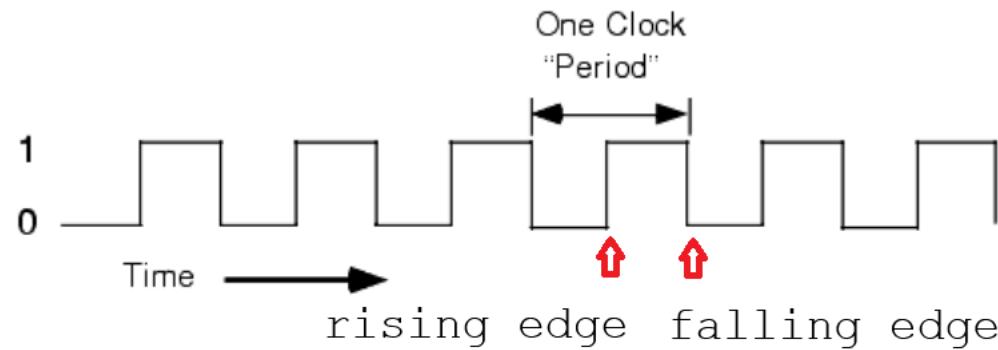
```
module flop_ar_en(input logic      clk,
                   input logic      reset,
                   input logic      en,
                   input logic [3:0] d,
                   output logic [3:0] q);
    // enable and asynchronous reset
    always_ff @ (posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else if (en) q <= d;
endmodule
```

Synthesis:



The always_ff block

- The **always_ff** block models flip-flops.
- Typically triggered on the edges of a clock or reset signal (e.g., posedge clk or negedge reset).
- It ensures that the code within the block is synthesised as a flip-flop or register.
- It enforces certain rules to avoid unintended behaviours, like mixing combinational and sequential logic.



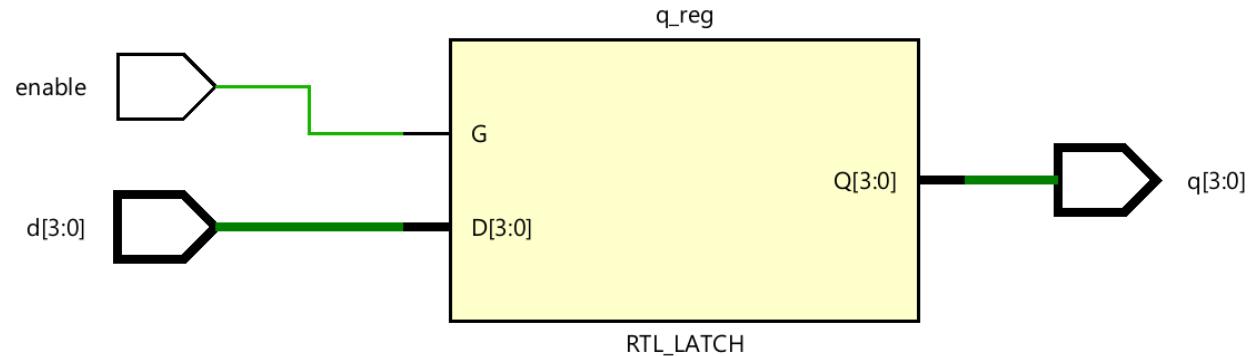
Latch

```
module latch(input logic      enable,
              input logic [3:0] d,
              output logic [3:0] q);

    always_latch
        if (enable) q <= d;

endmodule
```

Synthesis:



Warning: We don't use latches in this text. But you might write code that *inadvertently implies a latch*. Check synthesised hardware – if it has latches in it that you didn't intend to create, there's an **error**.

The `always_latch` block

- The `always_latch` block describes latches.
- A latch is a level-sensitive logic storage element
 - it is transparent when enabled, and
 - It holds its value when disabled.
- The `always_latch` block is triggered by changes in the sensitive list, typically an enable signal.
- SystemVerilog tools can verify that the code inside the block correctly describes a latch and doesn't mix in combinational or flip-flop logic.

General Structure:

```
always @ (sensitivity list)  
statement;
```

- **Flip-flop:** always_ff
 - posedge // rising edge
 - negedge // falling edge
- **Latch:** always_latch **(don't use)**

Chapter 4: Hardware Description Languages

**Combinational Logic
using always**

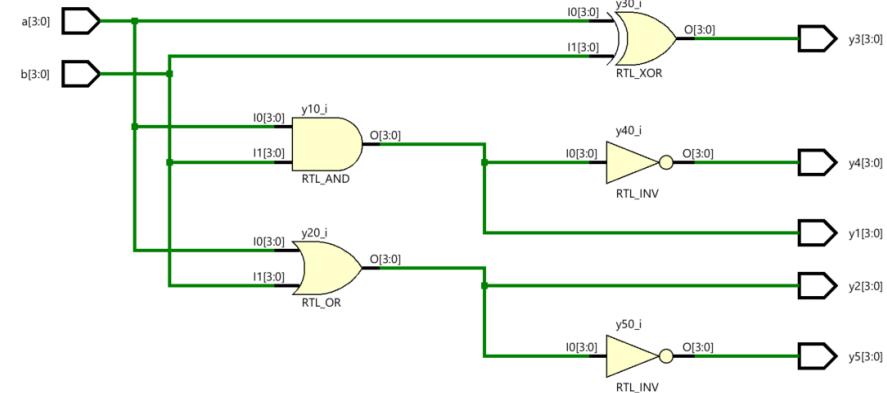
if/else and case/casez

Statements that must be inside always statements:

- if / else
- case, casez

Combinational Logic using always

```
// combinational logic using an always statement
module gates(input logic [3:0] a, b,
              output logic [3:0] y1, y2, y3, y4, y5);
  always_comb // need begin/end because there is
  begin // more than one statement in always
    y1 = a & b; // AND
    y2 = a | b; // OR
    y3 = a ^ b; // XOR
    y4 = ~(a & b); // NAND
    y5 = ~(a | b); // NOR
  end
endmodule
```

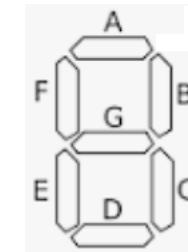


This hardware could be described with **assign statements using fewer lines** of code, so it's better to use **assign** statements in this case.

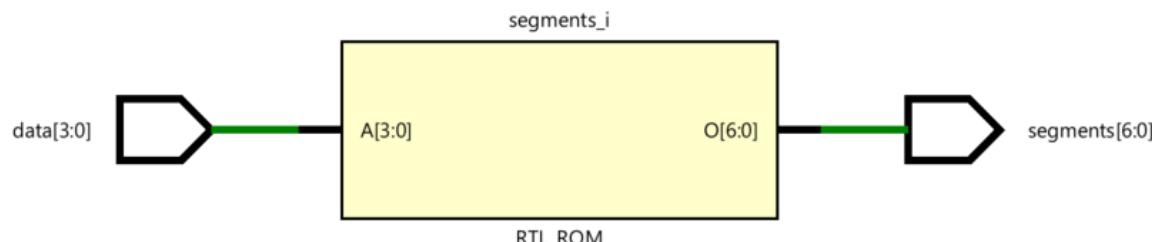
Combinational Logic using case

```
module sevenseg(input logic [3:0] data,
                  output logic [6:0] segments);

  always_comb
    case (data)
      // ABC_DEFG
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_0011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```



Don't
forget!



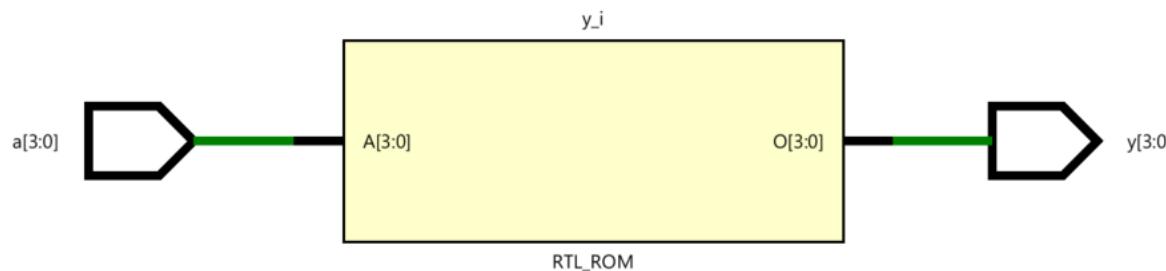
Combinational Logic using case

- case statement implies combinational logic
only if all possible input combinations described
- Remember to use **default** statement

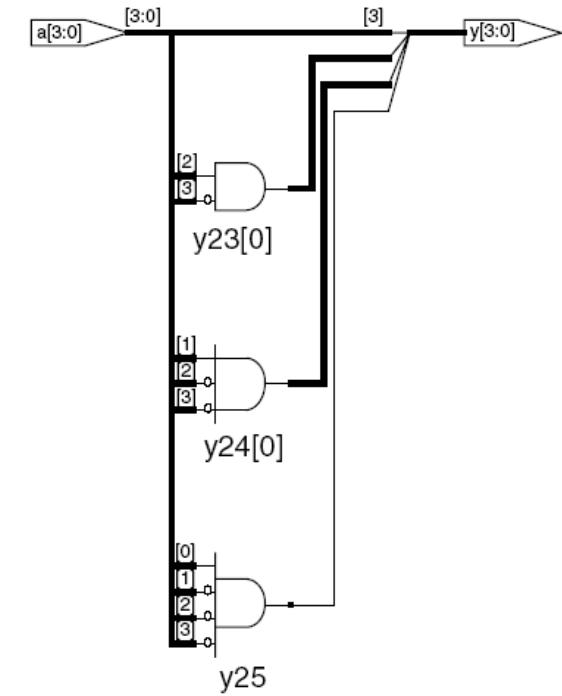
Combinational Logic using casez

```
module priority_casez(input logic [3:0] a,
                      output logic [3:0] y);

  always_comb
  casez(a)
    4'b1????: y = 4'b1000; // ? = don't care
    4'b01???: y = 4'b0100;
    4'b001?: y = 4'b0010;
    4'b0001: y = 4'b0001;
    default: y = 4'b0000;
  endcase
endmodule
```

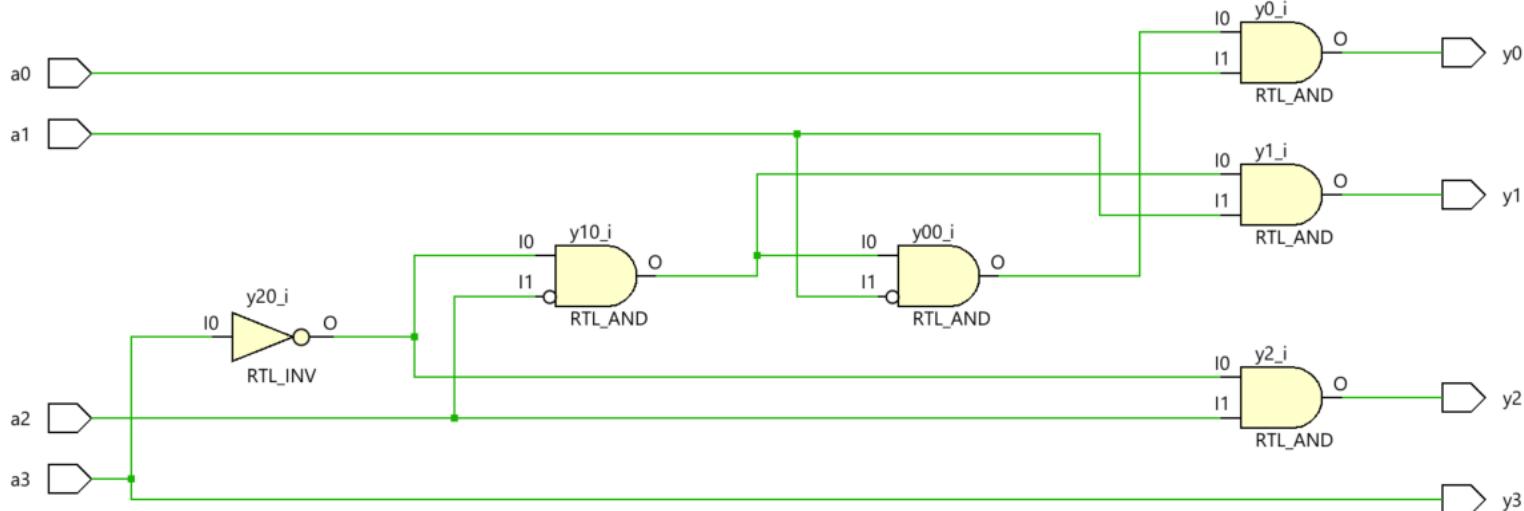


Synthesis:



Priority Encoder

```
module priority_encoder(input logic a3, a2, a1, a0,  
                      output logic y3, y2, y1, y0);  
  
    assign y3 = a3;  
    assign y2 = ~a3 & a2;  
    assign y1 = ~a3 & ~a2 & a1;  
    assign y0 = ~a3 & ~a2 & ~a1 & a0;  
  
endmodule
```



Different always blocks

Block Type	Used For	Sensitivity	Synthesis Check
always	General-purpose logic	User-defined, manual sensitivity list	None
always_comb	Pure combinational logic	Automatically inferred sensitivity	Checks for combinational logic
always_ff	Sequential logic (flip-flops)	Edge-triggered (e.g., clock)	Checks for sequential logic (flip-flops)
always_latch	Level-sensitive latches	Level-sensitive (e.g., enable signal)	Checks for latch behavior

In summary, `always_comb`, `always_ff`, and `always_latch` are specialized forms of the general `always` block, which help to ensure the correct description of specific types of logic and avoid synthesis issues.

case vs casez

Aspect	case	casez
Matching Type	Exact bit-by-bit comparison	Allows z and ? to represent don't-care bits
Don't-care Handling	Not supported	Supported via z or ? wildcards
Use Cases	Precise comparison without wildcards	Useful when ignoring specific bits in comparisons

Summary:

- case is used for exact matching where every bit matters.
- casez is used when some bits are irrelevant or "don't care" in the comparison, allowing for flexible matching using wildcards (z or ?).

Chapter 4: Hardware Description Languages

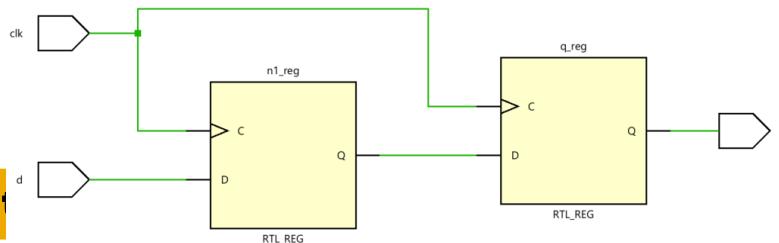
**Blocking and
Nonblocking
Assignments**

Blocking vs. Nonblocking Assignment

- **<=** is **nonblocking** assignment
 - Occurs simultaneously with others
- **=** is **blocking** assignment
 - Occurs in order it appears in file

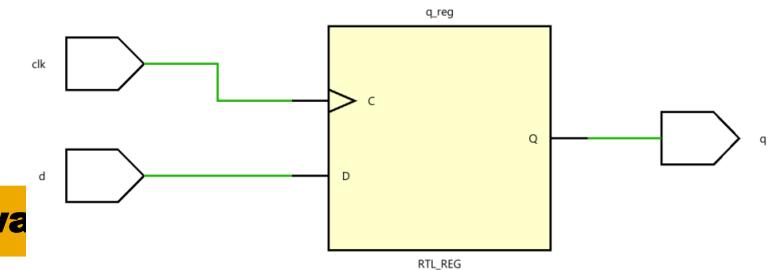
```
// Good synchroniser using
// nonblocking assignments
module syncgood(input logic clk,
                  input logic d,
                  output logic q);

  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```



```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic clk,
                input logic d,
                output logic q);

  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```



Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @ (posedge clk)` and nonblocking assignments (`<=`)

```
always_ff @ (posedge clk)  
q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (`assign...`)

```
assign y = a & b;
```

- **More complicated combinational logic:** use `always_comb` and blocking assignments (`=`)
- Assign a signal in **only one** always statement or continuous assignment statement.

Chapter 4: Hardware Description Languages

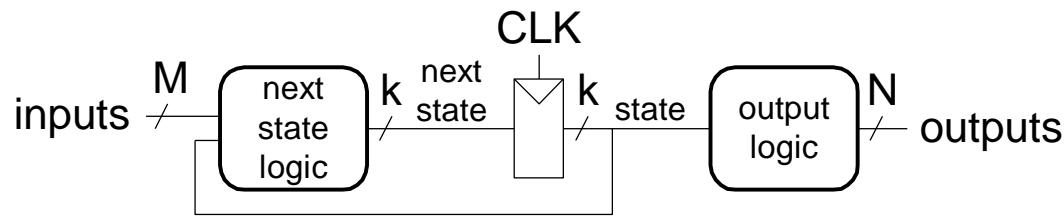
Finite State Machines

Finite State Machines

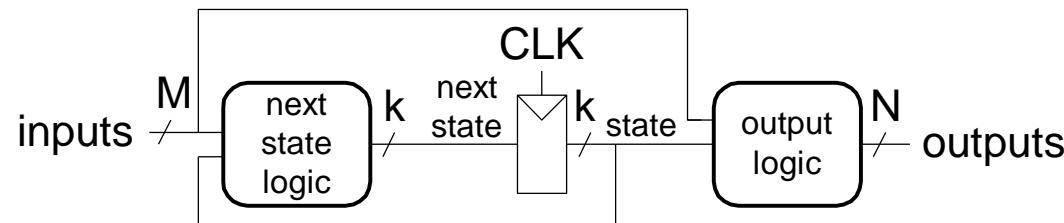
- **Three blocks:**

- next state logic
- state register
- output logic

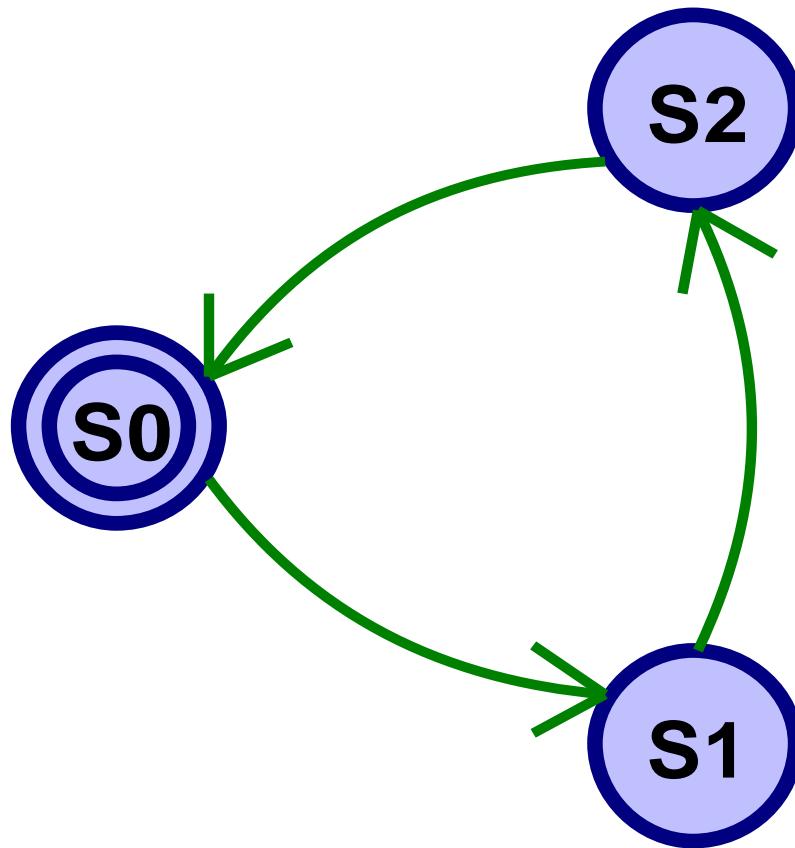
Moore FSM



Mealy FSM



FSM Example 1: Divide by 3



The double circle indicates the reset state

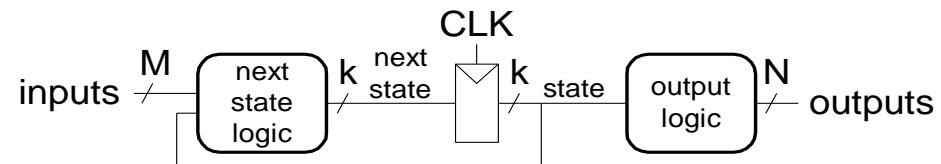
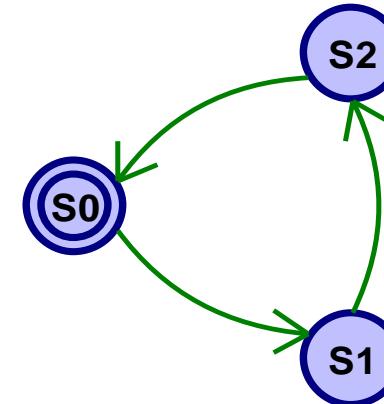
Divide by 3 FSM in SystemVerilog

```
module divideby3FSM(input logic clk, reset, output logic q);

    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype state, nextstate;

    // state register
    always_ff @(posedge clk, posedge reset)
        if (reset) state <= S0;
        else         state <= nextstate;

    // next state logic
    always_comb
        case (state)
            S0:      nextstate = S1;
            S1:      nextstate = S2;
            S2:      nextstate = S0;
            default: nextstate = S0;
        endcase
    // output logic
    assign q = (state == S0);
endmodule
```

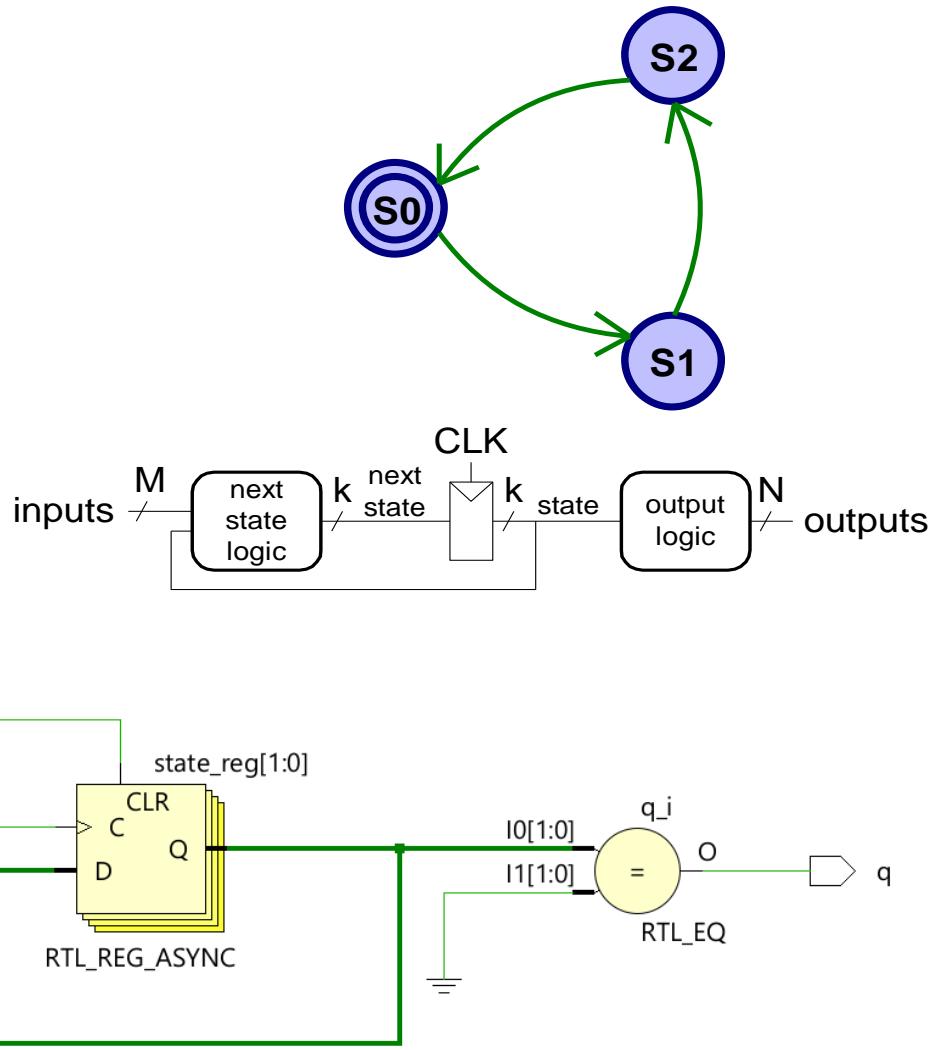


Divide by 3 FSM in SystemVerilog

```

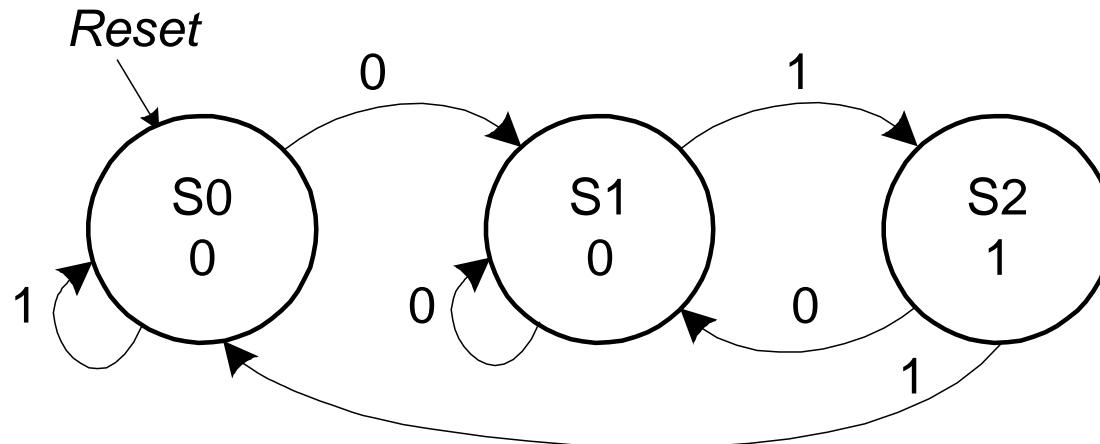
module divideby3FSM(input logic clk, reset, output logic q);
    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype state, nextstate;
    // state register
    always_ff @(posedge clk, posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;
    // next state logic
    always_comb
        case (state)
            S0:      nextstate = S1;
            S1:      nextstate = S2;
            S2:      nextstate = S0;
            default: nextstate = S0;
        endcase
    // output logic
    assign q = (state == S0);
endmodule

```



FSM Example 2: Sequence Detector

Moore FSM

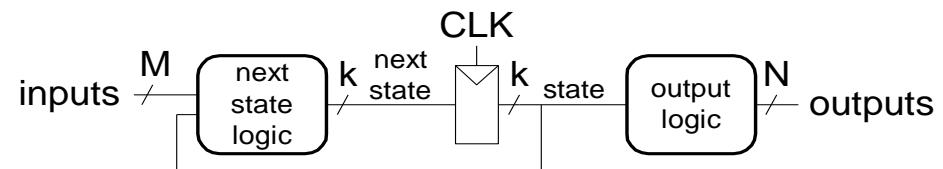
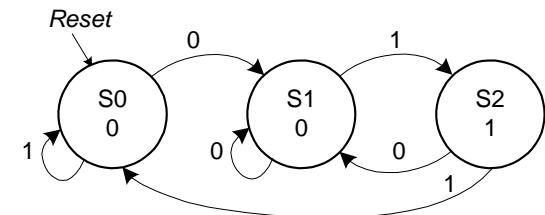


In a Moore FSM, the current state only determines the outputs.

Sequence Detector FSM: Moore

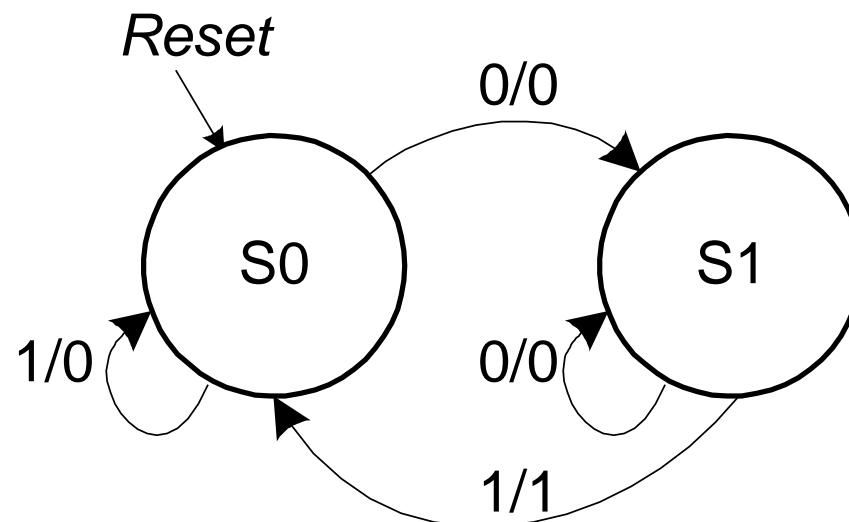
```
module seqDetectMoore(input logic clk, reset, a, output logic smile);
    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype state, nextstate;
    // state register
    always_ff @(posedge clk, posedge reset)
        if (reset) state <= S0;
        else         state <= nextstate;
    // next state logic
    always_comb
        case (state)
            S0:      if (a) nextstate = S0;
                      else     nextstate = S1;
            S1:      if (a) nextstate = S2;
                      else     nextstate = S1;
            S2:      if (a) nextstate = S0;
                      else     nextstate = S1;
            default:   nextstate = S0;
        endcase
    // output logic
    assign smile = (state == S2);
endmodule
```

Moore FSM



FSM Example 3: Sequence Detector

Mealy FSM



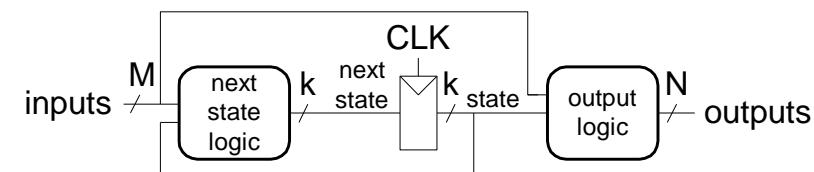
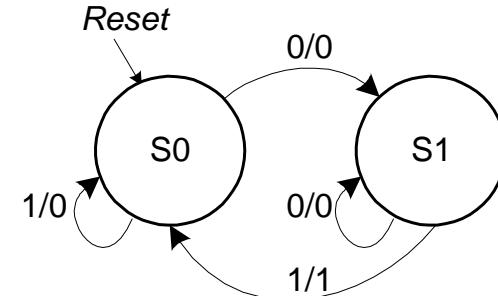
Sequence Detector FSM: Mealy

```
module seqDetectMealy(input logic clk, reset, a, output logic smile);
    typedef enum logic {S0, S1} statetype;
    statetype state, nextstate;

    // state register
    always_ff @(posedge clk, posedge reset)
        if (reset) state <= S0;
        else        state <= nextstate;

    // next state and output logic
    always_comb begin
        smile = 1'b0;
        case (state)
            S0:      if (a)    nextstate = S0;
                      else    nextstate = S1;
            S1:      if (a) begin
                      nextstate = S0;
                      smile = 1'b1;
                  end
                      else    nextstate = S1;
            default: nextstate = S0;
        endcase
    end
endmodule
```

Mealy FSM



Moore vs Mealy FSMs

Key Differences Between Moore and Mealy FSMs

Aspect	Moore FSM	Mealy FSM
Output Depends On	Current state only	Current state and current input
Output Change	Only on state transitions (clock edge)	Can change immediately with input
Output Logic	Simpler (state-dependent only)	More complex (state and input-dependent)
Response to Input	Slower (output changes after state transition)	Faster (output changes in the same clock cycle)
Simplicity	Easier to design and debug	More complex, requires careful design
Typical Usage	Applications where output changes with state transitions	Applications where quick response to input is needed

Summary:

- Moore FSMs have outputs that depend solely on the current state, resulting in simpler design and stable outputs that only change with state transitions.
- Mealy FSMs have outputs that depend on both the current state and the input, allowing for quicker responses to input changes, but leading to more complex output logic.

Chapter 4: Hardware Description Languages

Parameterized Modules

Parameterized Module

2:1 mux:

```
module mux2
  #(parameter width = 8) // name and default value
  (input logic [width-1:0] d0, d1,
   input logic           s,
   output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
mux2 #(12) lowmux(d0, d1, s, out);
```

Chapter 4: Hardware Description Languages

Testbenches

Testbenches

- HDL that **tests another module**: *device under test* (dut)
- **Not synthesizable**
- **Types:**
 - Simple
 - Self-checking
 - Self-checking with testvectors

Testbenches

- Write SystemVerilog code to implement the following function in hardware. Name the module sillyfunction.

$$y = \overline{bc} + ab$$

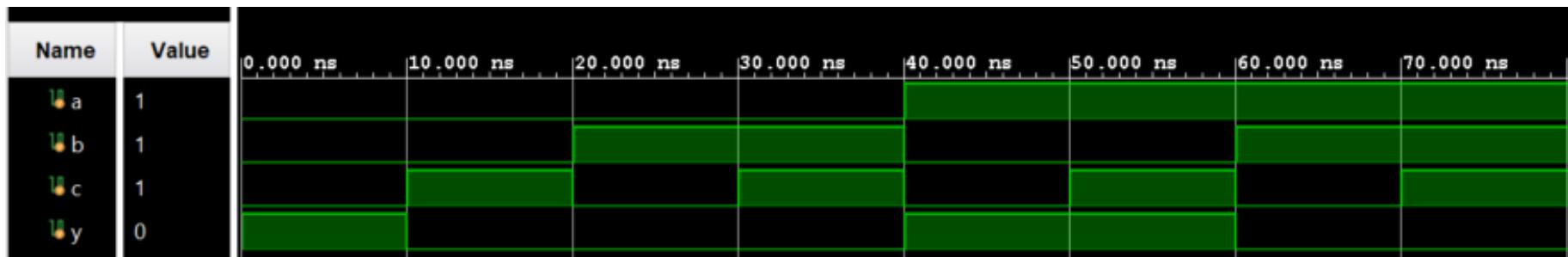
```
module sillyfunction(input logic a, b, c,
                      output logic y);
    assign y = ~b & ~c | a & ~b;
endmodule
```

Testbench 1: Simple Testbench

```
module testbench1();
    logic a, b, c;
    logic y;
    // instantiate device under test
    sillyfunction dut(a, b, c, y);
    // apply inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;
        a = 1; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;
    end
endmodule
```

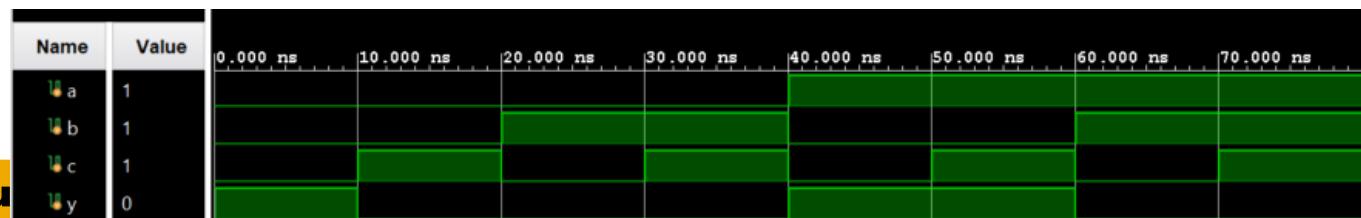
Testbench 1: Simple Testbench

```
module testbench1();
    logic a, b, c;
    logic y;
    // instantiate device under test
    sillyfunction dut(a, b, c, y);
    // apply inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;
        a = 1; b = 0; c = 0; #10;
        c = 1; #10;
        b = 1; c = 0; #10;
        c = 1; #10;
    end
endmodule
```



Testbench 2: Self-Checking Testbench

```
module testbench2();
    logic a, b, c;
    logic y;
    sillyfunction dut(a, b, c, y); // instantiate dut
    initial begin // apply inputs, check results one at a time
        a = 0; b = 0; c = 0; #10;
        if (y !== 1) $display("000 failed.");
        c = 1; #10;
        if (y !== 0) $display("001 failed.");
        b = 1; c = 0; #10;
        if (y !== 0) $display("010 failed.");
        c = 1; #10;
        if (y !== 0) $display("011 failed.");
        a = 1; b = 0; c = 0; #10;
        if (y !== 1) $display("100 failed.");
        c = 1; #10;
        if (y !== 1) $display("101 failed.");
        b = 1; c = 0; #10;
        if (y !== 0) $display("110 failed.");
        c = 1; #10;
        if (y !== 0) $display("111 failed.");
    end
endmodule
```

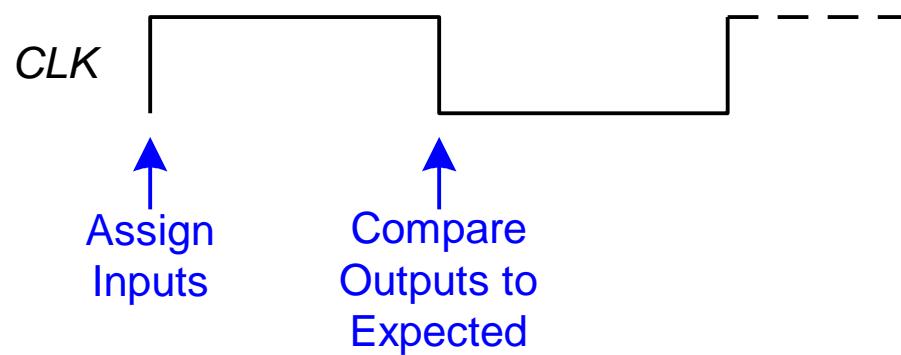


Testbench 3: Testbench w/ Testvectors

- Testvector file: inputs and expected outputs
- Testbench:
 1. Generate clock for assigning inputs, reading outputs
 2. Read testvectors file into array
 3. Assign inputs, expected outputs
 4. Compare outputs with expected outputs and report errors

Testbench 3: Testbench w/ Testvectors

- **Testbench clock:**
 - Assign **inputs** (on **rising edge**).
 - Compare **outputs** with expected outputs (**on falling edge**).

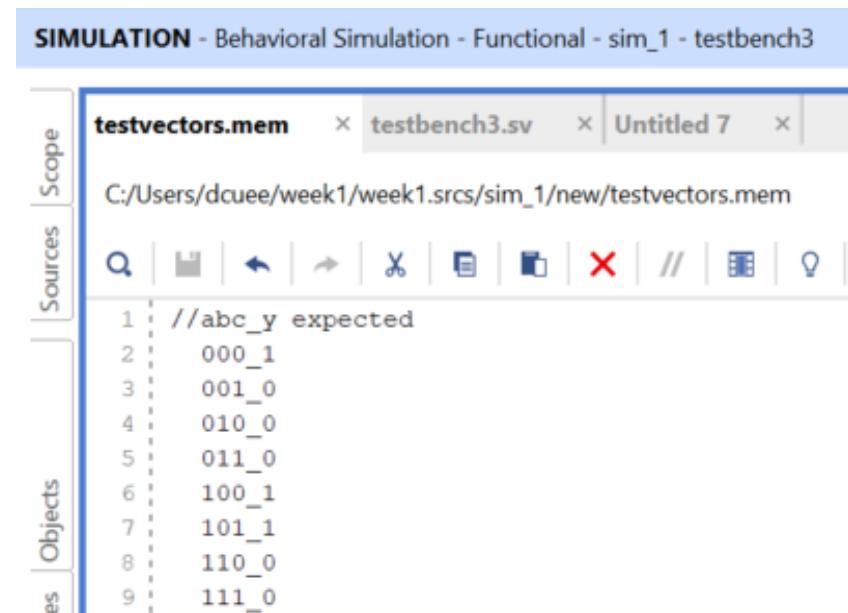


- Testbench clock **also used as clock** for synchronous sequential circuits

Testbench 3: Testvectors File

- **File:** testvectors.mem
- **contains vectors of abc_yexpected**

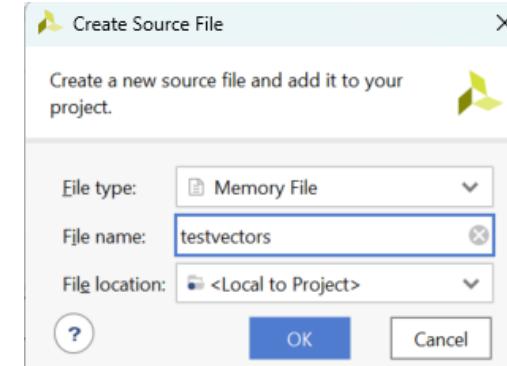
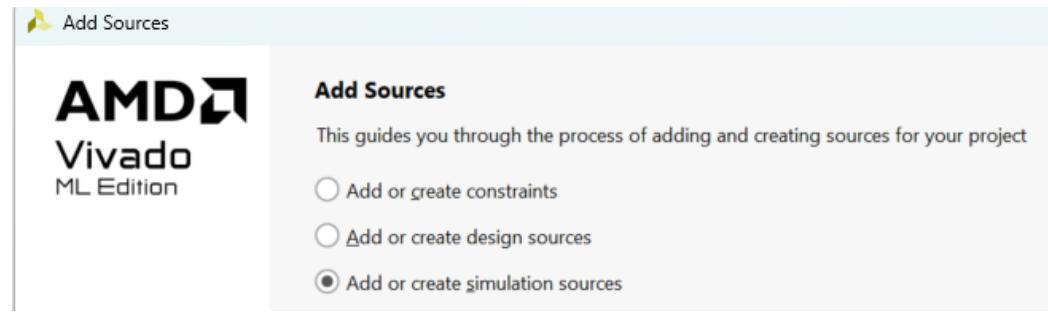
```
//abc_y expected  
000_1  
001_0  
010_0  
011_0  
100_1  
101_1  
110_0  
111_0
```



The screenshot shows a software interface for behavioral simulation. The title bar reads "SIMULATION - Behavioral Simulation - Functional - sim_1 - testbench3". Below the title bar, there are tabs for "testvectors.mem", "testbench3.sv", and "Untitled 7". The "testvectors.mem" tab is active. The file path "C:/Users/dcuee/week1/week1.srccs/sim_1/new/testvectors.mem" is displayed. The main area shows the contents of the file:

```
1 //abc_y expected  
2 000_1  
3 001_0  
4 010_0  
5 011_0  
6 100_1  
7 101_1  
8 110_0  
9 111_0
```

Testbench 3: Create Testvectors File



SIMULATION - Behavioral Simulation - Functional - sim_1 - testbench3

A screenshot of the Vivado interface showing the 'Sources', 'Objects', and 'testvectors.mem' tabs. The 'Sources' tab shows a list of files under 'sim_1': testbench3 (selected), full_adder, mux2_8, testbench1, testbench2, and gates. The 'Objects' tab shows a table of variables with their values: clk=0, reset=0, a=1, b=1, c=1, yexpected=0, y=0, vectornum[3:0]=8, errors[3:0]=0, and testvectors[7:0][3:0]=e,c,b,9,6,4,2,1. The 'testvectors.mem' tab shows the contents of the file: //abc_y expected, followed by a sequence of binary values: 000_1, 001_0, 010_0, 011_0, 100_1, 101_1, 110_0, and 111_0.

1. Generate Clock

```
module testbench3();
    logic      clk, reset;
    logic      a, b, c, yexpected;
    logic      y;
    logic [3:0] vectornum, errors;      // bookkeeping variables
    logic [3:0] testvectors[7:0];      // 8 4-bit test vectors

    // instantiate device under test4-
    sillyfunction dut(a, b, c, y);

    // generate clock
    always      // no sensitivity list, so it always executes
    begin
        clk = 1; #5; clk = 0; #5;
    end

```

2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

initial
begin
    $readmemb("testvectors.mem", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #22; reset = 0;
end

// Note: $readmemh reads testvector files written in
// hexadecimal
```

3. Assign Inputs and Expected Outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
end
```

4. Compare with Expected Outputs

```
// check results on falling edge of clk
always @(negedge clk)
if (~reset) begin // skip during reset
if (y !== yexpected) begin
    $display("Error: inputs = %b", {a, b, c});
    $display(" outputs = %b (%b expected)", y, yexpected);
    errors = errors + 1;
end

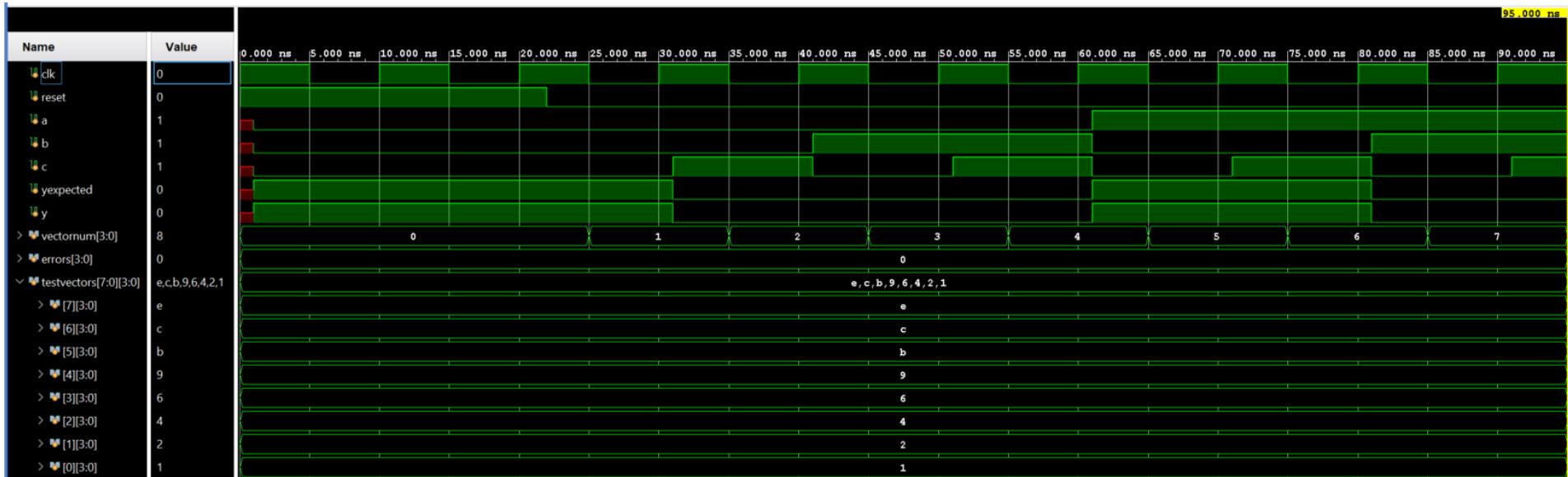
// Note: to print in hexadecimal, use %h. For example,
// $display("Error: inputs = %h", {a, b, c});

// increment array index and read next testvector
vectornum = vectornum + 1;
if (testvectors[vectornum] === 4'bx) begin 
    $display("%d tests completed with %d errors",
            vectornum, errors);
    $stop;
end
end
endmodule

// === and !== can compare values that are 1, 0, x, or z.
```

This bit width needs
to be the same as
test vector size!

4. Simulation Waveform



The figure shows the Tcl Console window with the following output:

```
Tcl Console x Messages Log

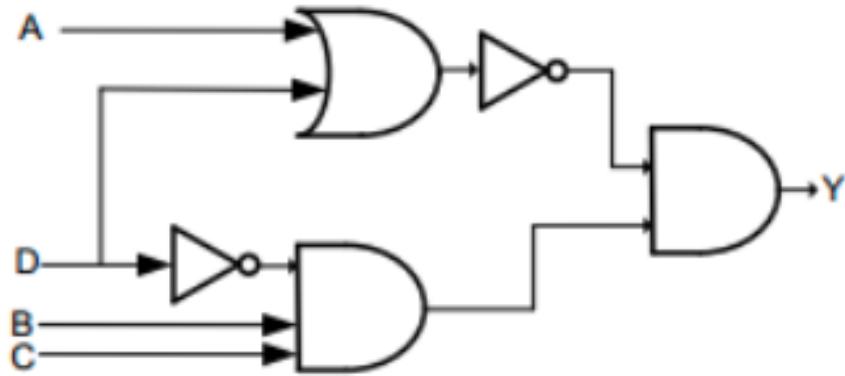
8 tests completed with 0 errors
$stop called at time : 95 ns : File "C:/Users/dcuee/week1/week1.srcts/sim_1/new/testbench3.sv" Line 73
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench3_behav' loaded.
INFO: [USF-XSim-97] Xsim simulation ran for 1000ns
```

End of week 3 slides

SystemVerilog

Testbench, Data Types

SystemVerilog code and testbench



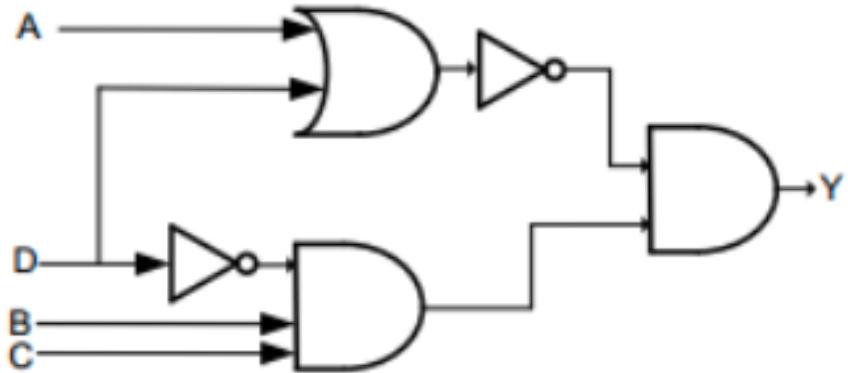
```
module prob1(input wire a,b,c,d,
              output wire out);
  assign y = !(a|d) & (!d&b&c);
endmodule
```

```
module prob1_tb();
  reg a,b,c,d;
  wire out;

  prob1 prob1_test(a,b,c,d, y);

  initial begin
    $monitor(a,b,c,d,y);
    for (int i=0; i<16; i=i+1) begin
      {a,b,c,d} = i;
      #1;
    end
  end
endmodule
```

SystemVerilog code and testbench



```
module prob1(input wire a,b,c,d,
              output wire out);
  assign y = !(a|d) & (!d&b&&c);
endmodule
```

```
module prob1_tb();
  reg a,b,c,d;
  wire out;

  prob1 prob1_test(a,b,c,d, y);

  initial begin
    $monitor(a,b,c,d,y);
    for (int i=0; i<16; i=i+1) begin
      {a,b,c,d} = i;
      #1;
    end
  end
endmodule
```



Output y

in Tcl

Console:

00000

00010

00100

00110

01000

01010

01101

01110

10000

10010

10100

10110

11000

11010

11100

11110

SystemVerilog Data Types

- ❑ SystemVerilog is an extension to Verilog
- ❑ Verilog has `reg` and `wire` data types to describe hardware behaviour
- ❑ SystemVerilog adds more C-like data types for easier testbench description, hardware, better encapsulation and compactness

[https://www.chipverify.com/systemverilog/systemverilog-datatypes](https://www.chipverify.com/systemverilog/systemverilog-datatatypes)

<https://www.chipverify.com/verilog/verilog-data-types>

SystemVerilog Data Types

- ❑ SystemVerilog is an extension to Verilog
- ❑ Verilog has `reg` and `wire` data types to describe hardware behaviour
- ❑ SystemVerilog adds more C-like data types for easier testbench description, hardware, better encapsulation and compactness

[https://www.chipverify.com/systemverilog/systemverilog-datatypes](https://www.chipverify.com/systemverilog/systemverilog-datatatypes)

<https://www.chipverify.com/verilog/verilog-data-types>

Verilog Data Types

- What values do variables hold ?
- What does the verilog value-set imply ?
- Nets and Variables
 - Nets
 - Variables
- Other data-types
 - integer
 - time
 - real
 - Datatypes Example
- Verilog Strings

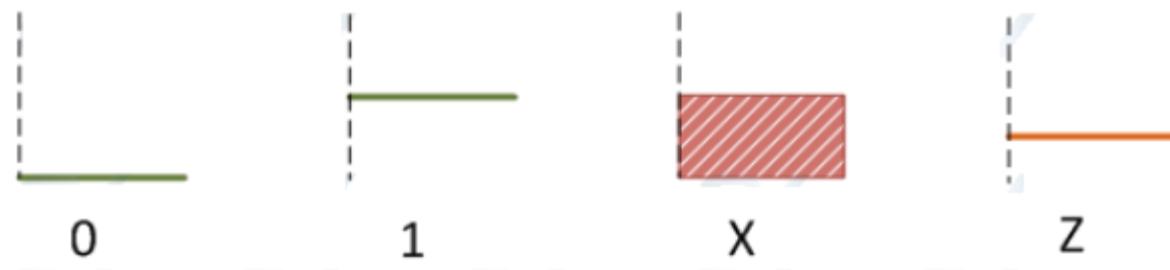
Verilog data types represent data storage elements like bits in a flip-flop and wires that connect gates and flip-flops.

What values do variables hold

- Almost all data types can only have four different values except for `real` and `event` data types

0	logic zero, or a false condition
1	Logic one, or a true condition
x or X	Unknown logic value (can be zero or one)
z or Z	High-impedance state – wire is not connected

- These values are represented in timing diagrams and simulation waveforms as follows



What does the Verilog value-set imply?

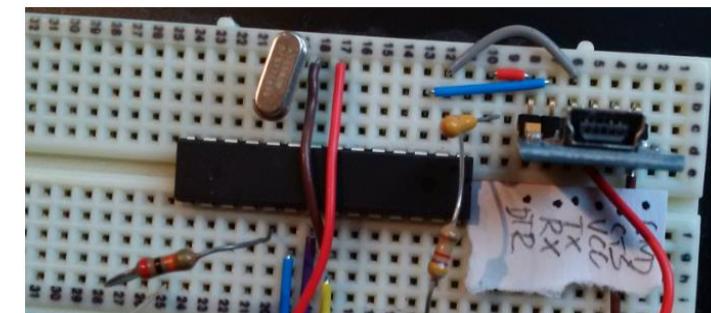
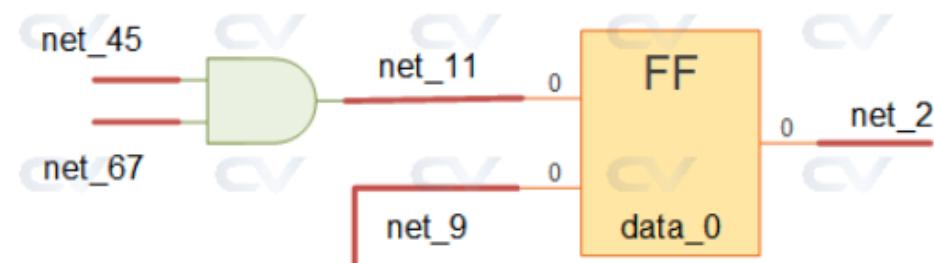
- ❑ Model the value system in hardware
- ❑ Logic one represents the voltage supply Vdd (0.8V – 3V)
- ❑ Logic zero represents ground, i.e. 0V
- ❑ X (or x) means the value is simply known at the time, and could be either 0 or 1.
 - ❑ Quite different from the way X is treated in Boolean logic, where X means “don’t care”.
- ❑ Z (or z) represents that the wire is not connected to anything, it will have high impedance at that node.
 - ❑ In Verilog, any unconnected wire will result in high impedance

Nets and Variables

- ❑ *Nets and variables* are the two main groups of data types representing different hardware structures and differ in how they are assigned and retain values.

Nets

- ❑ Nets are used to connect between hardware entities like logic gates and hence do not store any value on their own.
- ❑ There are different types of nets each with different characteristics. The most widely used net in digital design is of type *wire*.
- ❑ Wire is similar to the electrical wire connecting two breadboard components.



Nets

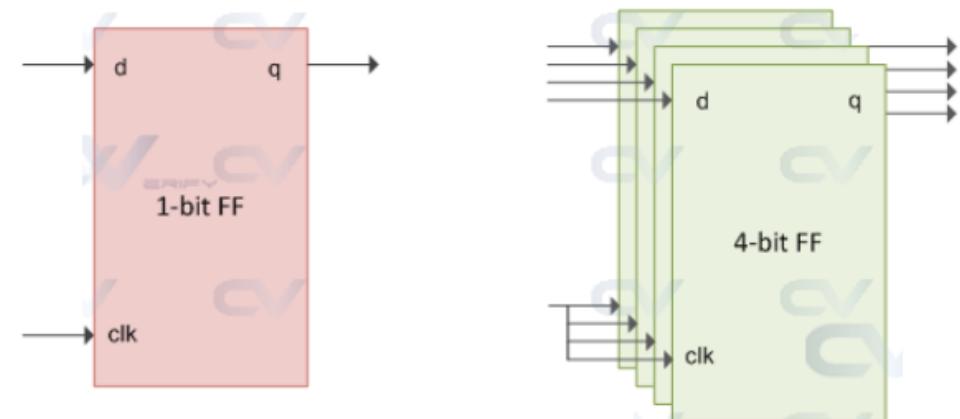
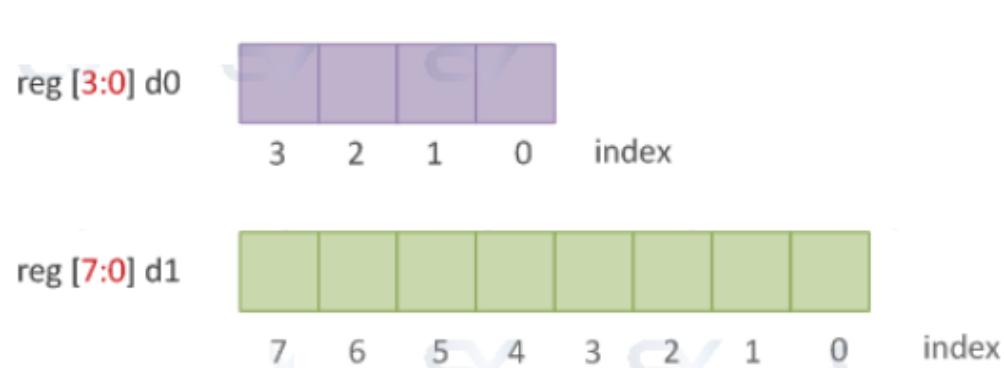
- ❑ Multiple nets can be bunched together to form a single `wire` vector.
- ❑ The following 4-bit wire can send 4 separate bit value one each of the wires.

```
    wire [3:0] n0;           // 4-bit wire -> this is a vector
```



Variables

- ❑ A variable is an abstraction of a data storage element and can hold values, e.g. a flip-flop.
- ❑ Verilog `reg` type can be used to model hardware registers since it can hold values between assignments.
 - ❑ A `reg` need not always represent a flip-flop, it can also be used to represent combinational logic.



Other data-types

- ❑ **Integer** – an `integer` is a general-purpose variable 32-bit wide that can be used for other purposes while modelling hardware and storing integer values.
- ❑ **Time** – a `time` variable is unsigned, 64-bit wide and can be used to store simulation time quantities for debugging purposes. A `realtime` variable simply stores time as a floating point quantity.
- ❑ **Real** – a `real` variable can store floating point values and be assigned the same way as `integer` and `reg`.

SystemVerilog Data Types

- SystemVerilog is an extension to Verilog, adding more C-like data types

Data-type	2-state/4-state	# Bits	signed/unsigned	C equivalent
S Y S T E M	reg	4	≥ 1	unsigned
	wire	4	≥ 1	unsigned
	integer	4	32	signed
	real			double
	time			
	realtime			double
V E R I L O G	logic	4	≥ 1	unsigned
	bit	2	≥ 1	unsigned
	byte	2	8	signed
	shortint	2	16	signed
	int	2	32	signed
	longint	2	64	signed
	shortreal			float

Data-type examples

```
module data_types_tb;

real pi;      // declare to be of type real
real freq;

initial begin

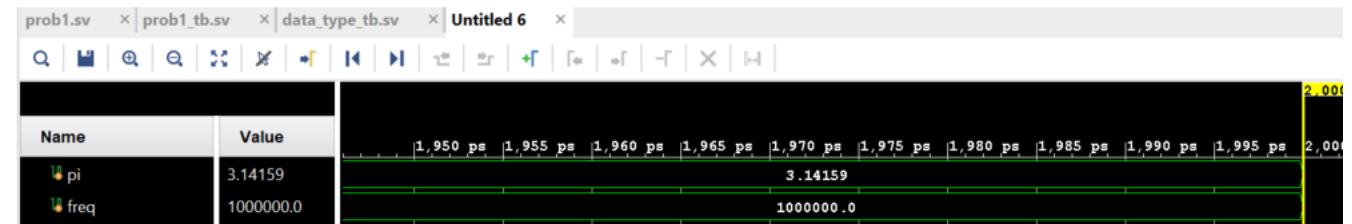
pi = 3.14159; // store floating point number
freq = 1e6; // store exponential number

$display("Value of pi = ", pi);
$display("Value of pi = %0.3f", pi);
$display("Value of freq = %0d", freq);

end

endmodule
```

Name	Value	Data Type
pi	3.14159	Float
freq	1000000.0	Float



```
Value of pi = 3.14159
Value of pi = 3.142
Value of freq = 1000000
```

Verilog Data Types

- The primary intent fo data-types
- 2-state type can take only 0, 1
- 4-state types can take 0, 1, X, Z
- A signal with more than one driver needs to be declared a net-type such as wire so that SystemVerilog can resolve the final value.

Chapter 6: Architecture

Logical / Shift Instructions

Programming

- **High-level languages:**
 - e.g., C, Java, Python
 - Written at higher level of abstraction
- **High-level constructs:** loops, conditional statements, arrays, function calls
- **First, introduce instructions that support these:**
 - Logical operations
 - Shift instructions
 - Multiplication & division
 - Branches & Jumps

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- English mathematician
- Daughter of the poet Lord Byron



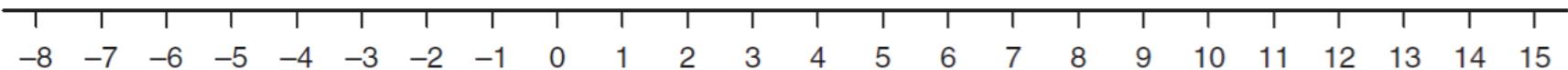
Logical Instructions

- **and, or, xor**
 - and: useful for **masking** bits
 - Masking all but the least significant byte of a value:
 $0xF234012F \text{ AND } 0x000000FF} = 0x0000002F$
 - or: useful for **combining** bit fields
 - Combine $0xF2340000$ with $0x000012BC$:
 $0xF2340000 \text{ OR } 0x000012BC} = 0xF23412BC$
 - xor: useful for **inverting** bits:
 - $A \text{ XOR } -1} = \text{NOT } A$ (remember that $-1} = 0xFFFFFFFF$)

Range of N -bit Number

Table 1.3 Range of N -bit numbers

System	Range
Unsigned	$[0, 2^N - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$



Unsigned 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

1000 1001 1010 1011 1100 1101 1110 1111 Two's Complement

1111 1110 1101 1100 1011 1010 1001 Sign/Magnitude

Figure 1.11 Number line and 4-bit binary encodings

Logical Instructions: Example 1

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

and s3, s1, s2

or s4, s1, s2

xor s5, s1, s2

Result

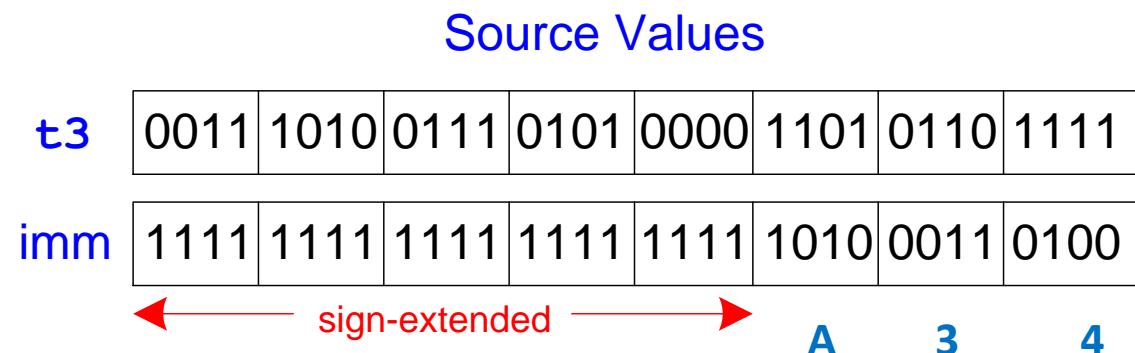
s3	0100 0110	1010 0001	0000 0000	0000 0000
s4	1111 1111	1111 1111	1111 0001	1011 0111
s5	1011 1001	0101 1110	1111 0001	1011 0111

Logical Instructions: Example 2

Source Values									
t3	0011 1010 0111 0101 0000					1101	0110	1111	
imm	1111 1111 1111 1111 1111					1010	0011	0100	
						← sign-extended →	A	3	4
Assembly Code									
andi s5, t3, -1484	s5								
ori s6, t3, -1484	s6								
xori s7, t3, -1484	s7								
Result									

$-1484 = \text{0xA34}$ in 12-bit 2's complement representation.

Logical Instructions: Example 2



Assembly Code

```
andi s5, t3, -1484  
ori s6, t3, -1484  
xori s7, t3, -1484
```

Result

s5	0011	1010	0111	0101	0000	1000	0010	0100
s6	1111	1111	1111	1111	1111	1111	0111	1111
s7	1100	0101	1000	1010	1111	0111	0101	1011

-1484 = **0xA34** in 12-bit 2's complement representation.

Shift Instructions

Shift amount is in (lower 5 bits of) a register

- sll: shift left logical

- **Example:** sll t0, t1, t2 # t0 = t1 << t2

t0: Destination register where the result will be stored.

t1: The source register whose value will be shifted.

t2: The register (lower 5 bits) specifies the number of bits to shift positions.

- srl: shift right logical

- **Example:** srl t0, t1, t2 # t0 = t1 >> t2

- sra: shift right arithmetic

- **Example:** sra t0, t1, t2 # t0 = t1 >>> t2

Immediate Shift Instructions

Shift amount is an immediate between 0 to 31

- **slli: shift left logical immediate**
 - **Example:** slli t0, t1, 23 # $t_0 = t_1 \ll 23$
 - Fills the rightmost bits with 0s
- **srlti: shift right logical immediate**
 - **Example:** srlti t0, t1, 18 # $t_0 = t_1 \gg 18$
 - Fills the leftmost bits with 0s
- **srai: shift right arithmetic immediate**
 - **Example:** srai t0, t1, 5 # $t_0 = t_1 \ggg 5$
 - Fills leftmost bits with the value of the sign bit (the most significant bit), preserving the sign for signed numbers

Immediate Shift Examples

Instruction	Initial Value (x5)	Result	Binary Result
slli x6, x5, 4	0xF000_000F	0x0000_00F0	0000 0000 0000 0000 0000 0000 1111 0000
srlti x7, x5, 4	0xF000_000F	0x0F00_0000	0000 1111 0000 0000 0000 0000 0000 0000
sraisi x8, x5, 4	0xF000_000F	0xFF00_0000	1111 1111 0000 0000 0000 0000 0000 0000

- **slli**: Fills the rightmost bits with 0s
- **srlti**: Fills the leftmost bits with 0s
- **sraisi**: Fills leftmost bits with the value of the sign bit (the most significant bit), preserving the sign for signed numbers

Chapter 6: Architecture

Multiplication and Division

Multiplication

32×32 multiplication \rightarrow 64 bit result

mul s3, s1, s2

s3 = lower 32 bits of result

mulh s4, s1, s2

s4 = upper 32 bits of result, treats operands as signed

{s4, s3} = s1 \times s2

Example: s1 = 0x40000000 = 2^{30} ; s2 = 0x80000000 = -2^{31}

$s1 \times s2 = -2^{61} = 0xE0000000\ 00000000$

s4 = 0xE0000000; s3 = 0x00000000

Division

32-bit division → 32-bit quotient & remainder

- div s3, s1, s2 # s3 = s1/s2
- rem s4, s1, s2 # s4 = s1%s2

Example: $s1 = 0x00000011 = 17$; $s2 = 0x00000003 = 3$

$$s1 / s2 = 5$$

$$s1 \% s2 = 2$$

$$s3 = 0x00000005; s4 = 0x00000002$$

Chapter 6: Architecture

Branches & Jumps

Branching

- Execute instructions out of sequence
- Types of branches:
 - **Conditional**
 - branch if equal (beq)
 - branch if not equal (bne)
 - branch if less than (blt)
 - branch if greater than or equal (bge)
 - **Unconditional**
 - jump (j)
 - jump register (jr)
 - jump and link (jal)
 - jump and link register (jalr)

Branching

- Execute instructions out of sequence
- Types of branches:
 - **Conditional**
 - branch if equal (beq)
 - branch if not equal (bne)
 - branch if less than (blt)
 - branch if greater than or equal (bge)
 - **Unconditional**
 - jump (j)
 - jump register (jr)
 - jump and link (jal)
 - jump and link register (jalr)

We'll talk
about these
when discuss
function calls

Conditional Branching

RISC-V assembly

```
addi s0, zero, 4          # s0 = 0 + 4 = 4
addi s1, zero, 1          # s1 = 0 + 1 = 1
slli s1, s1, 2            # s1 = 1 << 2 = 4
beq s0, s1, target        # branch is taken
addi s1, s1, 1            # not executed
sub s1, s1, s0             # not executed

target:                   # label
    add s1, s1, s0          # s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by a colon (:)

The Branch Not Taken (bne)

RISC-V assembly

addi	s0, zero, 4	# s0 = 0 + 4 = 4
addi	s1, zero, 1	# s1 = 0 + 1 = 1
slli	s1, s1, 2	# s1 = 1 << 2 = 4
bne	s0, s1, target	# branch not taken
addi	s1, s1, 1	# s1 = 4 + 1 = 5
sub	s1, s1, s0	# s1 = 5 - 4 = 1

target:

add	s1, s1, s0	# s1 = 1 + 4 = 5
-----	------------	------------------

Unconditional Branching (j)

RISC-V assembly

```
j      target          # jump to target
srai   s1, s1, 2       # not executed
addi   s1, s1, 1       # not executed
sub    s1, s1, s0      # not executed
```

target:

```
add   s1, s1, s0      # s1 = 1 + 4 = 5
```

Chapter 6: Architecture

Conditional Statements & Loops

Conditional Statements & Loops

- **Conditional Statements**
 - if statements
 - if/else statements
- **Loops**
 - while loops
 - for loops

If Statement

C Code

```
if (i == j)  
    f = g + h;
```

```
f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h  
# s3 = i, s4 = j
```

If Statement

C Code

```
if (i == j)
    f = g + h;

f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
        bne s3, s4, L1
        add s0, s1, s2

L1:
        sub s0, s0, s3
```

If Statement

C Code

```
if (i == j)  
    f = g + h;
```

```
f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h  
# s3 = i, s4 = j  
bne s3, s4, L1  
add s0, s1, s2  
  
L1:  
sub s0, s0, s3
```

Assembly tests opposite case (`i != j`) of high-level code (`i == j`)

If/Else Statement

C Code

```
if (i == j)
    f = g + h;
```

```
else
    f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
```

If/Else Statement

C Code

```
if (i == j)
    f = g + h;
```

```
else
    f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
        bne  s3, s4, L1
        add   s0, s1, s2
        j     done
```

```
L1:
```

```
        sub   s0, s0, s3
```

```
done:
```

If/Else Statement

C Code

```
if (i == j)
    f = g + h;
```

```
else
    f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
        bne  s3, s4, L1
        add   s0, s1, s2
        j     done
```

```
L1:
```

```
        sub   s0, s0, s3
```

```
done:
```

Assembly tests opposite case ($i \neq j$) of high-level code ($i == j$)

While Loops

C Code

```
// determines the power  
// of x such that 2x = 128  
int pow = 1;  
int x    = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x   = x + 1;  
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x
```

While Loops

C Code

```
// determines the power  
// of x such that 2x = 128  
  
int pow = 1;  
int x    = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x   = x + 1;  
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x  
  
addi s0, zero, 1  
add  s1, zero, zero  
addi t0, zero, 128  
  
while:  
    beq  s0, t0, done  
    slli s0, s0, 1  
    addi s1, s1, 1  
    j     while  
  
done:
```

While Loops

C Code

```
// determines the power  
// of x such that 2x = 128  
  
int pow = 1;  
int x    = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x   = x + 1;  
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x  
  
addi s0, zero, 1  
add  s1, zero, zero  
addi t0, zero, 128  
  
while:  
beq  s0, t0, done  
slli s0, s0, 1  
addi s1, s1, 1  
j     while  
  
done:
```

**Assembly tests opposite case (`pow == 128`) of high-level code
(`pow != 128`)**

For Loops

```
for (initialization; condition; loop operation)  
    statement
```

- **initialization**: executes **before** the loop begins
- **condition**: is tested **at the beginning** of each iteration
- **loop operation**: executes at the **end** of each iteration
- **statement**: executes **each time** the condition is met

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
```

For Loops

C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
addi s1, zero, 0
add s0, zero, zero
addi t0, zero, 10

for:
    beq s0, t0, done
    add s1, s1, s0
    addi s0, s0, 1
    j for

done:
```

Less Than Comparison

C Code

```
// add the powers of 2 from 1,  
// to the powers of 2 result  
// just below 101  
  
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
```

Less Than Comparison

C Code

```
// add the powers of 2 from 1,  
// to the powers of 2 result  
// just below 101  
  
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum  
addi s1, zero, 0  
addi s0, zero, 1  
addi t0, zero, 101  
  
loop:  
    bge s0, t0, done  
    add s1, s1, s0  
    slli s0, s0, 1  
    j loop  
  
done:
```

Less Than Comparison: Version 2

C Code

```
// add the powers of 2 from 1  
// to 100  
int sum = 0;  
int i;  
  
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum  
addi s1, zero, 0  
addi s0, zero, 1  
addi t0, zero, 101  
  
loop:  
    slt t2, s0, t0  
    beq t2, zero, done  
    add s1, s1, s0  
    slli s0, s0, 1  
    j loop  
  
done:
```

slt: set if less than instruction

```
slt t2, s0, t0 # if s0 < t0, t2 = 1  
                # otherwise t2 = 0
```

Chapter 6: Architecture

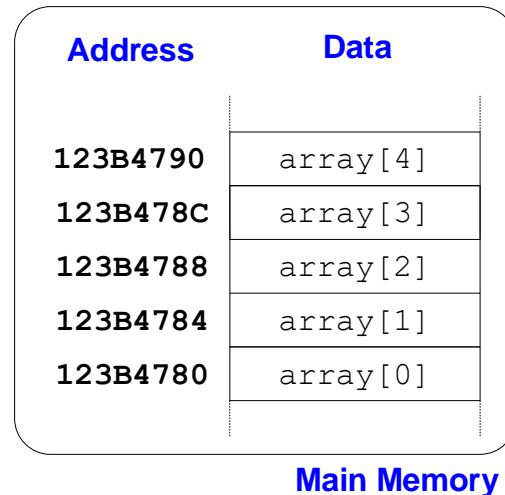
Arrays

Arrays

- Access large amounts of similar data
- **Index:** access each element
- **Size:** number of elements

Arrays

- 5-element array
- **Base address** = 0x123B4780 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



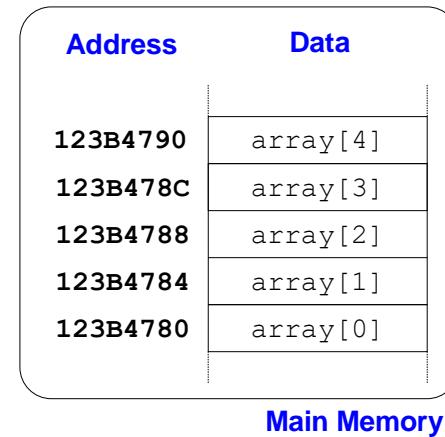
Accessing Arrays

// C Code

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

RISC-V assembly code

```
# s0 = array base address
```



Accessing Arrays

// C Code

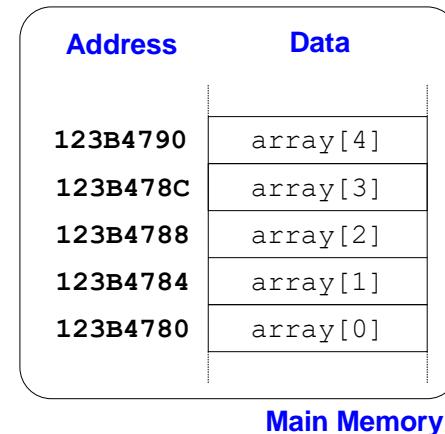
```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

RISC-V assembly code

```
# s0 = array base address
li    s0, 0x123B4780          # s0 = 0x123B4780

lw    t1, 0(s0)                # t1 = array[0]
slli  t1, t1, 1                # t1 = t1 * 2
sw    t1, 0(s0)                # array[0] = t1

lw    t1, 4(s0)                # t1 = array[1]
slli  t1, t1, 1                # t1 = t1 * 2
sw    t1, 4(s0)                # array[1] = t1
```



Main Memory

Accessing Arrays Using For Loops

// C Code

```
int array[1000];
int i;

for (i=0; i < 1000; i = i + 1)
    array[i] = array[i] * 8;
```

RISC-V assembly code

```
# s0 = array base address, s1 = i
```

Accessing Arrays Using For Loops

RISC-V assembly code

```
# s0 = array base address, s1 = i
# initialization code
li s0, 0x23B8F400          # s0 = 0x23B8F400

addi s1, zero, 0             # i = 0
addi t2, zero, 1000          # t2 = 1000

loop:
bge s1, t2, done            # if not then done
slli t0, s1, 2                # t0 = i * 4 (byte offset)
add t0, t0, s0                # address of array[i]
lw t1, 0(t0)                  # t1 = array[i]
slli t1, t1, 3                # t1 = array[i] * 8
sw t1, 0(t0)                  # array[i] = array[i] * 8
addi s1, s1, 1                # i = i + 1
j loop                         # repeat

done:
```

```
// C Code
int array[1000];
int i;
for (i=0; i < 1000; i = i + 1)
    array[i] = array[i] * 8;
```

ASCII Code

- **ASCII:** *American Standard Code for Information Interchange*
- Each text character has unique byte value
 - For example, S = 0x53, a = 0x61, A = 0x41
 - Lower-case and upper-case differ by 0x20 (32)

Cast of Characters: ASCII Encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Accessing Arrays of Characters

// C Code

```
char str[80] = "CAT";
int len = 0;

// compute length of string
while (str[len]) len++;
```

RISC-V assembly code

```
# s0 = array base address, s1 = len
```

Accessing Arrays of Characters

// C Code

```
char str[80] = "CAT";
int len = 0;

// compute length of string
while (str[len]) len++;
```

RISC-V assembly code

```
# s0 = array base address, s1 = len

    addi s1, zero, 0          # len = 0
while: add t0, s0, s1          # address of str[len]
      lw t1, 0(t0)           # load str[len]
      beq t1, zero, done      # are we at the end of the string?
      addi s1, s1, 1          # len++
      j while                # repeat while loop
done:
```

- End of week 3 slides

Chapter 6: Architecture

Function Calls

Function Calls

- **Caller:** calling function (in this case, main)
- **Callee:** called function (in this case, sum)

C Code

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```

Simple Function Call

C Code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

RISC-V assembly code

0x00000300	main:	jal simple	# call
0x00000304		add s0, s1, s2	
...		...	
0x0000051c	simple:	jr ra	# return

void means that simple doesn't return a value

jal simple:

$ra = PC + 4 (0x00000304)$

jumps to simple label ($PC = 0x0000051c$)

jr ra:

$PC = ra (0x00000304)$

RISC-V Register Set

Pseudoinstruction	RISC-V Instructions	Description	Operation	Name	Register Number	Usage
jal label	jal ra, label	jump and link	$PC = \text{label}, ra = PC + 4$	zero	x0	Constant value 0
jr rs1	jalr x0, rs1, 0	jump register	$PC = rs1$	ra	x1	Return address

Function Calling Conventions

- **Caller:**
 - passes **arguments** to callee
 - jumps to callee
- **Callee:**
 - **performs** the function
 - **returns** result to caller
 - **returns** to point of call
 - **must not overwrite** registers or memory needed by caller

RISC-V Function Calling Conventions

- **Call Function:** jump and link (jal func)
- **Return** from function: jump register (jr ra)
- **Arguments:** a0 – a7
- **Return value:** a0

RISC-V Register Set		
Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Input Arguments & Return Value

C Code

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

Input Arguments & Return Value

RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2 # argument 0 = 2
addi a1, zero, 3 # argument 1 = 3
addi a2, zero, 4 # argument 2 = 4
addi a3, zero, 5 # argument 3 = 5
jal diffofsums    # call function
add s7, a0, zero # y = returned value (return value in a0)
. . .
# s3 = result
diffofsums:
add t0, a0, a1    # t0 = f + g
add t1, a2, a3    # t1 = h + i
sub s3, t0, t1    # result = (f + g) - (h + i)
add a0, s3, zero # put return value in a0
jr ra            # return to caller
```

Input Arguments & Return Value

RISC-V assembly code

```
# s3 = result
diffofsums:
    add  t0, a0, a1    # t0 = f + g
    add  t1, a2, a3    # t1 = h + i
    sub  s3, t0, t1    # result = (f + g) - (h + i)
    add  a0, s3, zero # put return value in a0
    jr   ra            # return to caller
```

- diffofsums **overwrote** 3 registers: t0, t1, s3
- diffofsums **can use stack** to temporarily store registers

Chapter 6: Architecture

The Stack

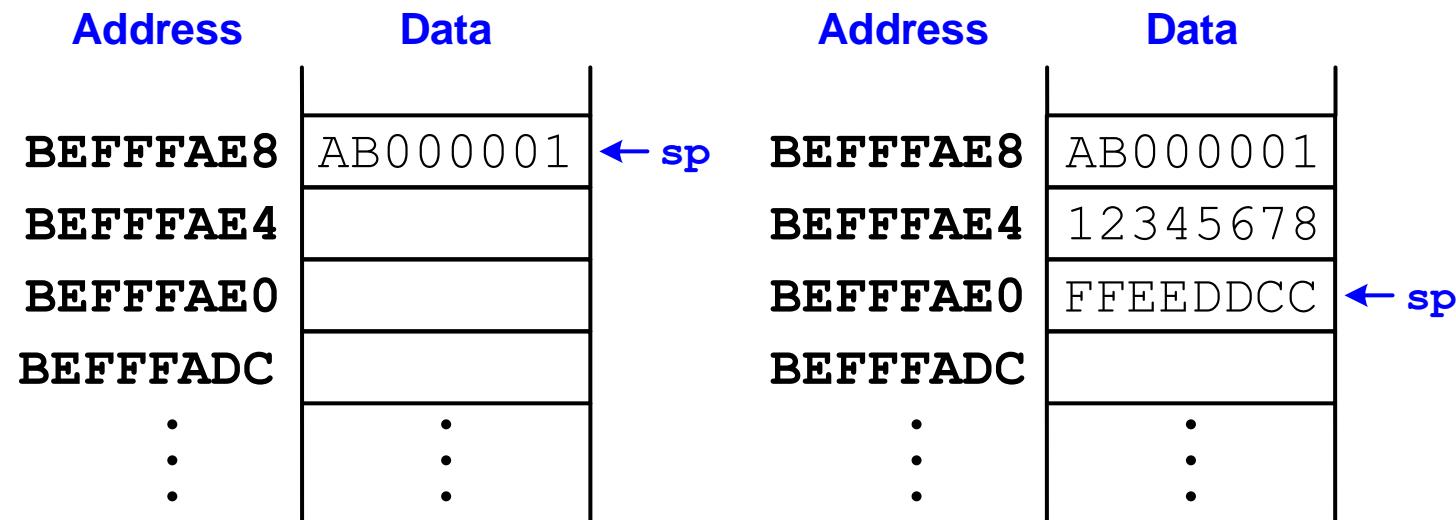
The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands:*** uses more memory when more space needed
- ***Contracts:*** uses less memory when the space is no longer needed



The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: sp points to top of the stack



Make room on stack for **2 words**.

How Functions use the Stack

- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0`, `t1`, `s3`

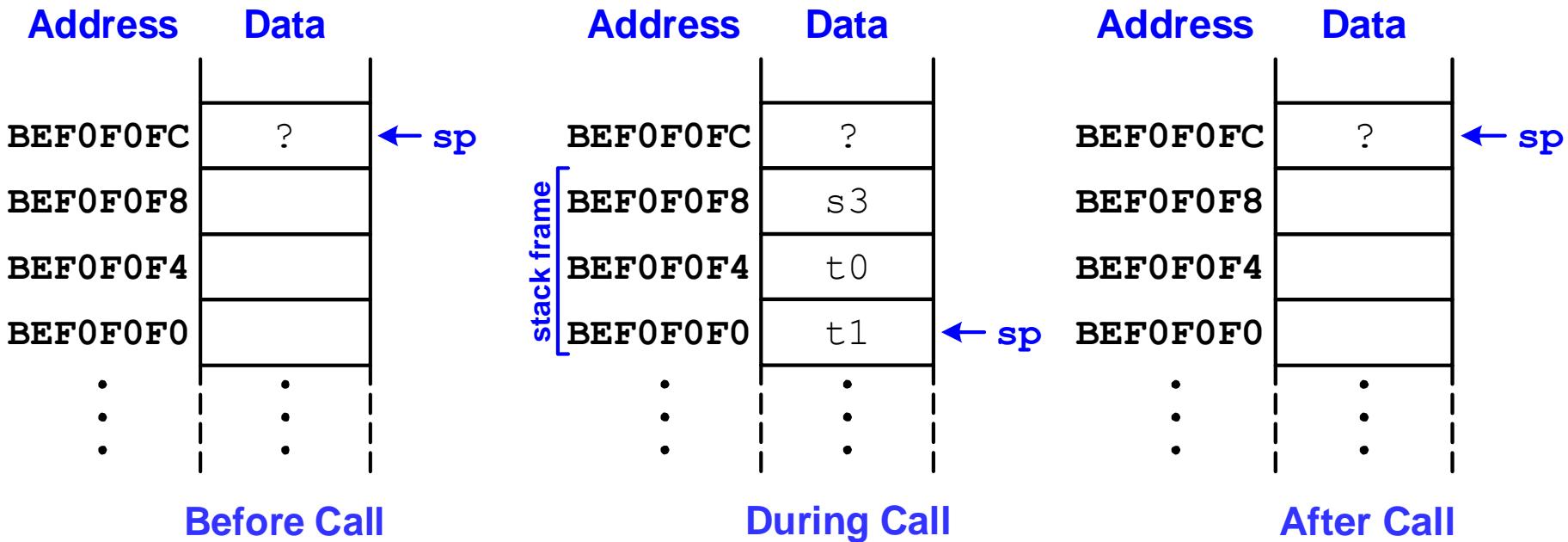
RISC-V assembly

```
# s3 = result
diffofsums:
    add  t0, a0, a1    # t0 = f + g
    add  t1, a2, a3    # t1 = h + i
    sub  s3, t0, t1    # result = (f + g) - (h + i)
    add  a0, s3, zero  # put return value in a0
    jr   ra             # return to caller
```

Storing Register Values on the Stack

```
# s3 = result
diffofsums:
    addi sp, sp, -12          # make space on stack to
                                # store three registers
    sw   s3, 8(sp)           # save s3 on stack
    sw   t0, 4(sp)           # save t0 on stack
    sw   t1, 0(sp)           # save t1 on stack
    add  t0, a0, a1          # t0 = f + g
    add  t1, a2, a3          # t1 = h + i
    sub  s3, t0, t1          # result = (f + g) - (h + i)
    add  a0, s3, zero        # put return value in a0
    lw   s3, 8(sp)           # restore s3 from stack
    lw   t0, 4(sp)           # restore t0 from stack
    lw   t1, 0(sp)           # restore t1 from stack
    addi sp, sp, 12          # deallocate stack space
    jr   ra                  # return to caller
```

The Stack During `diffofsums` Call



Preserved Registers

Preserved <i>Callee-Saved</i>	Nonpreserved <i>Caller-Saved</i>
s0-s11	t0-t6
sp	a0-a7
ra	
stack above sp	stack below sp

The preserved registers are s0 to s11 (hence their name, *saved*), sp, and ra. The nonpreserved registers, also called *scratch* registers, are t0 to t6 (hence their name, *temporary*) and a0 to a7, the argument registers. A function can change the nonpreserved registers freely but must save and restore any of the preserved registers that it uses.

Storing Saved Registers on the Stack

```
# s3 = result
diffofsums:
    addi sp, sp, -4          # make space on stack to
                            # store one register
    sw   s3, 0(sp)          # save s3 on stack
    add  t0, a0, a1          # t0 = f + g
    add  t1, a2, a3          # t1 = h + i
    sub  s3, t0, t1          # result = (f + g) - (h + i)
    add  a0, s3, zero        # put return value in a0
    lw   s3, 0(sp)          # restore s3 from stack
    addi sp, sp, 4           # deallocate stack space
    jr  ra                  # return to caller
```

Optimized diffofsums

```
# a0 = result
diffofsums:
    add t0, a0, a1      # t0 = f + g
    add t1, a2, a3      # t1 = h + i
    sub a0, t0, t1      # result = (f + g) - (h + i)
    jr ra               # return to caller
```

Non-Leaf Function Calls

Non-leaf function:

a function that calls another function

```
func1:  
    addi sp, sp, -4      # make space on stack  
    sw   ra, 0(sp)       # save ra on stack  
    jal  func2  
    ...  
    lw   ra, 0(sp)       # restore ra from stack  
    addi sp, sp, 4        # deallocate stack space  
    jr  ra                # return to caller
```

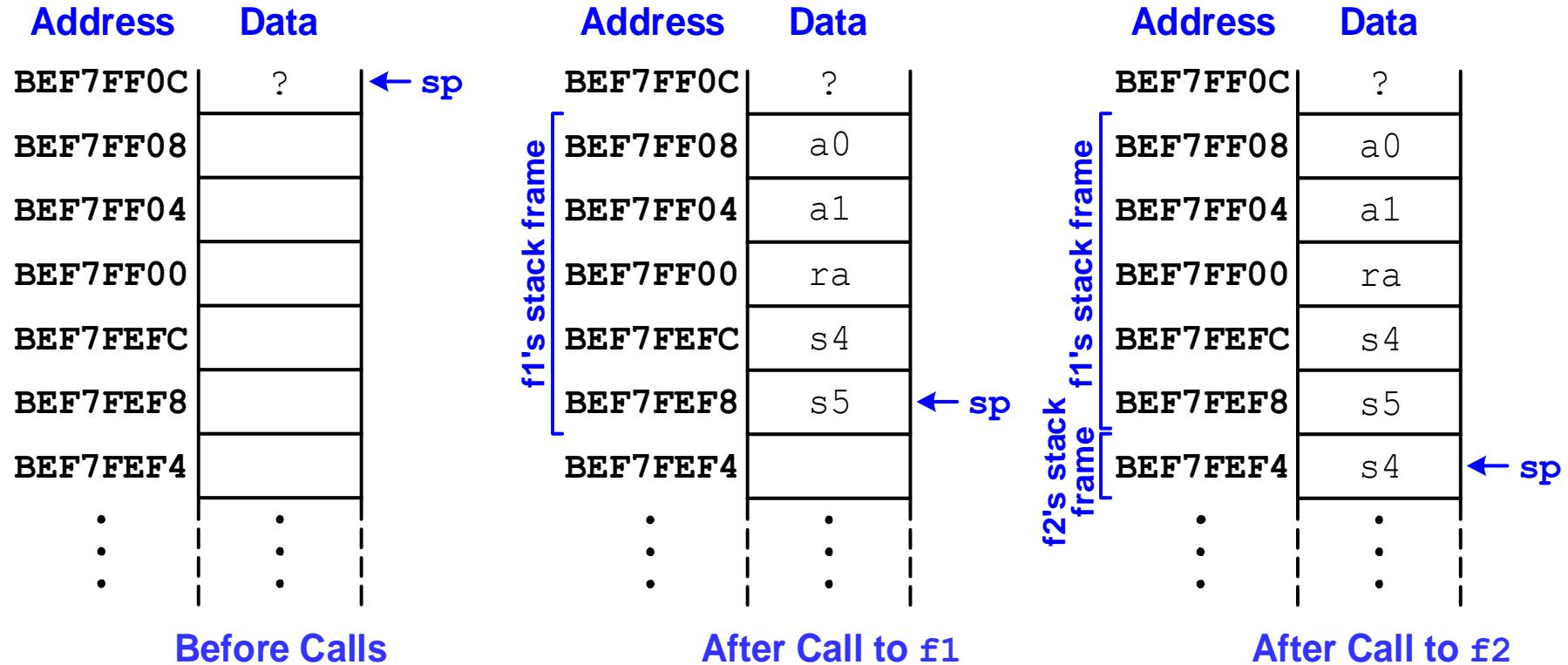
Must preserve **ra** before function call.

Non-Leaf Function Call Example

```
# f1 (non-leaf function) uses s4-s5 and needs a0-a1 after call to f2
f1:
    addi sp, sp, -20      # make space on stack for 5 words
    sw   a0, 16(sp)
    sw   a1, 12(sp)
    sw   ra, 8(sp)        # save ra on stack
    sw   s4, 4(sp)
    sw   s5, 0(sp)
    jal  func2
    ...
    lw   ra, 8(sp)        # restore ra (and other regs) from stack
    ...
    addi sp, sp, 20        # deallocate stack space
    jr   ra                # return to caller

# f2 (leaf function) only uses s4 and calls no functions
f2:
    addi sp, sp, -4        # make space on stack for 1 word
    sw   s4, 0(sp)
    ...
    lw   s4, 0(sp)
    addi sp, sp, 4        # deallocate stack space
    jr   ra                # return to caller
```

Stack during Function Calls



Function Call Summary

- **Caller**

- Save any needed registers (ra, maybe t0-t6/a0-a7)
- Put arguments in a0-a7
- Call function: jal callee
- Look for result in a0
- Restore any saved registers

- **Callee**

- Save registers that might be disturbed (s0-s11)
- Perform function
- Put result in a0
- Restore registers
- Return: jr ra

Chapter 6: Architecture

Recursive Functions

Recursive Function Example

- Function that **calls itself**
- When converting to assembly code:
 - In the first pass, treat recursive calls as if it's calling a different function and ignore overwritten registers.
 - Then save/restore registers on stack as needed.

Recursive Function Example

- **Factorial function:**

- $\text{factorial}(n) = n!$
 $= n * (n-1) * (n-2) * (n-3) \dots * 1$

- **Example:** $\text{factorial}(6) = 6!$
 $= 6 * 5 * 4 * 3 * 2 * 1$
 $= 720$

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

Example: n = 3

factorial(3) : returns 3*factorial(2)

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

Example: n = 3

factorial(3) : returns 3*factorial(2)
factorial(2) : returns 2*factorial(1)

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

Example: n = 3

factorial(3) : returns 3*factorial(2)
factorial(2) : returns 2*factorial(1)
factorial(1) : returns 1

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1)); } Thus,
```

Example: n = 3

```
factorial(3) : returns 3*factorial(2)  
factorial(2) : returns 2*factorial(1)  
factorial(1) : returns 1  
factorial(1) : returns 1
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1));  
}
```

Example: n = 3

factorial(3) : returns $3 * \text{factorial}(2)$
factorial(2) : returns $2 * \text{factorial}(1)$
factorial(1) : returns 1

Thus,

factorial(1) : returns 1
factorial(2) : returns $2 * 1 = 2$

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n*factorial(n-1)); }
```

Example: n = 3

factorial(3) : returns $3 * \text{factorial}(2)$
factorial(2) : returns $2 * \text{factorial}(1)$
factorial(1) : returns 1

factorial(1) : returns 1
factorial(2) : returns $2 * 1 = 2$
factorial(3) : returns $3 * 2 = \mathbf{6}$

Thus,

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

RISC-V Assembly

```
factorial:
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

RISC-V Assembly

```
factorial:  
  
    addi t0, zero, 1    # temporary = 1  
    bgt a0, t0, else   # if n>1, go to else  
    addi a0, zero, 1    # otherwise, return 1  
  
    jr ra              # return  
else:  
    addi a0, a0, -1    # n = n - 1  
    jal factorial      # recursive call
```

Pass 1. Treat as if calling another function. Ignore stack.

```
mul a0, a0, a0    # a0=n*factorial(n-1)  
jr ra              # return
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

RISC-V Assembly

```
factorial:  
  
    addi t0, zero, 1    # temporary = 1  
    bgt a0, t0, else   # if n>1, go to else  
    addi a0, zero, 1    # otherwise, return 1  
  
    jr ra              # return  
else:  
    addi a0, a0, -1    # n = n - 1  
    jal factorial      # recursive call
```

Pass 1. Treat as if calling another function. Ignore stack.

n return value: factorial(n-1)

```
    mul a0, a0, a0      # a0=n*factorial(n-1)  
    jr ra              # return
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

RISC-V Assembly

```
factorial:  
  
    addi t0, zero, 1    # temporary = 1  
    bgt a0, t0, else   # if n>1, go to else  
    addi a0, zero, 1    # otherwise, return 1  
  
    jr ra              # return  
else:  
    addi a0, a0, -1    # n = n - 1  
    jal factorial      # recursive call
```

Pass 1. Treat as if calling another function. Ignore stack.

Pass 2. Save overwritten registers (needed after function call) on the stack before call.

n return value: factorial(n-1)

```
    mul a0, a0, a0      # a0=n*factorial(n-1)  
    jr ra              # return
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

Pass 1. Treat as if calling another function. Ignore stack.

Pass 2. Save overwritten registers (needed after function call) on the stack before call.

RISC-V Assembly

```
factorial:  
  
    addi t0, zero, 1    # temporary = 1  
    bgt a0, t0, else    # if n>1, go to else  
    addi a0, zero, 1    # otherwise, return 1  
  
    jr ra                # return  
else:  
    addi a0, a0, -1      # n = n - 1  
    jal factorial         # recursive call
```

n return value: factorial (n-1)

↓ ↓

```
mul a0, a0, a0    # a0=n*factorial (n-1)  
jr ra                # return
```

Problem: n (a0) was overwritten by function call!
Must save it (and `ra`) on stack before function call.

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

RISC-V Assembly

```
factorial:  
    addi sp, sp, -8      # save regs  
    sw   a0, 4(sp)  
    sw   ra, 0(sp)  
    addi t0, zero, 1     # temporary = 1  
    bgt a0, t0, else    # if n>1, go to else  
    addi a0, zero, 1     # otherwise, return 1  
  
    jr   ra              # return  
else:  
    addi a0, a0, -1      # n = n - 1  
    jal  factorial       # recursive call
```

Pass 1. Treat as if calling another function. Ignore stack.

Pass 2. Save overwritten registers (needed after function call) on the stack before call.

```
mul  a0, t1, a0      # a0=n*factorial(n-1)  
jr   ra              # return
```

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

- Pass 1.** Treat as if calling another function. Ignore stack.
Pass 2. Save overwritten registers (needed after function call) on the stack before call.

RISC-V Assembly

```
factorial:  
    addi sp, sp, -8      # save regs  
    sw   a0, 4(sp)  
    sw   ra, 0(sp)  
    addi t0, zero, 1    # temporary = 1  
    bgt a0, t0, else    # if n>1, go to else  
    addi a0, zero, 1    # otherwise, return 1  
  
    jr   ra             # return  
else:  
    addi a0, a0, -1    # n = n - 1  
    jal  factorial     # recursive call  
    lw   t1, 4(sp)      # restore n into t1  
    lw   ra, 0(sp)      # restore ra  
    addi sp, sp, 8      # restore sp  
    mul a0, t1, a0      # a0=n*factorial(n-1)  
    jr   ra             # return
```

Note: n is restored from stack into t1 so it doesn't overwrite return value in a0.

Recursive Function Example

High-Level Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

Pass 1. Treat as if calling another function. Ignore stack.

Pass 2. Save overwritten registers (needed after function call) on the stack before call.

RISC-V Assembly

```
factorial:  
    addi sp, sp, -8      # save regs  
    sw   a0, 4(sp)  
    sw   ra, 0(sp)  
    addi t0, zero, 1    # temporary = 1  
    bgt a0, t0, else    # if n>1, go to else  
    addi a0, zero, 1    # otherwise, return 1  
    addi sp, sp, 8       # restore sp  
    jr   ra              # return  
  
else:  
    addi a0, a0, -1      # n = n - 1  
    jal  factorial        # recursive call  
    lw   t1, 4(sp)        # restore n into t1  
    lw   ra, 0(sp)        # restore ra  
    addi sp, sp, 8       # restore sp  
    mul  a0, t1, a0        # a0=n*factorial(n-1)  
    jr   ra              # return
```

Note: n is restored from stack into t1 so it doesn't overwrite return value in a0.

Recursive Functions

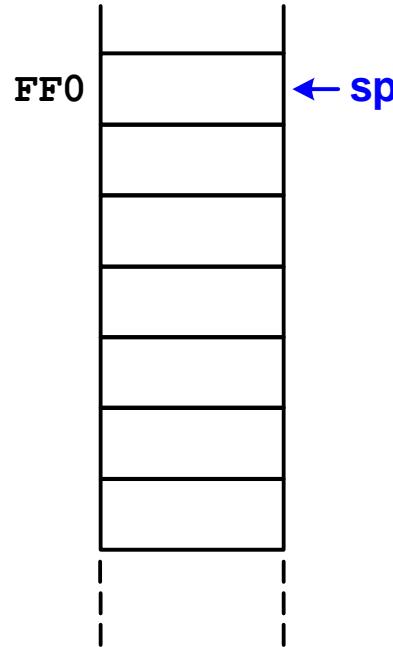
```
0x8500 factorial: addi sp, sp, -8      # save registers
0x8504             sw   a0, 4(sp)        # save n in a0 to stack
0x8508             sw   ra, 0(sp)        # save return address ra
0x850C             addi t0, zero, 1      # temporary = 1
0x8510             bgt  a0, t0, else      # if n > 1, go to else
0x8514             addi a0, zero, 1      # otherwise, return 1
0x8518             addi sp, sp, 8       # restore sp
0x851C             jr   ra              # return
0x8520 else:          addi a0, a0, -1    # n = n - 1
0x8524             jal   factorial      # recursive call
0x8528             lw    t1, 4(sp)        # restore n into t1
0x852C             lw    ra, 0(sp)        # restore ra
0x8530             addi sp, sp, 8       # restore sp
0x8534             mul   a0, t1, a0      # a0 = n*factorial(n-1)
0x8538             jr   ra              # return
```

PC+4 = 0x8528 when factorial is called recursively.

Stack During Recursive Function

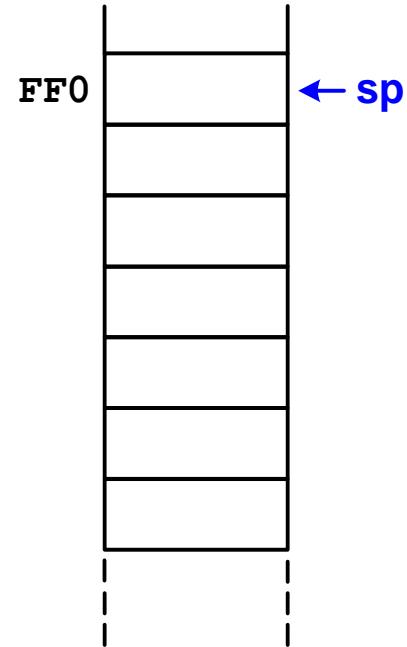
When **factorial (3)** is called:

Address Data



Before Calls

Address Data

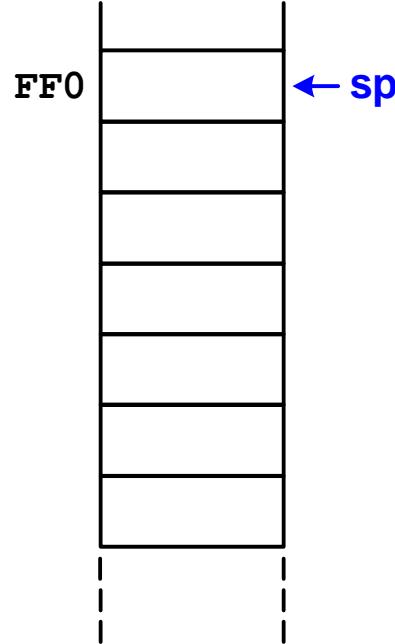


After Recursive Calls

Stack During Recursive Function

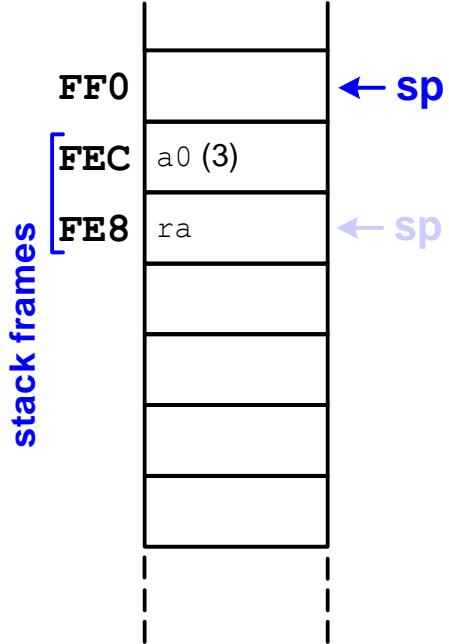
When **factorial (3)** is called:

Address Data



Before Calls

Address Data

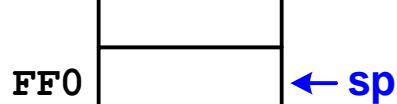


After Recursive Calls

Stack During Recursive Function

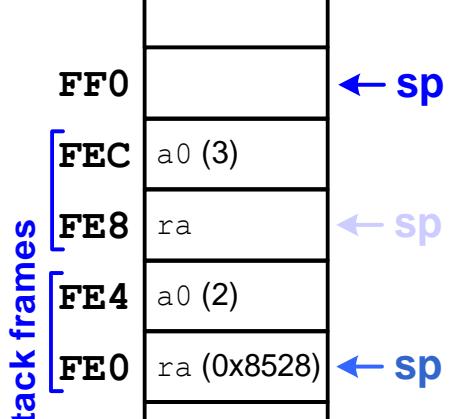
When **factorial (3)** is called:

Address Data



Before Calls

Address Data

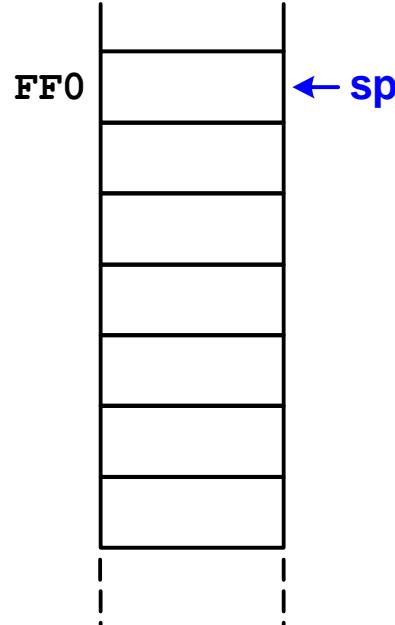


After Recursive Calls

Stack During Recursive Function

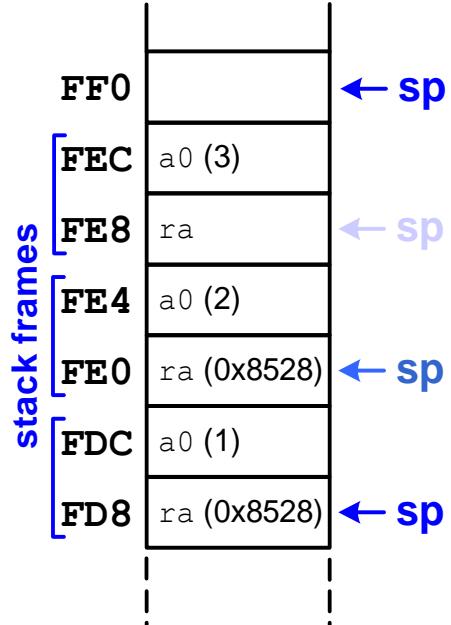
When **factorial (3)** is called:

Address Data



Before Calls

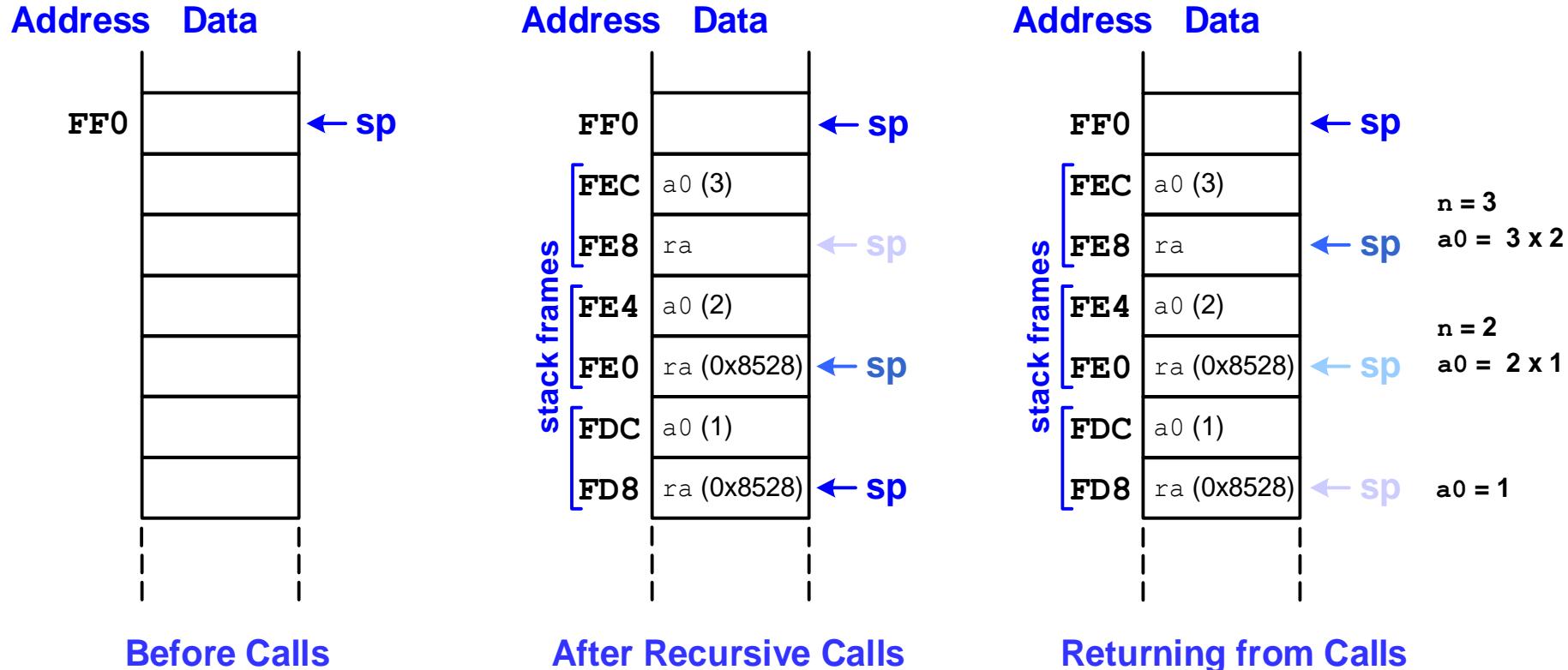
Address Data



After Recursive Calls

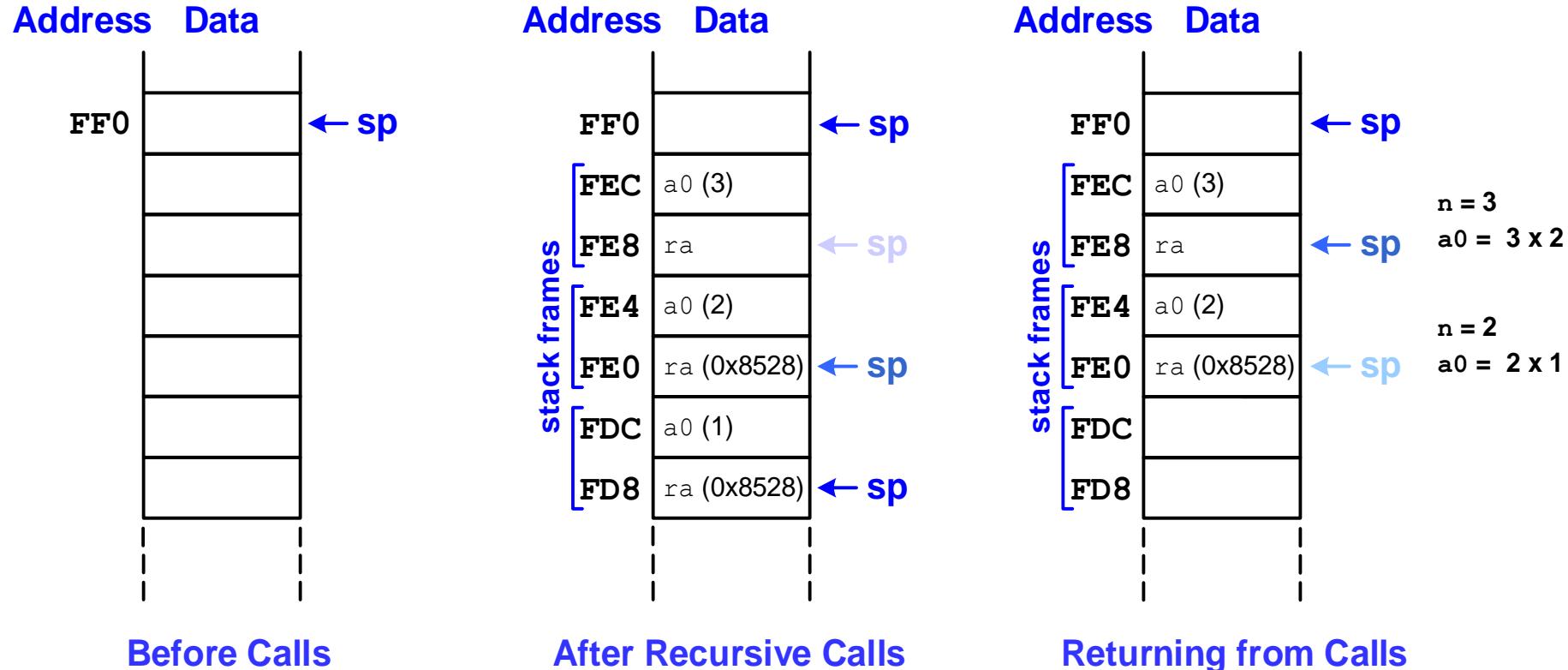
Stack During Recursive Function

When **factorial (3)** is called:



Stack During Recursive Function

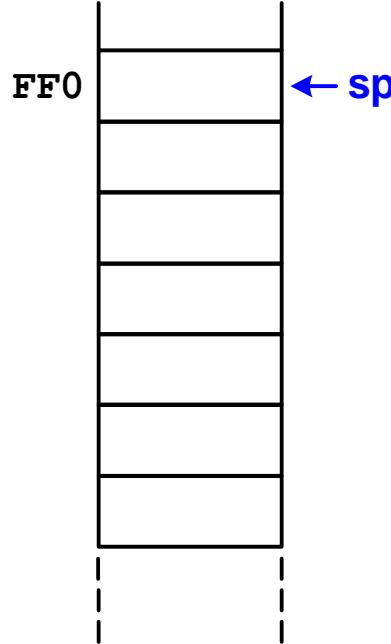
When **factorial (3)** is called:



Stack During Recursive Function

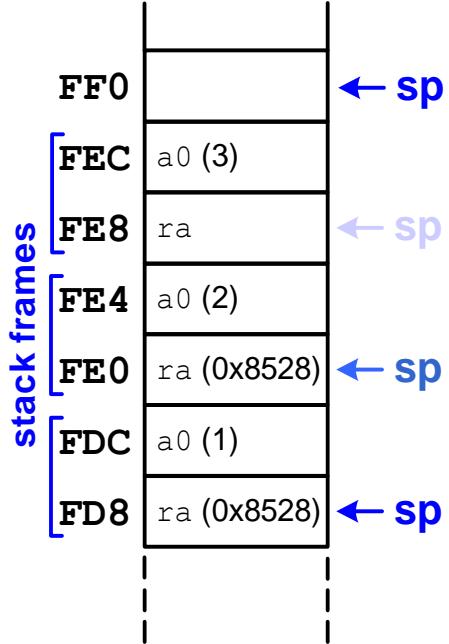
When **factorial (3)** is called:

Address Data



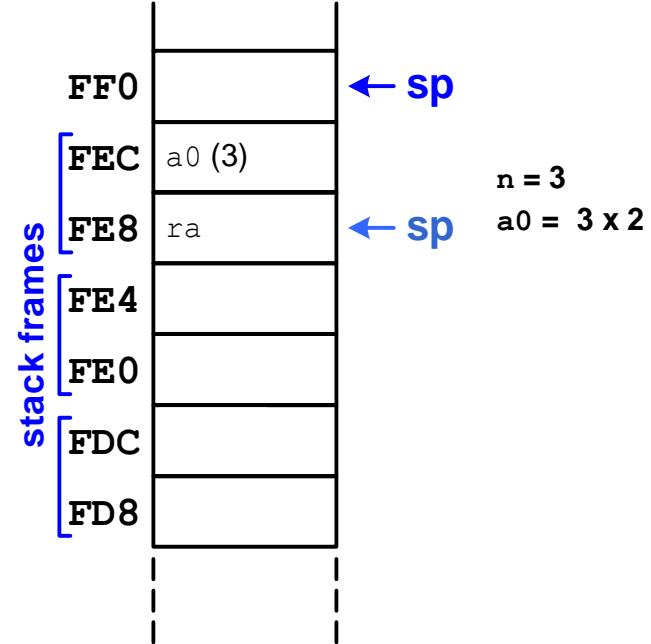
Before Calls

Address Data



After Recursive Calls

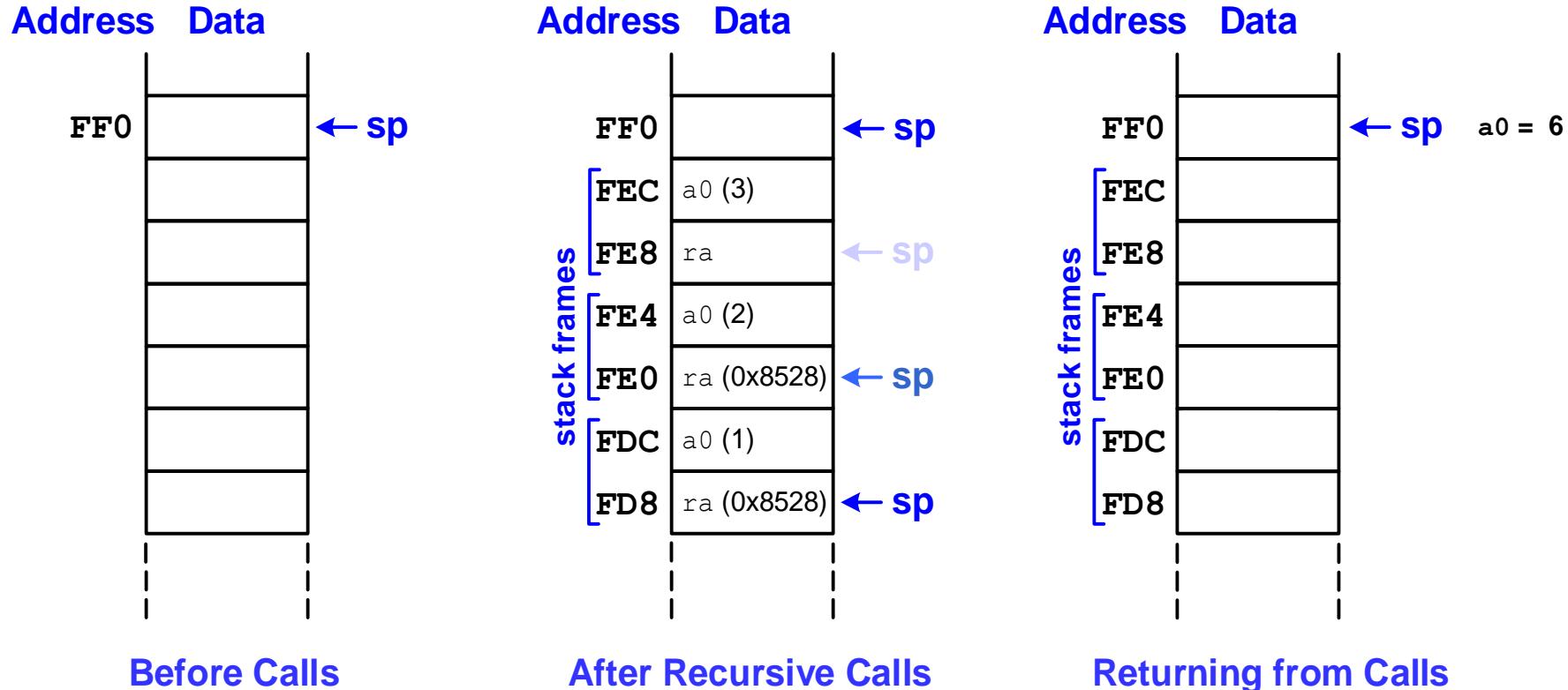
Address Data



Returning from Calls

Stack During Recursive Function

When **factorial (3)** is called:



The mul Instruction

- the RV32I (RISC-V 32-bit integer base instruction set) does not include the MUL (multiply) instruction.
- The RV32I instruction set only contains basic integer arithmetic, logic, control flow, and memory access instructions.
- To get access to the MUL instruction and other multiplication and division operations, you need the M (Multiply) extension, which is often referred to as RV32IM when combined with the base integer instruction set.
- The MUL instruction only keeps the lower 32 bits of the result. The upper bits are discarded. This means that for standard 32-bit multiplication, the result will be modulo 2^{32}
- However, RISC-V provides other instructions to handle cases where the full result (including the upper bits) is needed.

Chapter 6: Architecture

More on Jumps & Pseudoinstructions

Jumps

- RISC-V has two types of unconditional jumps
 - Jump and link (`jal rd, imm20:0`)
 - $rd = PC + 4$; $PC = PC + imm$
 - jump and link register (`jalr rd, rs, imm11:0`)
 - $rd = PC + 4$; $PC = [rs] + \text{SignExt}(imm)$

Pseudoinstructions

- **Pseudoinstructions** are not actual RISC-V instructions but they are often more convenient for the programmer.
- Assembler converts them to real RISC-V instructions.

Jump Pseudoinstructions

- RISC-V has four jump pseudoinstructions
 - j imm jal x0, imm
 - jal imm jal ra, imm
 - jr rs jalr x0, rs, 0
 - ret jr ra (i.e., jalr x0, ra, 0)

Labels

- Label indicates where to jump
- Represented in jump as immediate offset
 - **imm** = # bytes past jump instruction
 - In example, below, **imm** = (0x51C-0x300) = 0x21C
 - `jal simple` = `jal ra, 0x21C`

RISC-V assembly code

```
0x00000300 main:    jal  simple          # call
0x00000304          add   s0, s1, s1
...
0x0000051c simple:  jr    ra              # return
```

Long Jumps

- The immediate is limited in size
 - 20 bits for `jal`, 12 bits for `jalr`
 - Limits how far a program can jump
- Special instruction to help jumping further
 - `auipc rd, imm`: add upper immediate to PC
 - $rd = PC + \{imm_{31:12}, 12'b0\}$
- Pseudoinstruction: `call imm31:0`
 - Behaves like `jal imm`, but allows 32-bit immediate offset
 - `auipc ra, imm31:12`
 - `jalr ra, ra, imm11:0`

More RISC-V Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
j label	jal zero, label
jr ra	jalr zero, ra, 0
mv t5, s3	addi t5, s3, 0
not s7, t2	xori s7, t2, -1
nop	addi zero, zero, 0
li s8, 0x56789DEF	lui s8, 0x5678A addi s8, s8, 0xDEF
bgt s1, t3, L3	blt t3, s1, L3
bgez t2, L7	bge t2, zero, L7
call L1	auipc ra, imm _{31:12} jalr ra, ra, imm _{11:0}
ret	jalr zero, ra, 0

See Appendix B for more pseudoinstructions.

Chapter 6: Architecture

Machine Language

Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
 - Simplicity favors regularity: 32-bit data & instructions
- **4 Types of Instruction Formats:**
 - R-Type
 - I-Type
 - S/B-Type
 - U/J-Type

R-Type

- **Register-type**
- 3 register operands:
 - rs1, rs2: source registers
 - rd: destination register
- Other fields:
 - op: the *operation code* or *opcode*
 - funct7, funct3:
 - the *function* (7 bits and 3-bits, respectively)
 - with opcode, tells computer what operation to perform

R-Type

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

R-Type Examples

Assembly

```
add s2, s3, s4
add x18,x19,x20
sub t0, t1, t2
sub x5, x6, x7
```

Field Values

funct7	rs2	rs1	funct3	rd	op
0	20	19	0	18	51
32	7	6	0	5	51

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

Machine Code

funct7	rs2	rs1	funct3	rd	op
0000,000	10100	10011	000	1001,0	011,0011
0100,000	00111	00110	000	0010,1	011,0011

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

(0x01498933)

(0x407302B3)



RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

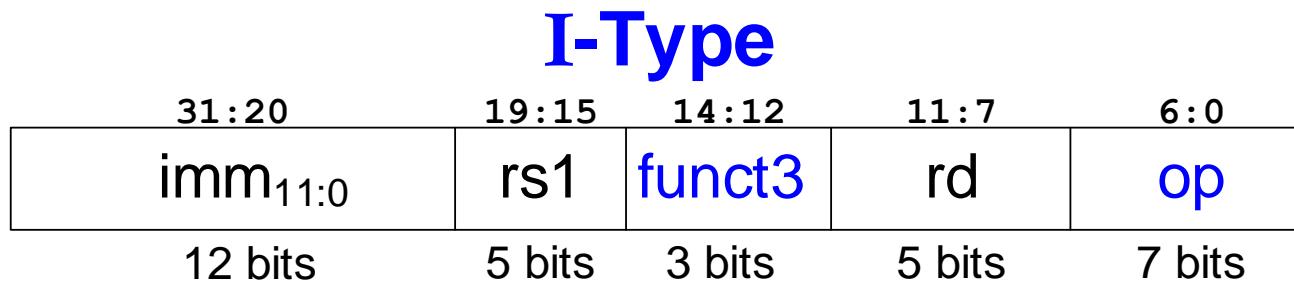
32-bit instruction stored in memory

Chapter 6: Architecture

Machine Language: More Formats

I-Type

- ***Immediate-type***
- 3 operands:
 - rs1: register source operand
 - rd: register destination operand
 - imm: 12-bit two's complement immediate
- Other fields:
 - op: the opcode
 - Simplicity favors regularity: all instructions have opcode
 - funct3: the function (3-bit function code)
 - with opcode, tells computer what operation to perform



I-Type Examples

Assembly

```

addi s0, s1, 12
addi x8, x9, 12
addi s2, t1, -14
addi x18,x6, -14
lw t2, -6($3)
lw x7, -6($19)
lh s1, 27(zero)
lh x9, 27($0)
lb s4, 0x1F($4)
lb x20,0x1F($20)

```

Field Values

	$imm_{11:0}$	rs1	funct3	rd	op
	12	9	0	8	19
	-14	6	0	18	19
	-6	19	2	7	3
	27	0	1	9	3
	0x1F	20	0	20	3
	12 bits	5 bits	3 bits	5 bits	7 bits

Machine Code

	$imm_{11:0}$	rs1	funct3	rd	op
	0000 0000 1100	01001	000	01000	001 0011
	1111 1111 0010	00110	000	10010	001 0011
	1111 1111 1010	10011	010	00111	000 0011
	0000 0001 1011	00000	001	01001	000 0011
	0000 0001 1111	10100	000	10100	000 0011
	12 bits	5 bits	3 bits	5 bits	7 bits

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

S/B-Type

- *Store-Type*
- *Branch-Type*
- Differ only in immediate encoding

31:25	24:20	19:15	14:12	11:7	6:0	
$\text{imm}_{11:5}$	rs2	rs1	funct3	$\text{imm}_{4:0}$	op	S-Type
$\text{imm}_{12,10:5}$	rs2	rs1	funct3	$\text{imm}_{4:1,11}$	op	B-Type

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

S-Type

- **Store-Type**
- 3 operands:
 - rs1: base register
 - rs2: value to be stored to memory
 - imm: 12-bit two's complement immediate
- Other fields:
 - op: the opcode
 - Simplicity favors regularity: all instructions have opcode
 - funct3: the function (3-bit function code)
 - with opcode, tells computer what operation to perform

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

S-Type Examples

Assembly

```

sw t2, -6($s3)
sw x7, -6($x19)
sh s4, 23($t0)
sh x20,23($x5)
sb t5, 0x2D(zero)
sb x30,0x2D($x0)

```

Field Values

imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
1111 111	7	19	2	11010	35
0000 000	20	5	1	10111	35
0000 001	30	0	0	01101	35

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

Machine Code

imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
1111 111	00111	10011	010	11010	010 0011
0000 000	10100	00101	001	10111	010 0011
0000 001	11110	00000	000	01101	010 0011

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

B-Type

- **Branch-Type** (similar format to S-Type)
- 3 operands:
 - rs1: register source 1
 - rs2: register source 2
 - $\text{imm}_{12:1}$: 12-bit two's complement immediate – address offset
- Other fields:
 - op: the opcode
 - Simplicity favors regularity: all instructions have opcode
 - funct3: the function (3-bit function code)
 - with opcode, tells computer what operation to perform

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$\text{imm}_{12,10:5}$	rs2	rs1	funct3	$\text{imm}_{4:1,11}$	op

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

B-Type Example

- The 13-bit immediate encodes where to branch (relative to the branch instruction)
- Immediate encoding is strange
- Example:**

RISC-V Assembly

```
0x70      beq    s0, t5, L1
0x74      add    s1, s2, s3
0x78      sub    s5, s6, s7
0x7C      lw      t0, 0(s1)
0x80 L1:  addi   s1, s1, -15
```

$$0x80 - 0x70 = 0x10 = 16$$

imm_{12:0} = 16 0 0 0 0 0 0 0 1 0 0 0 0
bit number 12 11 10 9 8 7 6 5 4 3 2 1 0

Assembly	Field Values						Machine Code					
	imm _{12:10:5}	rs2	rs1	funct3	imm _{4:1,11}	op	imm _{12:10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
beq s0, t5, L1	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011
beq x8, x30, 16	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011

B-Type Example

- The 13-bit immediate encodes where to branch (relative to the branch instruction)
- Immediate encoding is strange
- Example:**

RISC-V Assembly

```
0x70      beq    s0, t5, L1  ↗1  
0x74      add     s1, s2, s3 ↗2  
0x78      sub     s5, s6, s7 ↗3  
0x7C      lw      t0, 0(s1) ↗4  
0x80 L1: addi   s1, s1, -15 ↗4
```

$$0x80 - 0x70 = 0x10 = 16$$

L1 is 4 instructions (i.e., **16 bytes**) past beq

imm _{12:0} = 16	0	0	0	0	0	0	0	1	0	0	0	0
bit number	12	11	10	9	8	7	6	5	4	3	2	1

Assembly

Field Values

Machine Code

	imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op	imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op	
beq s0, t5, L1	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011	(0x01E40863)
beq x8, x30, 16		7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

U/J-Type

- *Upper-Immediate-Type*
- *Jump-Type*
- Differ only in immediate encoding

31:12	11:7	6:0	
$\text{imm}_{31:12}$	rd	op	U-Type
$\text{imm}_{20,10:1,11,19:12}$	rd	op	J-Type

20 bits 5 bits 7 bits

U-Type

- ***Upper-immediate-Type***
- Used for load upper immediate (lui)
- 2 operands:
 - rd: destination register
 - $\text{imm}_{31:12}$:upper 20 bits of a 32-bit immediate
- Other fields:
 - op: the *operation code* or *opcode* – tells computer what operation to perform

U-Type

31:12	11:7	6:0
$\text{imm}_{31:12}$	rd	op
20 bits	5 bits	7 bits

U-Type Example

- **Upper-immediate-Type**
- Used for load upper immediate (lui)
- 2 operands:
 - rd: destination register
 - imm_{31:12}:upper 20 bits of a 32-bit immediate
- Other fields:
 - op: the *operation code* or *opcode* – tells computer what operation to perform

Assembly

```
lui s5, 0x8CDEF  
lui x21, 0x8CDEF
```

Field Values

imm _{31:12}	rd	op
0x8CDEF	21	55
20 bits	5 bits	7 bits

Machine Code

imm _{31:12}	rd	op
1000 1100 1101 1110 1111	10101	011 0111
20 bits	5 bits	7 bits

(0x8CDEFAB7)

J-Type

- ***Jump-Type***
- Used for jump-and-link instruction (`jal`)
- 2 operands:
 - `rd`: destination register
 - $\text{imm}_{20,10:1,11,19:12}$: 20 bits (20:1) of a 21-bit immediate
- Other fields:
 - `op`: the operation code or opcode – tells computer what operation to perform

J-Type

31:12	11:7	6:0
$\text{imm}_{20,10:1,11,19:12}$	<code>rd</code>	<code>op</code>

20 bits 5 bits 7 bits

- Note: `jalr` is I-type, not j-type, to specify `rs1`

J-Type Example

# Address	RISC-V Assembly
0x0000540C	jal ra, func1
0x00005410	add s1, s2, s3
...	...
0x000ABC04	func1: add s4, s5, s8
...	...
	func1 is 0xA67F8 bytes past jal

imm = 0xA67F8 0 1 0 1 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1
bit number 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Assembly	Field Values	Machine Code	
jal ra, func1 jal x1, 0xA67F8	imm _{20,10:1,11,19:12} 0111 1111 1000 1010 0110 20 bits	rd op 1 111 5 bits 7 bits imm _{20,10:1,11,19:12} 0111 1111 1000 1010 0110 20 bits	rd op 00001 110 1111 5 bits 7 bits (0x7F8A60EF)

$$0x000ABC04 - 0x0000540C = 0x000A67F8$$

J-Type Example

# Address	RISC-V Assembly
0x0000540C	jal ra, func1
0x00005410	add s1, s2, s3
...	...
0x000ABC04	func1: add s4, s5, s8
...	...
func1 is 0xA67F8 bytes past jal	

$$0xABC04 - 0x540C = \\ \text{0xA67F8}$$

imm = 0xA67F8 0 1 0 1 0 0 1 1 1 0 0 1 1 1 1 1 1 0
bit number 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Assembly	Field Values	Machine Code												
jal ra, func1 jal x1, 0xA67F8	imm _{20,10:1,11,19:12} <table border="1"><tr><td>0111 1111 1000 1010 0110</td><td>rd 1</td><td>op 111</td></tr><tr><td>20 bits</td><td>5 bits</td><td>7 bits</td></tr></table>	0111 1111 1000 1010 0110	rd 1	op 111	20 bits	5 bits	7 bits	imm _{20,10:1,11,19:12} <table border="1"><tr><td>0111 1111 1000 1010 0110</td><td>rd 00001</td><td>op 110 1111</td></tr><tr><td>20 bits</td><td>5 bits</td><td>7 bits</td></tr></table> (0x7F8A60EF)	0111 1111 1000 1010 0110	rd 00001	op 110 1111	20 bits	5 bits	7 bits
0111 1111 1000 1010 0110	rd 1	op 111												
20 bits	5 bits	7 bits												
0111 1111 1000 1010 0110	rd 00001	op 110 1111												
20 bits	5 bits	7 bits												

Review: Instruction Formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	R-Type
imm _{11:0}		rs1	funct3	rd	op	I-Type
imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op	S-Type
imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op	B-Type
	imm _{31:12}			rd	op	U-Type
	imm _{20,10:1,11,19:12}			rd	op	J-Type
20 bits				5 bits	7 bits	

R-type (*register-type*) instructions
I-type (*immediate*) instructions
S/B-type (*store/branch*) instructions
U/J-type (*upper immediate/jump*) instructions

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
 - add, sub: use 3 register operands
 - lw, sw, addi: use 2 register operands and a constant
- Number of instruction formats kept small
 - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

- End of week 6 slides

Chapter 6: Architecture

Immediate Encodings

Constants / Immediates

- `lw` and `sw` use constants or *immediates*
- *immediately* available from instruction
- 12-bit two's complement number
- `addi`: add immediate

C Code

```
a = a + 4;  
b = a - 12;
```

RISC-V assembly code

```
# s0 = a, s1 = b  
addi s0, s0, 4  
addi s1, s0, -12
```

Constants / Immediates

- `lw` and `sw` use constants or *immediates*
- *immediately* available from instruction
- 12-bit two's complement number
- `addi`: add immediate
- **Is subtract immediate (`subi`) necessary?**

C Code

```
a = a + 4;  
b = a - 12;
```

RISC-V assembly code

```
# s0 = a, s1 = b  
addi s0, s0, 4  
addi s1, s0, -12
```

Constants / Immediates

Immediate Bits

imm_{11}	$\text{imm}_{11:1}$	imm_0	I
imm_{12}	$\text{imm}_{11:1}$	0	S
$\text{imm}_{31:21}$	$\text{imm}_{20:12}$	0	B
imm_{20}	$\text{imm}_{20:12}$	$\text{imm}_{11:1}$	U
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0			J

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
funct7	rs2	rs1	funct3	rd	op
imm _{11:0}	rs1	funct3	rd	op	
imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
imm _{31:12}				rd	op
imm _{20,10:1,11,19:12}				rd	op
20 bits			5 bits	7 bits	

R-Type
I-Type
S-Type
B-Type
U-Type
J-Type

R-type (*register-type*) instructions
I-type (*immediate*) instructions
S/B-type (*store/branch*) instructions
U/J-type (*upper immediate/jump*) instructions

- Immediate bits are extracted from an instruction and sign-extended to 32 bits.

Immediate Encodings

Instruction Bits

funct7	4 3 2 1 0	rs1	funct3	rd	R
11 10 9 8 7 6 5	4 3 2 1 0	rs1	funct3	rd	I
11 10 9 8 7 6 5	rs2	rs1	funct3	4 3 2 1 0	S
12 10 9 8 7 6 5	rs2	rs1	funct3	4 3 2 1 11	B
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12				rd	U
20 10 9 8 7 6 5 4 3 2 1 11 19 18 17 16 15 14 13 12				rd	J
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7					

- Immediate bits *mostly* occupy **consistent instruction bits**.
 - Simplifies hardware to build the microprocessor
- **Sign bit** of signed immediate is in **msb** of instruction.
- Recall that **rs2** of R-type can encode immediate shift amount.

Composition of 32-bit Immediates

Instruction Bits

funct7	4 3 2 1 0	rs1	funct3	rd	R
11 10 9 8 7 6 5	4 3 2 1 0	rs1	funct3	rd	I
11 10 9 8 7 6 5	rs2	rs1	funct3	4 3 2 1 0	S
12 10 9 8 7 6 5	rs2	rs1	funct3	4 3 2 1 11	B
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12				rd	U
20 10 9 8 7 6 5 4 3 2 1 11 19 18 17 16 15 14 13 12				rd	J
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7					

instruction bit	31	31	31	31	30:25	24:21	20	I
	31	31	31	31	30:25	11:8	7	S
	31	31	31	7	30:25	11:8	0	B
	31	30:20	19:12	0	0	0	0	U
	31	31	19:12	20	30:25	24:21	0	J
	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0							

Immediate Bits

Chapter 6: Architecture

**Reading
Machine Language &
Addressing Operands**

Instruction Fields & Formats

Instruction	op	funct3	Funct7	Type
add	0110011 (51)	000 (0)	0000000 (0)	R-Type
sub	0110011 (51)	000 (0)	0100000 (32)	R-Type
and	0110011 (51)	111 (7)	0000000 (0)	R-Type
or	0110011 (51)	110 (6)	0000000 (0)	R-Type
addi	0010011 (19)	000 (0)	-	I-Type
beq	1100011 (99)	000 (0)	-	B-Type
bne	1100011 (99)	001 (1)	-	B-Type
lw	0000011 (3)	010 (2)	-	I-Type
sw	0100011 (35)	010 (2)	-	S-Type
jal	1101111 (111)	-	-	J-Type
jalr	1100111 (103)	000 (0)	-	I-Type
lui	0110111 (55)	-	-	U-Type

See Appendix B for other instruction encodings

Interpreting Machine Code

- Write in binary
- Start with **op**: tells how to parse rest
- Extract fields
- **op**, **funct3**, and **funct7** fields tell operation
- Ex: 0x41FE83B3 and 0xFDA58393

0x41FE83B3: 0100 0001 1111 1110 1000 0011 1011 0011
 op = 51, funct3 = 0: add or sub (R-type)
 funct7 = 0100000: sub

0xFDA48393: 1111 1101 1010 0100 1000 0011 1001 0011
 op = 19, funct3 = 0: addi (I-type)

Interpreting Machine Code

- Write in binary
- Start with **op**: tells how to parse rest
- Extract fields
- **op**, **funct3**, and **funct7** fields tell operation
- Ex: 0x41FE83B3 and 0xFDA48393

	Machine Code						Field Values						Assembly	
(0x41FE83B3)	funct7 0100 000 7 bits	rs2 11111 5 bits	rs1 11101 5 bits	funct3 000 3 bits	rd 00111 5 bits	op 011 0011 7 bits	funct7 32 7 bits	rs2 31 5 bits	rs1 29 5 bits	funct3 0 3 bits	rd 7 5 bits	op 51 7 bits	sub x7, x29, x31 sub t2, t4, t6	
(0xFDA48393)	imm _{11:0} 1111 1101 1010 12 bits	rs1 01001 5 bits	funct3 000 3 bits	rd 00111 5 bits	op 001 0011 7 bits	imm _{11:0} -38 12 bits	rs1 9 5 bits	funct3 0 3 bits	rd 7 5 bits	op 19 7 bits	addi x7, x9, -38 addi t2, s1, -38			

Value in the Immediate field is in two's complement format
Invert 1111 1101 1010
+1 0000 0010 0101
= 1
0000 0010 0110 = 38, so (1111 1101 1010)₂ = (-38)₁₀

Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative

Addressing Modes

Register Only

- Operands found in registers
 - **Example:** add s0, t2, t3
 - **Example:** sub t6, s1, zero

Immediate

- 12-bit signed immediate used as an operand
 - **Example:** addi s4, t5, -73
 - **Example:** ori t3, t7, 0xFF

Addressing Modes

Base Addressing

- Loads and Stores
- Address of operand is:

base address + immediate

– **Example:** lw s4, 72(zero)

- address = 0 + 72

– **Example:** sw t2, -25(t1)

- address = t1 - 25

Addressing Modes

PC-Relative Addressing: branches and jal

Example:

Address	Instruction
0x354	L1: addi s1, s1, 1
0x358	sub t0, t1, s7
...	...
0xEB0	bne s8, s9, L1

The label is $(0xEB0 - 0x354) = 0xB5C$ (**2908**) instructions before bne

$\text{imm}_{12:0} = -2908$ 1 0 1 0 0 1 0 1 0 0 1 0
bit number 12 11 10 9 8 7 6 5 4 3 2 1 0

Assembly

	imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
bne s8, s9, L1	1100 101	24	25	1	0010 0	99
(bne x24, x25, L1)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Field Values

Machine Code

	imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
	1100 101	11000	11001	001	0010 0	110 0011
						(0xCB8C9263)

Chapter 6: Architecture

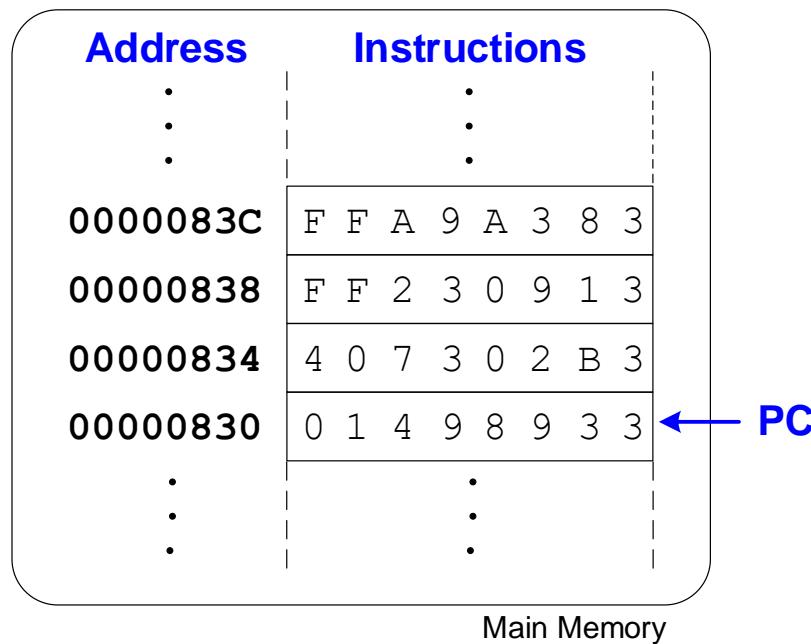
Compiling, Assembling, & Loading Programs

The Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
 - No rewiring required
 - Simply store new program in memory
- Program Execution:
 - Processor *fetches* (reads) instructions from memory in sequence
 - Processor performs the specified operation

The Stored Program

Assembly Code	Machine Code
add s2, s3, s4	0x01498933
sub t0, t1, t2	0x407302B3
addi s2, t1, -14	0xFF230913
lw t2, -6(s3)	0xFFA9A383



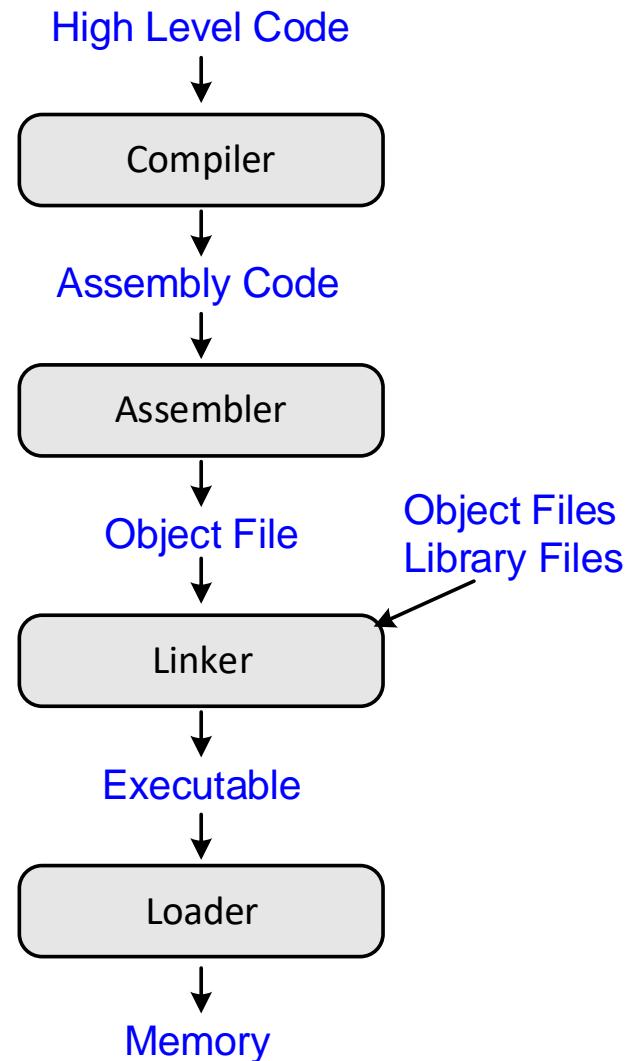
Program Counter (PC): keeps track of current instruction

Alan Turing, 1912 - 1954

- British mathematician and computer scientist
- Founder of theoretical computer science
- Invented the Turing machine: a mathematical model of computation
- Designed the Automatic Computing Engine, one of the first stored program computers
- In 1952, was prosecuted for homosexual acts. Two years later, he died of cyanide poisoning.
- The Turing Award, named in his honour, is the highest distinction in computing.



How to Compile & Run a Program



Grace Hopper, 1906 - 1992

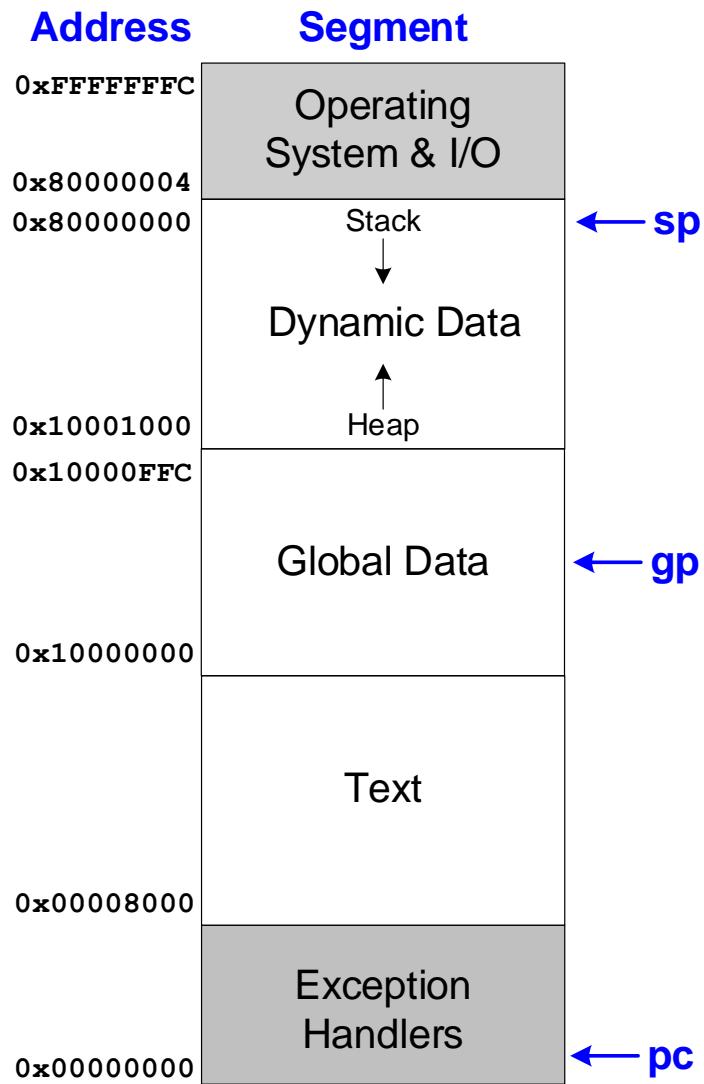
- Graduated from Yale University with a Ph.D. in mathematics
- Developed first compiler
- Helped develop the COBOL programming language
- Highly awarded naval officer
- Received World War II Victory Medal and National Defense Service Medal, among others



What is Stored in Memory?

- **Instructions** (also called *text*)
- **Data**
 - **Global/static**: allocated before program begins
 - **Dynamic**: allocated within program
- How **big** is memory?
 - At most $2^{32} = 4$ gigabytes (4 GB)
 - From address 0x00000000 to 0xFFFFFFFF

Example RISC-V Memory Map



Chapter 6: Architecture

Endianness

Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address is the same** for big- or little-endian

Big-Endian				Little-Endian			
Byte Address				Word Address			
	:				:		
C	D	E	F	C	F	E	D
8	9	A	B	8	B	A	9
4	5	6	7	4	7	6	5
0	1	2	3	0	3	2	1
MSB		LSB		MSB		LSB	

Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!

Big-Endian				Little-Endian			
Byte Address				Word Address			
	:				:		
	C	D	E	F			
8	8	9	A	B	8	F	E
4	4	5	6	7	4	B	A
0	0	1	2	3	0	9	8
	MSB	LSB			MSB	LSB	

Big-Endian & Little-Endian Example

- Suppose t0 initially contains 0x23456789
- After following code runs on **big-endian** system, what value is s0?

```
sw t0, 0(zero)
lb s0, 1(zero)
```

Big-Endian & Little-Endian Example

- Suppose t_0 initially contains 0x23456789
- After following code runs on **big-endian** system, what value is s_0 ?

```
sw t0, 0(zero)
```

```
lb s0, 1(zero)
```

- **Big-endian:** $s_0 = 0x00000045$

Big-Endian

Byte Address	0	1	2	3	Word Address
Data Value	23	45	67	89	0
	MSB		LSB		

Big-Endian & Little-Endian Example

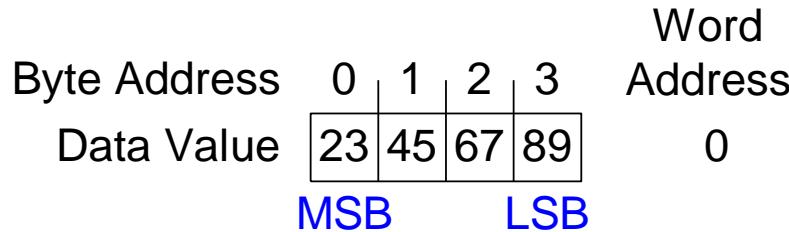
- Suppose t_0 initially contains 0x23456789
- After following code runs on **big-endian** system, what value is s_0 ?
- In a **little-endian** system?

sw t_0 , 0 (zero)

lb s_0 , 1 (zero)

- **Big-endian:** $s_0 = 0x00000045$

Big-Endian



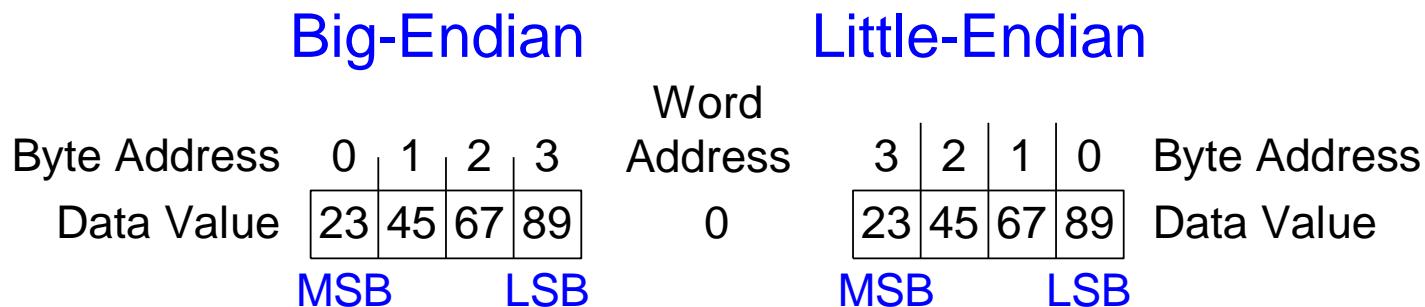
Big-Endian & Little-Endian Example

- Suppose t_0 initially contains 0x23456789
- After following code runs on **big-endian** system, what value is s_0 ?
- In a **little-endian** system?

sw t_0 , 0 (zero)

lb s_0 , 1 (zero)

- **Big-endian:** $s_0 = 0x00000045$
- **Little-endian:** $s_0 = 0x00000067$



Chapter 6: Architecture

Signed & Unsigned Instructions

Signed & Unsigned Instructions

- Multiplication and division
- Branches
- Set less than
- Loads
- Detecting overflow

Multiplication

- **Signed:** mulh
- **Unsigned:** mulhu, mulhsu
 - mulhu: treat both operands as unsigned
 - mulhsu: treat first operand as signed, second as unsigned
 - 32 lsbs are identical whether signed/unsigned; use mul

Example: $s1 = 0x80000000$; $s2 = 0xC0000000$

mulh s4, s1, s2
mul s3, s1, s2

$s1 = -2^{31}; s2 = -2^{30}$
 $s1 \times s2 = 2^{61}$
 $s4 = 0x20000000$
 $s3 = 0x00000000$

mulhu s4, s1, s2
mul s3, s1, s2

$s1 = 2^{31}; s2 = 3 \times 2^{30}$
 $s1 \times s2 = 3 \times 2^{61}$
 $s4 = 0x60000000$
 $s3 = 0x00000000$

mulhsu s4, s1, s2
mul s3, s1, s2

$s1 = -2^{31}; s2 = 3 \times 2^{30}$
 $s1 \times s2 = -3 \times 2^{61}$
 $s4 = 0xA0000000$
 $s3 = 0x00000000$

Division & Remainder

- **Signed:** `div, rem`
- **Unsigned:** `divu, remu`

Branches

- **Signed:** blt, bge
- **Unsigned:** bltu, bgeu

Examples: s1 = 0x80000000; s2 = 0x40000000

blt s1, s2
s1 = - 2^{31} ; s2 = 2^{30}
taken

bltu s1, s2
s1 = 2^{31} ; s2 = 2^{30}
not taken

Set Less Than

- **Signed:** slt, slti
- **Unsigned:** sltu, sltiu

Note: RISC-V always **sign-extends** the immediate, even for sltiu

Examples: s1 = 0x80000000; s2 = 0x40000000

slt t0, s1, s2
s1 = -2^{31} ; s2 = 2^{30}
t0 = 1

slti t2, s1, -1 # -1 = 0xFFFF
s1 = -2^{31} ; imm = 0xFFFFFFFF = -1
t2 = 1

sltu t1, s1, s2
s1 = 2^{31} ; s2 = 2^{30}
t1 = 0

sltiu t3, s1, -1 # -1 = 0xFFFF
s1 = 2^{31} ; imm = 0xFFFFFFFF = $2^{32} - 1$
t3 = 1

Loads

- **Signed:**

- Sign-extends to create 32-bit value to load into register
- Load halfword: lh
- Load byte: lb

- **Unsigned:**

- Zero-extends to create 32-bit value
- Load halfword unsigned: lhu
- Load byte: bu

Detecting Overflow

- RISC-V does not provide unsigned addition or instructions or overflow detection because it can be done with existing instructions:
- **Example: Detecting unsigned overflow:**

```
add t0, t1, t2  
bltu t0, t1, overflow
```

- **Example: Detecting signed overflow:**

```
add t0, t1, t2  
slti t3, t2, 0          # t3=1 if t2 neg.  
slt t4, t0, t1          # t4=1 if result < t1  
bne t3, t4, overflow   # overflow if:  
                        # t2 neg & result>=t1 or  
                        # t2 pos & result<t1
```

Chapter 6: Architecture

Compressed Instructions

Compressed Instructions

- **16-bit** RISC-V instructions
- Replace common integer and floating-point instructions with 16-bit versions.
- Most RISC-V compilers/processors can use a **mix** of 32-bit and 16-bit instructions (and use 16-bit instructions whenever possible).
- Uses prefix: **c**.
- **Examples:**
 - add → c.add
 - lw → c.lw
 - addi → c.addi

Compressed Instructions Example

C Code

```
int i;  
int scores[200];  
  
for (i=0; i<200; i=i+1)  
    scores[i] = scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i  
  
c.li s1, 0          # i = 0  
addi t2, zero, 200  # t2 = 200  
  
for:  
    bge s1, t2, done  # I >= 200? done  
    c.lw a3, 0(s0)   # a3 = scores[i]  
    c.addi a3, 10     # a3 = scores[i]+10  
    c.sw a3, 0(s0)   # scores[i] = a3  
    c.addi s0, 4      # next element  
    c.addi s1, 1      # i = i+1  
    c.j for           # repeat  
  
done:
```

Compressed Instructions Example

C Code

```
int i;  
int scores[200];
```

```
for (i=0; i<200; i=i+1)
```

```
    scores[i] = scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i
```

```
c.li s1, 0          # i = 0  
addi t2, zero, 200  # t2 = 200
```

```
for:
```

```
bge s1, t2, done   # I >= 200? done  
c.lw a3, 0(s0)     # a3 = scores[i]  
c.addi a3, 10       # a3 = scores[i]+10  
c.sw a3, 0(s0)      # scores[i] = a3  
c.addi s0, 4         # next element  
c.addi s1, 1         # i = i+1  
c.j for             # repeat
```

```
done:
```

- 200 is too big to fit in compressed immediate, so noncompressed addi used instead.
- **c.addi s0, 4** is equivalent to **addi s0, s0, 4**.
- c.bge doesn't exist, so bge is used.

Compressed Machine Formats

- Some compressed instructions use a **3-bit register code** (instead of 5-bit). These specify registers x8 to x15.
- **Immediates** are 6-11 bits.
- **Opcode** is 2 bits.

Compressed Machine Formats

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0										
funct4				rd/rs1				rs2				op													
funct3	imm	rd/rs1				imm				op				CR-Type											
funct3	imm	rs1'			imm			rs2'			op				CI-Type										
funct6				rd'/rs1'			funct2	rs2'			op				CS-Type										
funct3	imm	rs1'			imm			imm			op				CS'-Type										
funct3	imm	funct	rd'/rs1'			imm			imm			op				CB-Type									
funct3	imm														CB'-Type										
funct3	imm														CJ-Type										
funct3	imm														CSS-Type										
funct3	imm														CIW-Type										
funct3	imm	rs1'				imm			rd'			op				CL-Type									
funct3	imm	rs1'			imm			rd'			op														

Chapter 6: Architecture

Floating-Point Instructions

RISC-V Floating-Point Extensions

- RISC-V offers three floating point extensions:
 - **RVF**: single-precision (32-bit)
 - 8 exponent bits, 23 fraction bits
 - **RVD**: double-precision (64-bit)
 - 11 exponent bits, 52 fraction bits
 - **RVQ**: quad-precision (128-bit)
 - 15 exponent bits, 112 fraction bits

Floating-Point Registers

- **32** Floating point registers
- **Width** is the highest precision – for example if RVQ is implemented, registers are 128 bits wide
- When multiple floating point extensions are implemented, the lower-precision values occupy the lower bits of the register

Floating-Point Registers

Name	Register Number	Usage
ft0-7	f0-7	Temporary variables
fs0-1	f8-9	Saved variables
fa0-1	f10-11	Function arguments/Return values
fa2-7	f12-17	Function arguments
fs2-11	f18-27	Saved variables
ft8-11	f28-31	Temporary variables

Floating-Point Instructions

- Append .s (single), .d (double), .q (quad) for precision. I.e., fadd.s, fadd.d, and fadd.q
- **Arithmetic operations:**
fadd, fsub, fdiv, fsqrt, fmin, fmax, multiply-add
(fmadd, fmsub, fnmadd, fnmsub)
- **Other instructions:**
move (fmv.x.w, fmv.w.x)
convert (fcvt.w.s, fcvt.s.w, etc.)
comparison (feq, flt, fle)
classify (fclass)
sign injection (fsgnj, fsgnjn, fsgnjx)

See Appendix B for additional RISC-V floating-point instructions.

Floating-Point Multiply-Add

- fmadd is the most critical instruction for signal processing programs.
- Requires four registers.

```
fmadd.f f1, f2, f3, f4      # f1 = f2 x f3 + f4
```

Floating-Point Example

C Code

```
int i;  
float scores[200];  
  
for (i=0; i<200; i=i+1)  
    scores[i]=scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i  
addi s1, zero, 0          # i = 0  
addi t2, zero, 200        # t2 = 200  
addi t0, zero, 10         # ft0 = 10.0  
fcvt.s.w ft0, t0  
  
for:  
    bge    s1, t2, done      # i>=200? done  
    slli   t0, s1, 2          # t0 = i*4  
    add    t0, t0, s0         # scores[i] address  
    f lw ft1, 0(t0)          # ft1=scores[i]  
    fadd.s ft1, ft1, ft0    # ft1=scores[i]+10  
    f sw ft1, 0(t0)          # scores[i] = t1  
    addi   s1, s1, 1          # i = i+1  
    j      for                # repeat  
  
done:
```

Floating-Point Instruction Formats

- Use R-, I-, and S-type formats
- Introduce another format for multiply-add instructions that have 4 register operands:
R4-type

R4-Type

31:27	26:25	24:20	19:15	14:12	11:7	6:0
rs3	funct2	rs2	rs1	funct3	rd	op
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Chapter 6: Architecture

Exceptions

Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
 - Hardware, also called an *interrupt*, e.g., keyboard
 - Software, also called *traps*, e.g., undefined instruction
- When exception occurs, the processor:
 - Records the cause of the exception
 - Jumps to exception handler
 - Returns to the program

Exception Causes

Exception	Cause
Instruction address misaligned	0
Instruction access fault	1
Illegal instruction	2
Breakpoint	3
Load address misaligned	4
Load access fault	5
Store address misaligned	6
Store access fault	7
Environment call from U-Mode	8
Environment call from S-Mode	9
Environment call from M-Mode	11

RISC-V Privilege Levels

- In RISC-V, exceptions occur at various **privilege levels**.
- Privilege levels limit access to memory or certain (privileged) instructions.
- RISC-V privilege modes are (from highest to lowest):
 - **Machine** mode (bare metal)
 - **System** mode (operating system)
 - **User** mode (user program)
 - **Hypervisor** mode (to support virtual machines)
- For example, a program running in M-mode (machine mode) can access all memory or instructions – it has the highest privilege level.

Exception Registers

- Each privilege level has registers to handle exceptions
- These registers are called control and status registers (**CSRRs**)
- We discuss **M-mode** (machine mode) exceptions, but other modes are similar
- M-mode registers used to handle exceptions are:
 - mtvec, mcause, mepc, mscratch

(Likewise, S-mode exception registers are: stvec, scause, sepc, and mscratch; and so on for the other modes.)

Exception Registers

- CSRRs are not part of register file
- M-mode CSRRs used to handle exceptions
 - **mtvec**: holds address of exception handler code
 - **mcause**: Records cause of exception
 - **mepc** (Exception PC): Records PC where exception occurred
 - **mscratch**: scratch space in memory for exception handlers

Exception-Related Instructions

- Called **privileged instructions** (because they access CSRRs)
 - **csrr**: CSR register read
 - **csrw**: CSR register write
 - **csrrw**: CSR register read/write
 - **mret**: returns to address held in mepc
- **Examples:**

```
csrr t1, mcause          # t1 = mcause
csrw mepc, t2            # mepc = t2
cwrrw t0, mscratch, t1  # t0 = mscratch
                           # mscratch = t1
```

Exception Handler Summary

- When a processor **detects an exception**:
 - It **jumps to exception handler** address in mtvec
 - The exception handler then:
 - **saves registers** on small stack pointed to by mscratch
 - Uses csrr (CSR read) to **look at cause** of exception (in mcause)
 - **Handles exception**
 - When finished, optionally **increments mepc** by 4 and **restores registers** from memory
 - And then either **aborts** the program or **returns to user code** (using mret, which returns to address held in mepc)

Example Exception Handler Code

- Check for **two types of exceptions**:
 - **Illegal instruction** (`mcause = 2`)
 - **Load address misaligned** (`mcause = 4`)

Example Exception Handler Code

```
# save registers that will be overwritten
    csrrw t0, mscratch, t0      # swap t0 and mscratch
    sw    t1, 0(t0)             # [mscratch] = t1
    sw    t2, 4(t0)             # [mscratch+4] = t2

# check cause of exception
    csrr  t1, mcause           # t1=mcause
    addi  t2, x0, 2             # t2=2 (illegal instruction exception code)

illegalinstr:
    bne   t1, t2, checkother  # branch if not an illegal instruction
    csrr  t2, mepc             # t2=exception PC
    addi  t2, t2, 4             # increment exception PC
    csrw  mepc, t2             # mepc=t2
    j     done                 # restore registers and return

checkother:
    addi  t2, x0, 4             # t2=4 (load address misaligned exception code)
    bne   t1, t2, done          # branch if not a misaligned load
    j     exit                 # exit program

# restore registers and return from the exception
done:
    lw    t1, 0(t0)             # t1 = [mscratch]
    lw    t2, 4(t0)             # t2 = [mscratch+4]
    csrrw t0, mscratch, t0     # swap t0 and mscratch
    mret                         # return to program

exit:
    ...

Checks for two types of exceptions:
• Illegal instruction (mcause = 2)
• Load address misaligned (mcause = 4)
```

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.

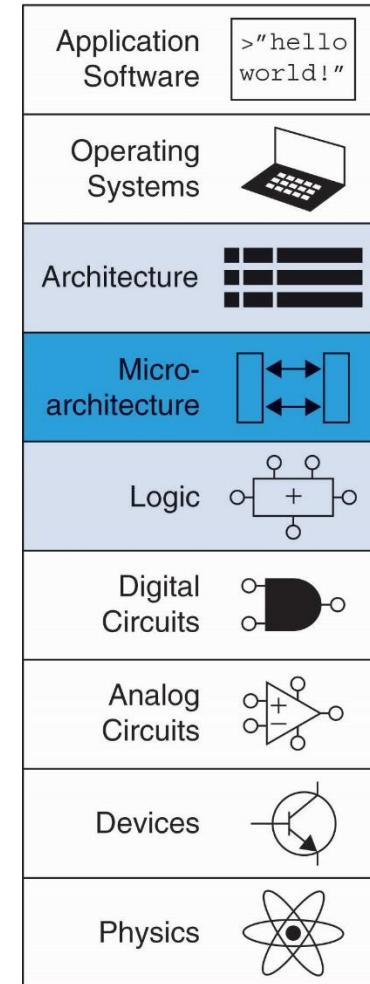
Digital Design & Computer Architecture

Sarah Harris & David Harris

Chapter 7: **Microarchitecture**

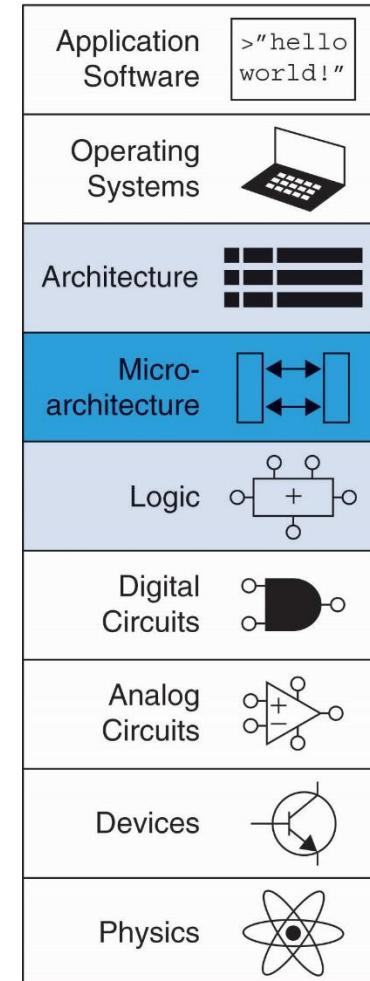
Chapter 7 :: Topics

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Advanced Microarchitecture**



Introduction

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
 - **Datapath:** functional blocks
 - **Control:** control signals



Microarchitecture

- **Multiple implementations** for a single architecture:
 - **Single-cycle**: Each instruction executes in a single cycle
 - **Multicycle**: Each instruction is broken up into series of shorter steps
 - **Pipelined**: Each instruction broken up into series of steps & multiple instructions execute at once

Single-cycle Microarchitecture

- Executes an entire instruction in one cycle
- The cycle time is limited by the slowest instruction.
- Requires separate instruction and data memories
- Easiest to understand, but
- Generally unrealistic

Multicycle Microarchitecture

- Executes an instruction in a series of shorter cycles.
- Simpler instructions execute in fewer cycles than complicated ones.
- Reduce the hardware cost by reusing expensive hardware blocks, e.g. adders, and memories.
 - E.g. an adder can be used on different cycles for several purposes while carrying out a single instruction.
 - Achieved by introducing several non-architectural registers to hold intermediate results.
- Executes one instruction at a time, each instruction takes multiple clock cycles.
- Requires only one memory, accessing it on one cycle to fetch the instruction and on another to read or write data.
- Use less hardware than single-cycle processors.
- Historical choice for inexpensive systems.

Pipelined Microarchitecture

- Applies pipelining to the single-cycle microarchitecture.
- Can execute several instructions simultaneously, improving throughput
- Must add logic to handle dependencies between simultaneously executing instructions
- Require non-architectural pipeline registers.
- Must access instructions and data in the same cycle,
 - Use separate instruction and data caches for this purpose
- All commercial high-performance processors use pipeline

Processor Performance

- **Program execution time**

Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)

- **Definitions:**

- CPI: Cycles/instruction
- clock period: seconds/cycle
- IPC: instructions/cycle = $1/CPI$

- **Challenge is to satisfy constraints of:**

- Cost
- Power
- Performance

RISC-V Processor

- Consider **subset** of RISC-V instructions:
 - R-type ALU instructions:
 - **add, sub, and, or, slt**
 - Memory instructions:
 - **lw, sw**
 - Branch instructions:
 - **beq**

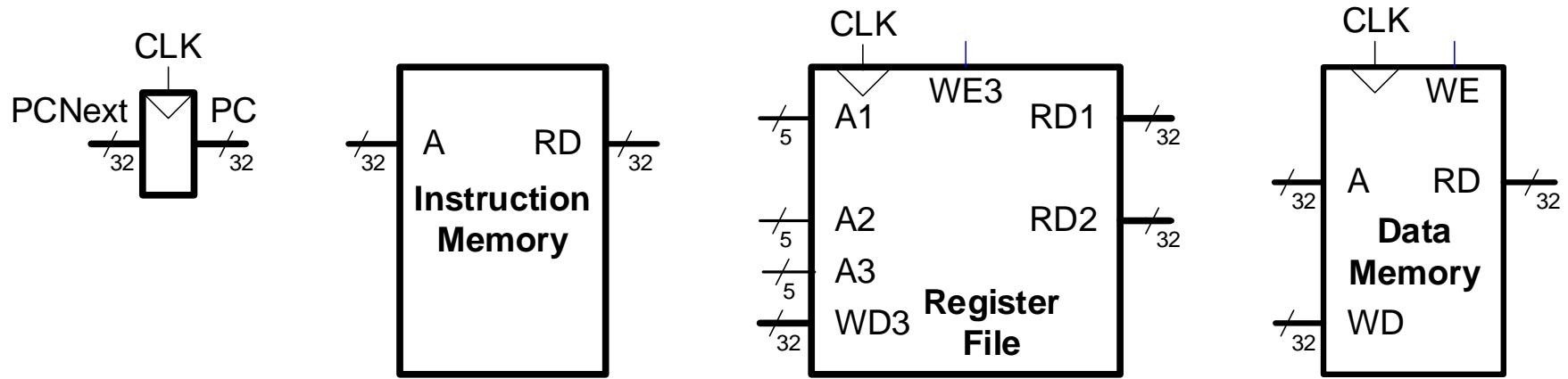
Architectural State Elements

Determines everything about a processor:

- **Architectural state:**

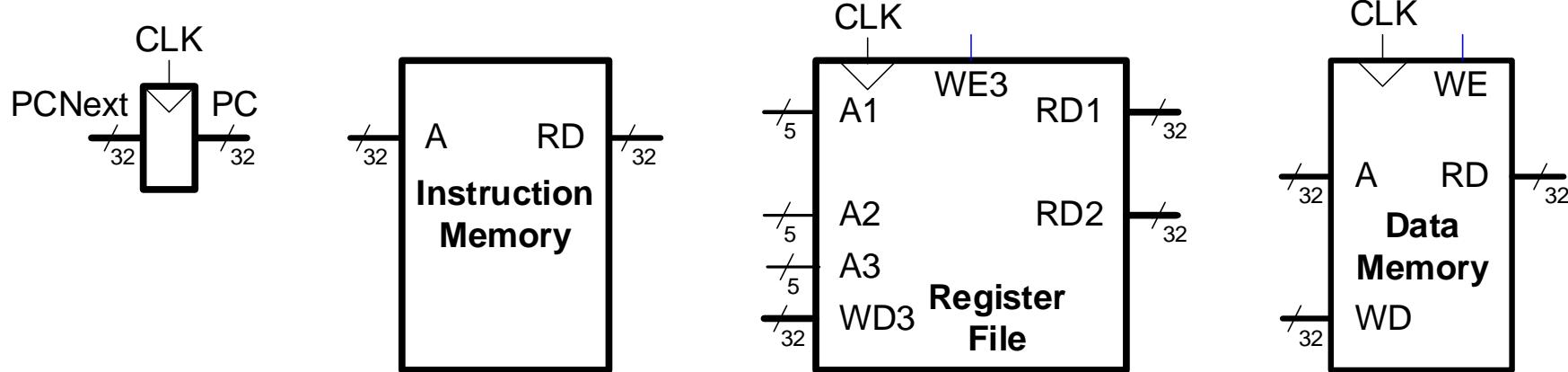
- 32 registers
- PC
- Memory

RISC-V Architectural State Elements



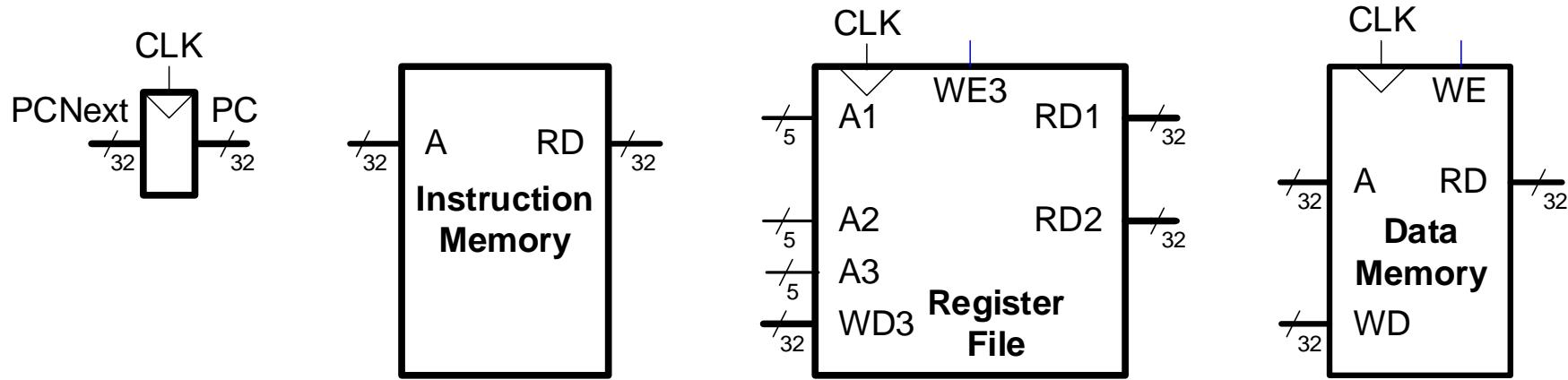
RISC-V Architectural State Elements

- Four state elements: 1. program counter, 2. registers file, 3. instruction memory, 4. data memory



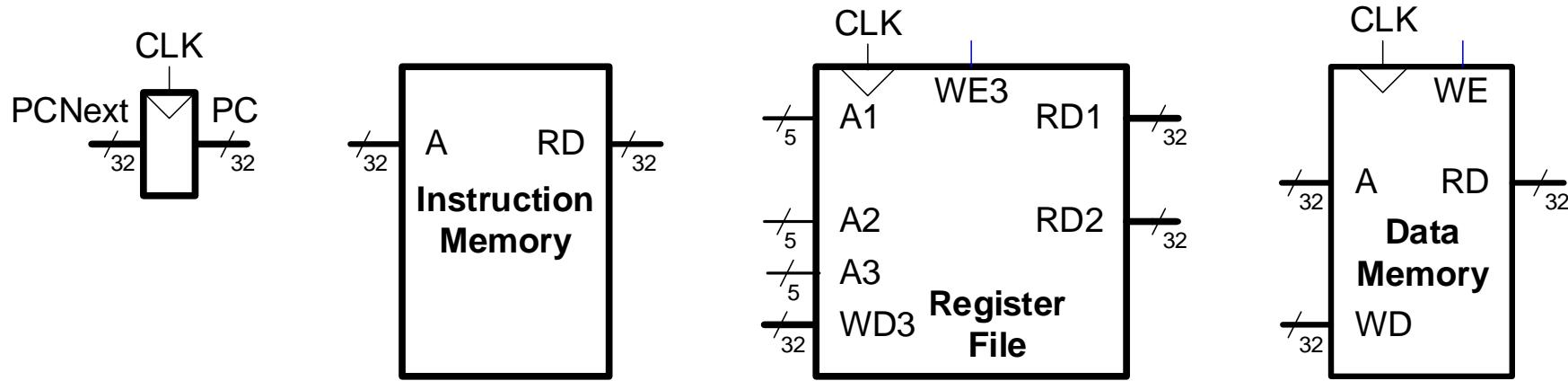
- Heavy lines indicate 32-bit data buses
- Medium lines indicate narrower buses, e.g. 5-bit address buses on the register file.
- Narrow lines indicate 1-bit wires
- Blue lines for control signals

RISC-V Architectural State Elements



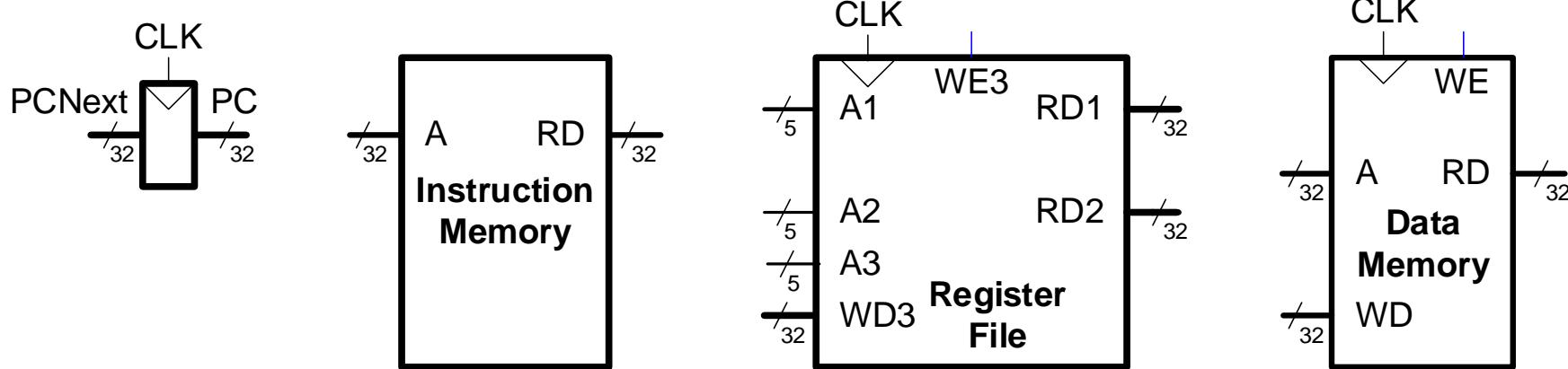
- **PC (program counter)** points to the current instruction.
- PCNext indicates the address of the next instruction.
- Registers usually have a reset input to put them into a known state at start-up, but reset is not shown to reduce clutter.

RISC-V Architectural State Elements



- **The Instruction memory** has a single read port
- It takes a 32-bit instruction address input, A
- It reads the 32-bit data (i.e. instruction) from that address onto the read data output, RD.

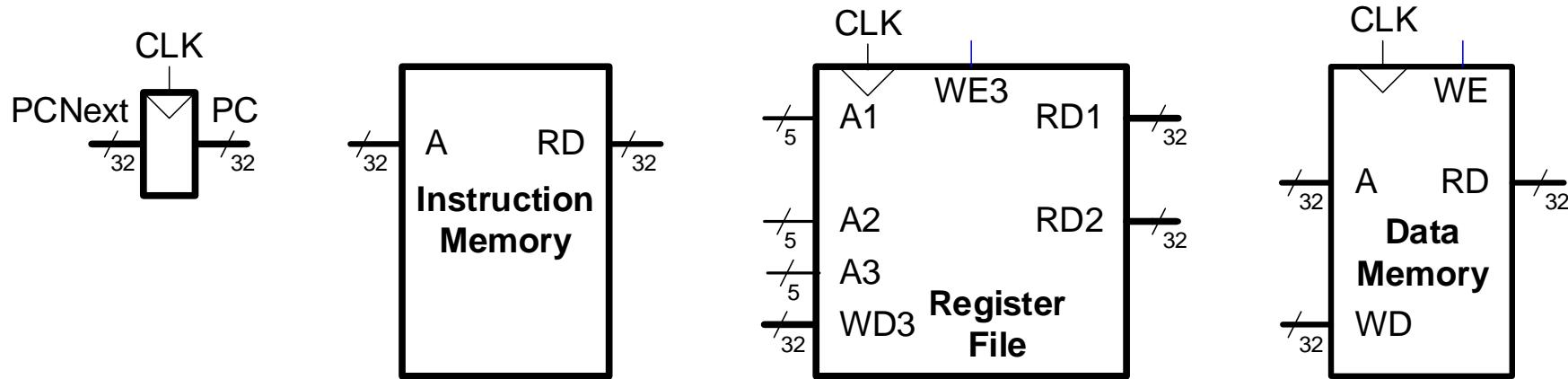
RISC-V Architectural State Elements



Port Name	Description	Width
A1	Read address 1	5
A2	Read address 2	5
RD1	Read data output 1	32
RD2	Read data output 2	32
A3	Read address 3	5
WD3	Write data input	32
WE3	Write enable input	1

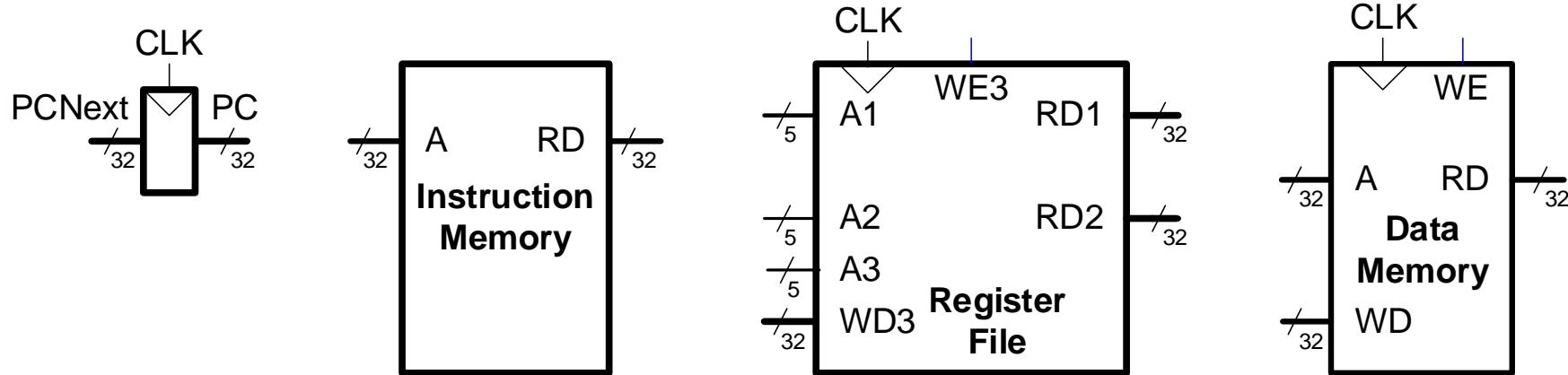
- The 32-element x 32-bit **register file** holds registers $\times 0 - \times 31$.
- $\times 0$ is hardwired to 0.
- A1, A2, and A3 are 5-bit address inputs, each specifying one of the $2^5 = 32$ registers.
- When WE3 is asserted, WD3 writes to the register specified by A3 on the clock-rising edge.
- RD1, RD2, and WD3 are 32-bit data

RISC-V Architectural State Elements



- **The Data memory** has a single read/write port
- If its write-enable input, **WE**, is asserted, then it writes data **WD** into address **A** on the clock rising edge.
- If its write enable is 0, it reads from address **A** onto the read data bus, **RD**.

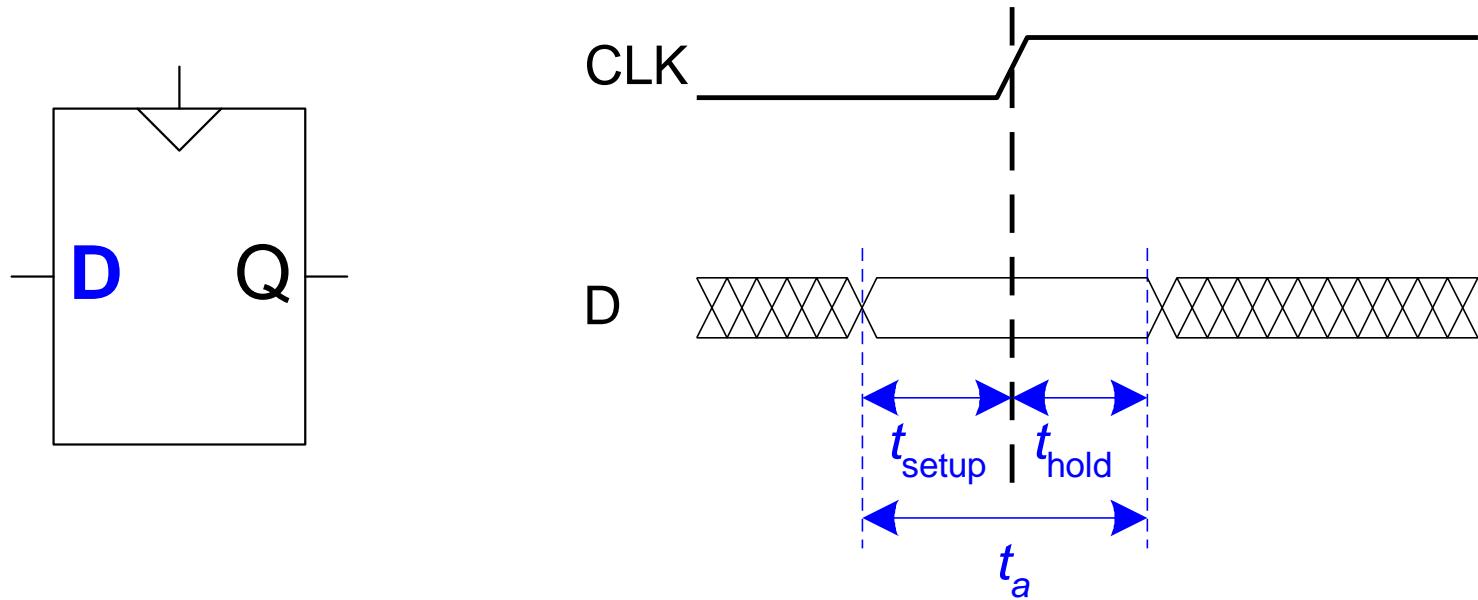
RISC-V Architectural State Elements



- The **Instruction memory**, **Register file**, and **Data memory** are all read *combinationally*.
 - if the address changes, the new data appears at RD after some propagation delay; no clock is involved.
- Write to Register file and Data memory are only on clock rising edge.
 - The system state changes only at the clock rising edge
 - The address, data, and write enable must *set up* before the clock edge and must remain stable until a *hold* time after the clock edge.

Input Timing Constraints

- **Setup time:** t_{setup} = time *before* clock edge data must be stable (i.e. not changing)
- **Hold time:** t_{hold} = time *after* clock edge data must be stable
- **Aperture time:** t_a = time *around* clock edge data must be stable ($t_a = t_{\text{setup}} + t_{\text{hold}}$)



Chapter 7: Microarchitecture

Single-Cycle RISC-V Processor

Single-Cycle RISC-V Processor

- **Datapath**
 - State elements and combinational logic that can execute the various instructions
- **Control**
 - Contains combinational logic that generates the appropriate control signals based on the current instruction.
 - Control signals determine which specific instruction the data path performs at any given time.

Example Program

- Design datapath
 - By connecting the state elements with combinational logic
- View the example program executing

Example Program:

Address	Instruction	Type	Fields						Machine Language	
0x1000	L7: lw x6, -4 (x9)	I	imm_{11:0} 111111111100	rs1 01001	f3 010	rd 00110	op 0000011		FFC4A303	
0x1004	sw x6, 8 (x9)	S	imm_{11:5} 0000000	rs2 00110	rs1 01001	f3 010	imm_{4:0} 01000	op 0100011		0064A423
0x1008	or x4, x5, x6	R	funct7 0000000	rs2 00110	rs1 00101	f3 110	rd 00100	op 0110011		0062E233
0x100C	beq x4, x4, L7	B	imm_{12,10:5} 1111111	rs2 00100	rs1 00100	f3 000	imm_{4:1,11} 10101	op 1100011		FE420AE3

Example Program

Example Program:

Address	Instruction	Type	Fields					Machine Language	
0x1000	l7: lw x6, -4(x9)	I	imm _{11:0} 111111111100	rs1 01001	f3 010	rd 00110	op 0000011	FFC4A303	
0x1004	sw x6, 8(x9)	S	imm _{11:5} 0000000	rs2 00110	rs1 01001	f3 010	imm _{4:0} 01000	op 0100011	0064A423
0x1008	or x4, x5, x6	R	funct7 0000000	rs2 00110	rs1 00101	f3 110	rd 00100	op 0110011	0062E233
0x100C	beq x4, x4, l7	B	imm _{12,10:5} 1111111	rs2 00100	rs1 00100	f3 000	imm _{4:1,11} 10101	op 1100011	FE420AE3

- Suppose the program is stored in memory starting at address 0x1000
- The above figure indicates
 - the address of each instruction,
 - the instruction type,
 - the instruction fields, and
 - the hexadecimal machine language code for the instruction.

Example Program

Example Program:

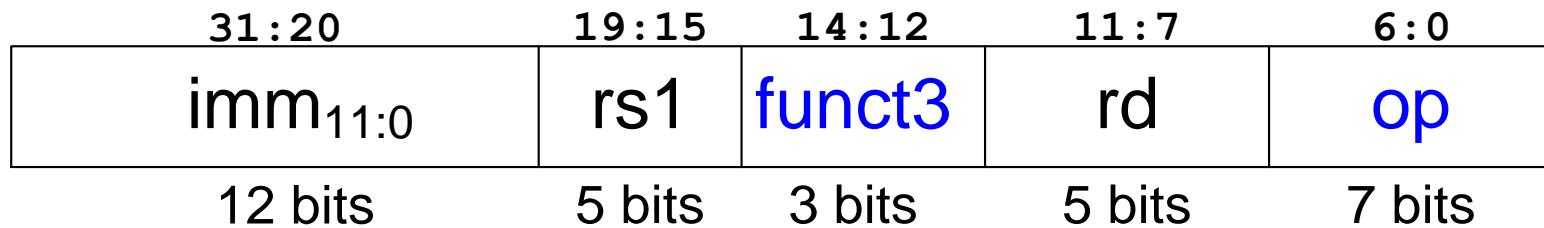
Address	Instruction	Type	Fields						Machine Language	
0x1000	l7: lw x6, -4(x9)	I	imm _{11:0} 111111111100	rs1 01001	f3 010	rd 00110	op 0000011	FFC4A303		
0x1004	sw x6, 8(x9)	S	imm _{11:5} 0000000	rs2 00110	rs1 01001	f3 010	imm _{4:0} 01000	op 0100011	0064A423	
0x1008	or x4, x5, x6	R	funct7 0000000	rs2 00110	rs1 00101	f3 110	rd 00100	op 0110011	0062E233	
0x100C	beq x4, x4, l7	B	imm _{12,10:5} 1111111	rs2 00100	rs1 00100	f3 000	imm _{4:1,11} 10101	op 1100011	FE420AE3	

- Assume initially x5 contains value 6; x9 contains 0x2004, and the memory location 0x2000 contains the value 10
- The program counter begins at 0x1000.
- The lw reads 10 from address (0x2004-4)= 0x2000 and puts it in x6.
- The sw writes 10 to address (0x2004+8) = 0x200C.
- The or computes x4 = 6 | 10 = 0110₂ | 1010₂ = 1110₂ = 14
- Then, beq goes back to label l7, so the program repeats forever.

Single-Cycle RISC-V Processor

- **Datapath:** start with `lw` instruction
- **Example:** `lw x6, -4(x9)`
`lw rd, imm(rs1)`

I-Type



Instruction	op	funct3	Funct7	Type
<code>lw</code>	0000011 (3)	010 (2)	-	I-Type

Single-Cycle RISC-V Processor

- **Datapath:** start with `lw` instruction
- **Example:** `lw x6, -4(x9)`
`lw rd, imm(rs1)`

I-Type

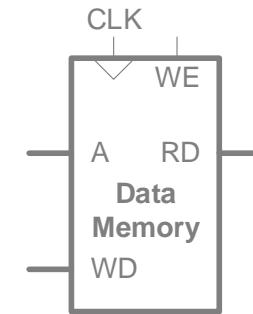
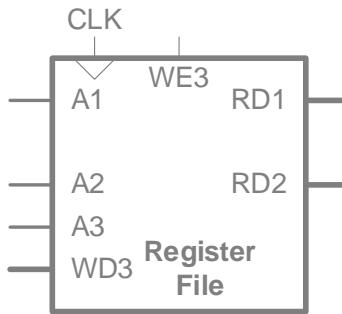
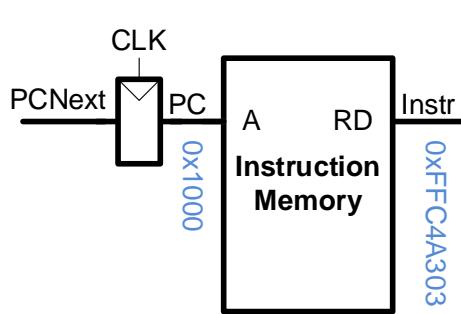
31:20	19:15	14:12	11:7	6:0
$\text{imm}_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

imm_{11:0} **rs1** **f3** **rd** **op**
111111111100 01001 010 00110 0000011

Instruction	op	funct3	Funct7	Type
<u>lw</u>	0000011 (3)	010 (2)	-	I-Type

Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction

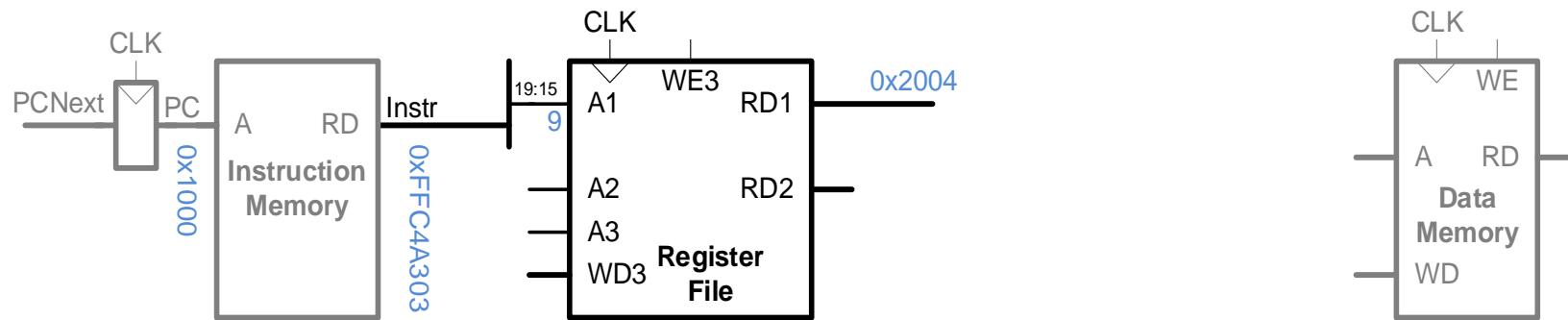


Address	Instruction	Type	Fields	Machine Language
0x1000	<code>l7: lw x6, -4 (x9)</code>	I	$\text{imm}_{11:0}$ 111111111100 rs1 01001 f3 010 rd 00110 op 0000011	FFC4A303

- The program counter contains the address of the instruction to execute.
- The first step is to read (*fetch*) this instruction from the instruction memory.
 - PC is connected to the address input of the instruction memory.
- The instruction memory reads out (*fetches*) the 32-bit instruction, labeled *Instr*.
- The PC is 0x1000
 - PC is really 0x00001000, leading zeros omitted to avoid cluttering
- We will work out the datapath connections for the `lw` instruction.
- We will consider generalising the datapath to handle other instructions.

Single-Cycle Datapath: lw Reg Read

STEP 2: Read source operand (rs1) from RF

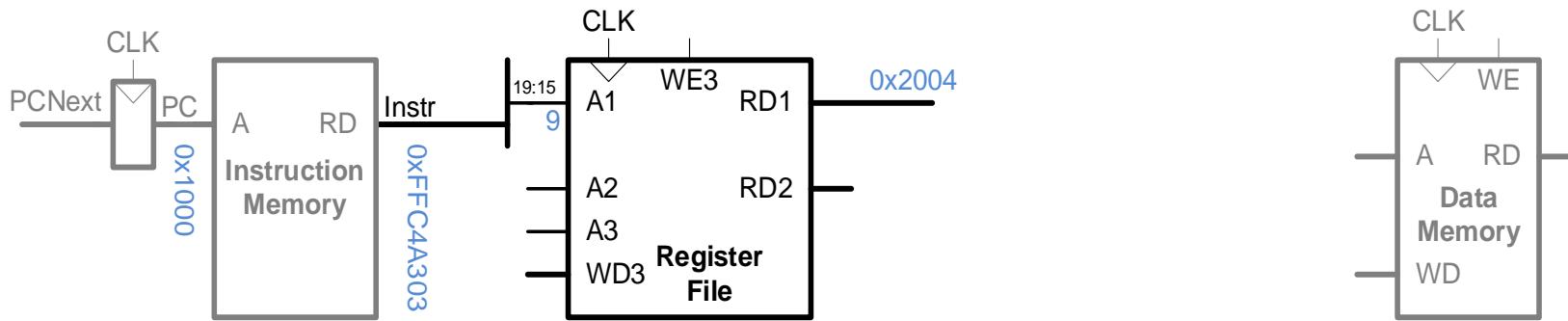


Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4 (x9)	I	$\text{imm}_{11:0}$: 111111111100 rs1 : 01001 f3 : 010 rd : 00110 op : 0000011	FFC4A303

- The *Instr* is **lw**, 0xFFC4A303, as also shown at the bottom of the figure.
- These sample values are annotated in light blue on the diagram.

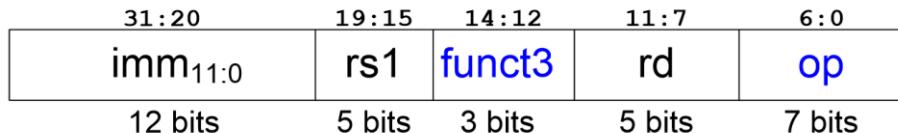
Single-Cycle Datapath: lw Reg Read

STEP 2: Read source operand (rs1) from RF



Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4 (x9)	I	$\text{imm}_{11:0}$ rs1 f3 rd op	01001 010 00110 0000011 FFC4A303

I-Type

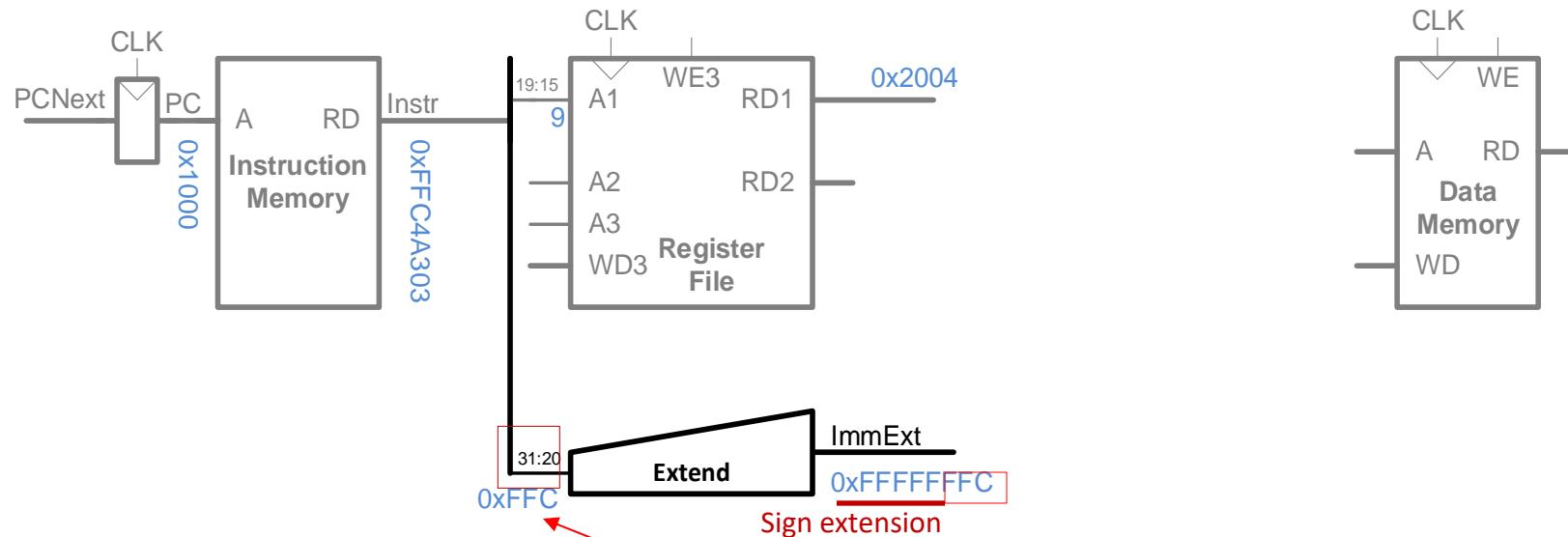


imm_{11:0} 111111111100 rs1 01001 f3 010 rd 00110 op 0000011

For the lw instruction, the next step is to read the source register containing the base address. Recall that lw is an I-type instruction, and the base register is specified in the rs1 field of the instruction, Instr_{19:15}. These bits of the instruction connect to the A1 address input of the register file, as shown in Figure 7.4. The register file reads the register value onto RD1. In our example, the register file reads 0x2004 from x9.

Single-Cycle Datapath: lw Immediate

STEP 3: Extend the immediate

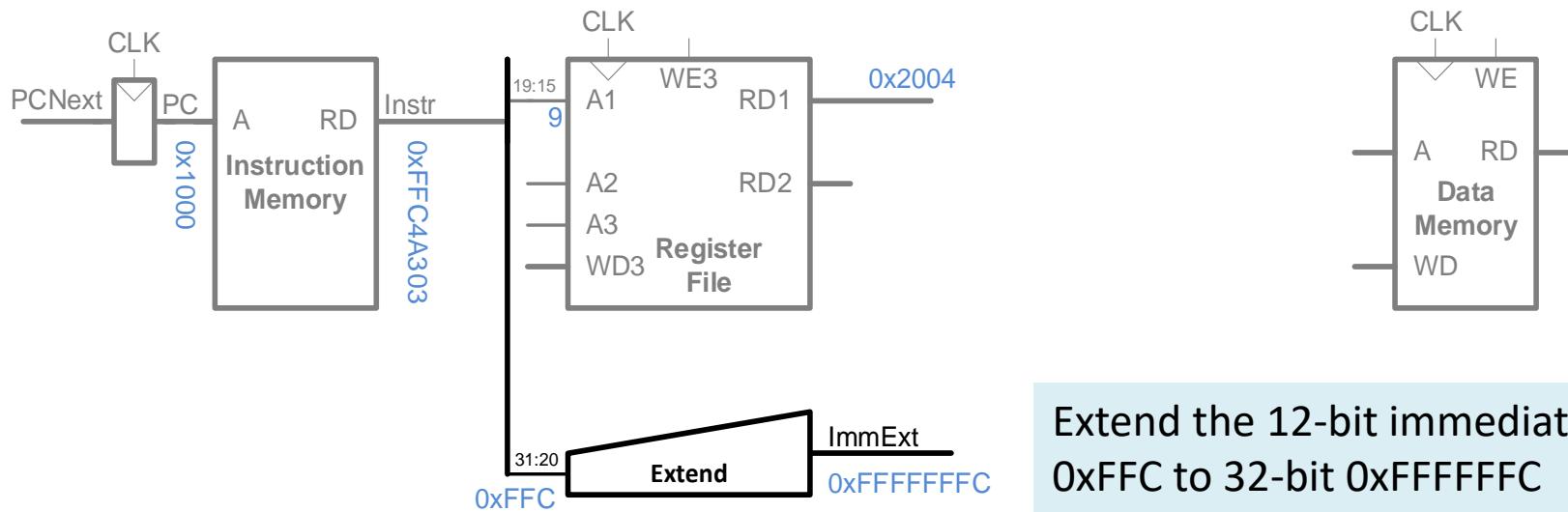


Address	Instruction	Type	Fields	Machine Language
0x1000	<code>l7: lw x6, -4 (x9)</code>	I	<code>imm_{11:0}</code> 0xFFC rs1 01001 f3 010 rd 00110 op 0000011	FFC4A303

- The offset is stored in the 12-bit signed immediate field of the instruction, $\text{Instr}_{31:20}$.
- The 12-bit signed immediate is extended to 32-bit signed immediate.

Single-Cycle Datapath: lw Immediate

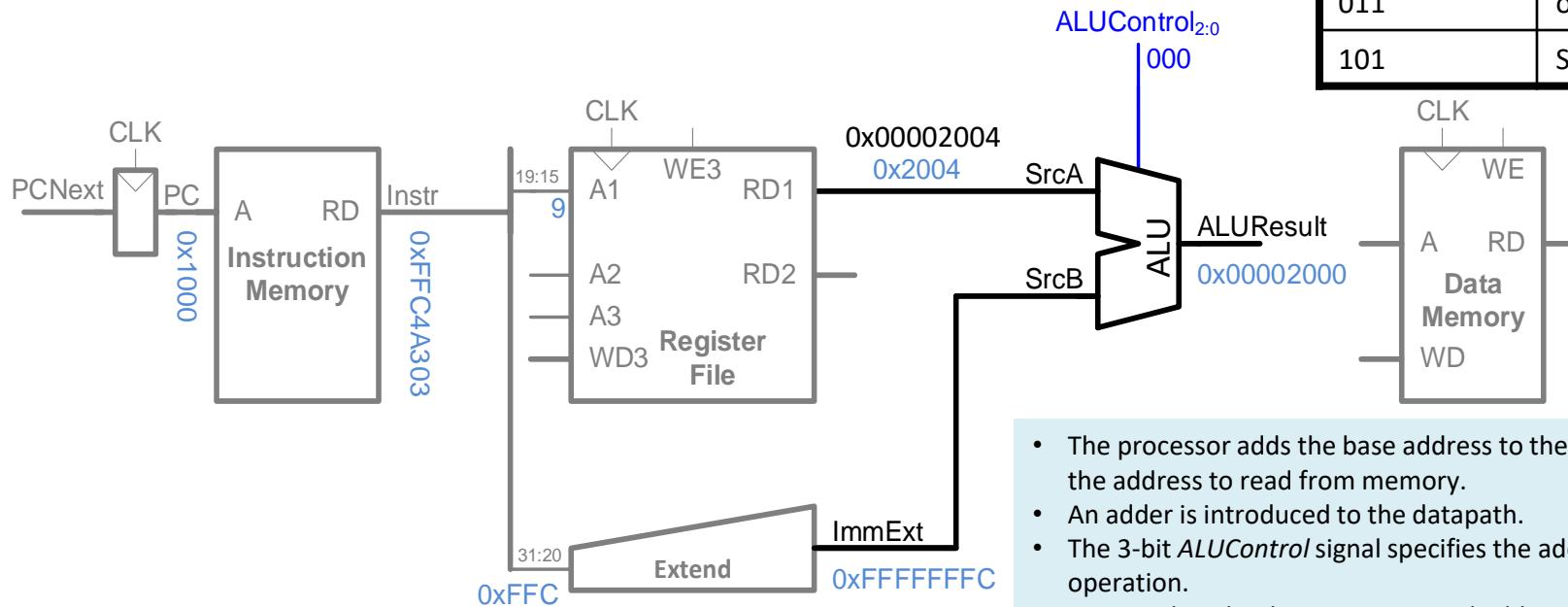
STEP 3: Extend the immediate



Address	Instruction	Type	Fields	Machine Language
0x1000	I7: lw x6, -4 (x9)	I	imm _{11:0} rs1 f3 rd op	1111111111100 01001 010 00110 0000011 FFC4A303

Single-Cycle Datapath: lw Address

STEP 4: Compute the memory address

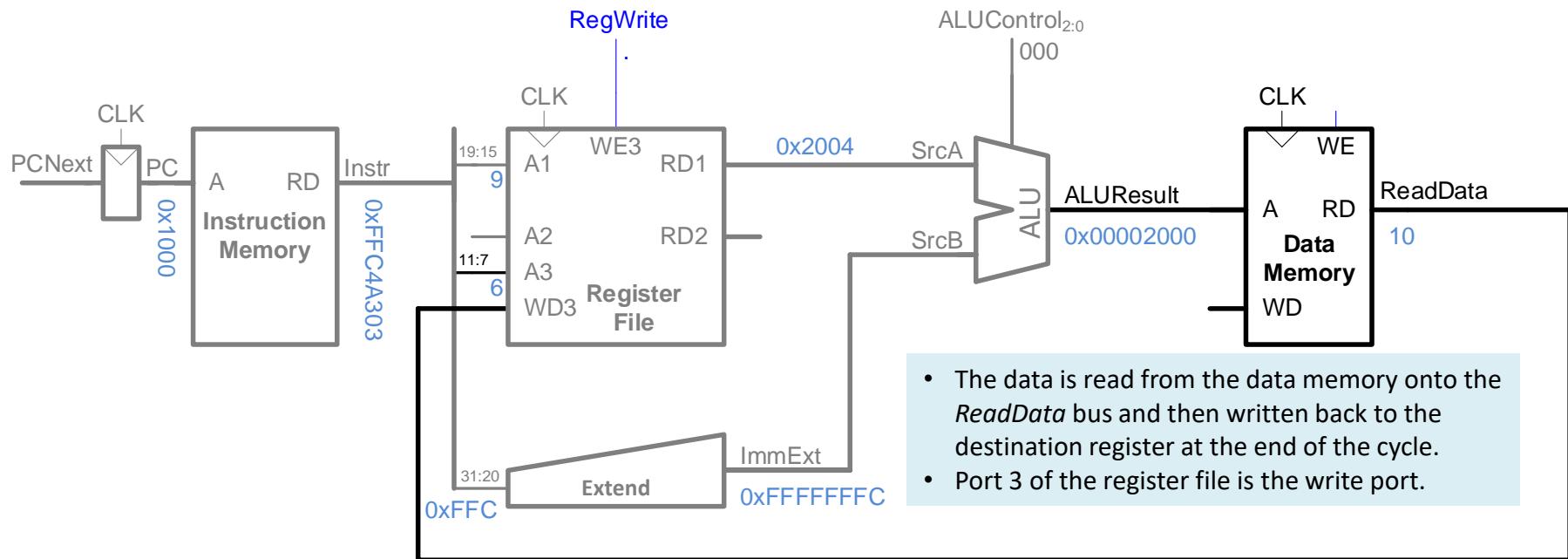


- The processor adds the base address to the offset to find the address to read from memory.
- An adder is introduced to the datapath.
- The 3-bit `ALUControl` signal specifies the addition operation.
- `ALUResult` is the data memory read address.

Address	Instruction	Type	Fields	Machine Language
0x1000	I7: lw x6, -4(x9)	I	$\text{imm}_{11:0}$ rs1 f3 rd op	111111111100 01001 010 00110 0000011 FFC4A303

Single-Cycle Datapath: lw Mem Read

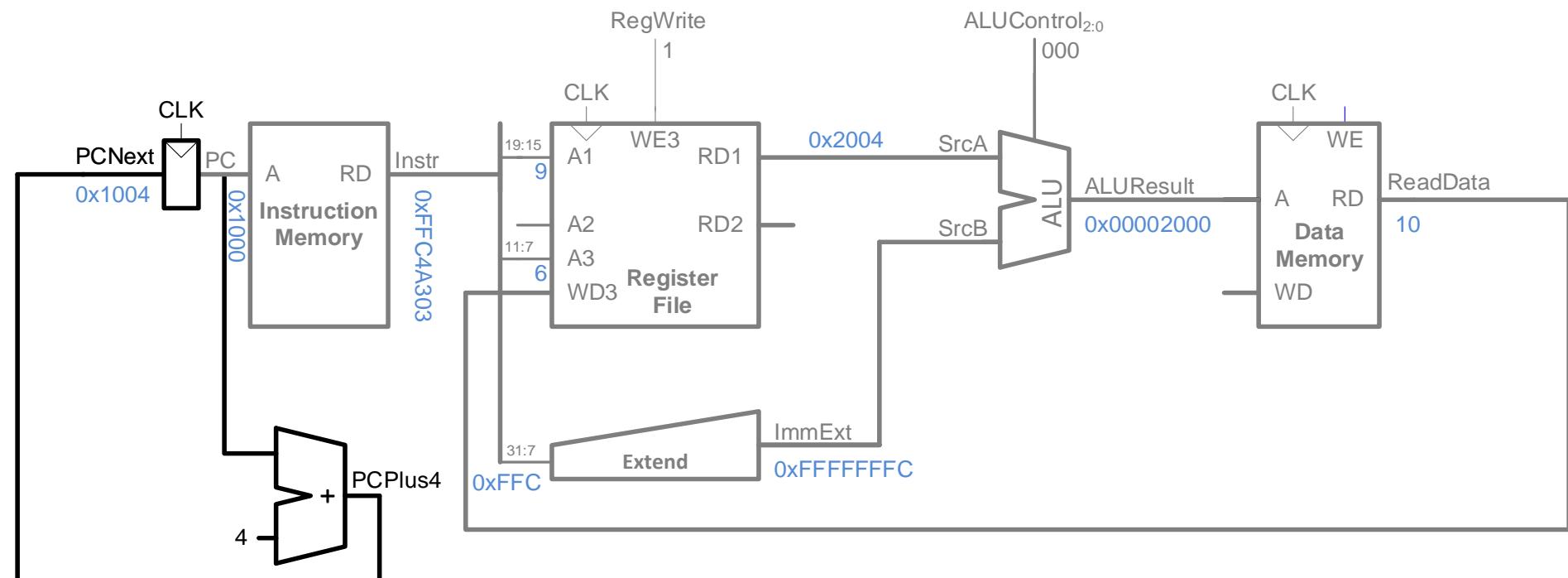
STEP 5: Read data from memory and write it back to register file



Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4 (x9)	I	imm _{11:0} : 111111111100 rs1: 01001 f3: 010 rd: 00110 op: 0000011	FFC4A303

Single-Cycle Datapath: PC Increment

STEP 6: Determine address of next instruction



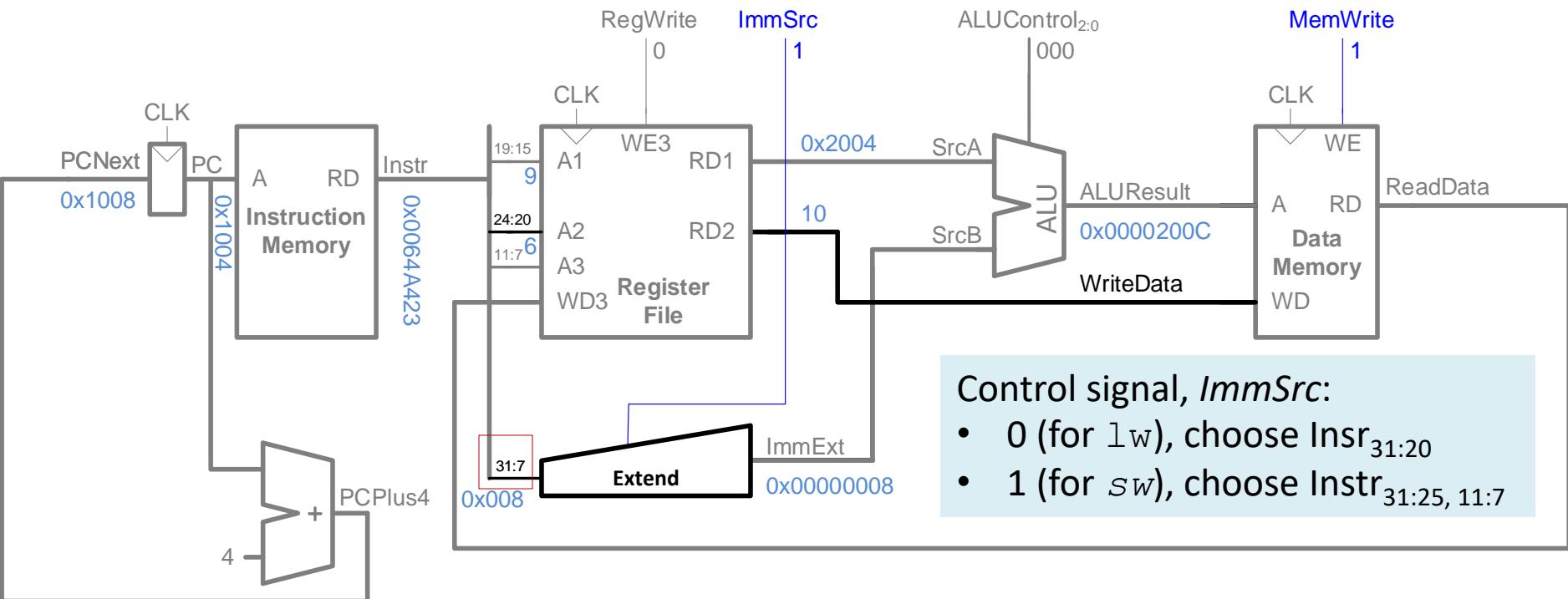
Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4(x9)	I	imm _{11:0} : 111111111100 rs1: 01001 f3: 010 rd: 00110 op: 0000011	FFC4A303

Chapter 7: Microarchitecture

Single-Cycle Datapath: Other Instructions

Single-Cycle Datapath: sw

- **Immediate:** now in $\{\text{instr}[31:25], \text{instr}[11:7]\}$
- **Add control signals:** ImmSrc, MemWrite

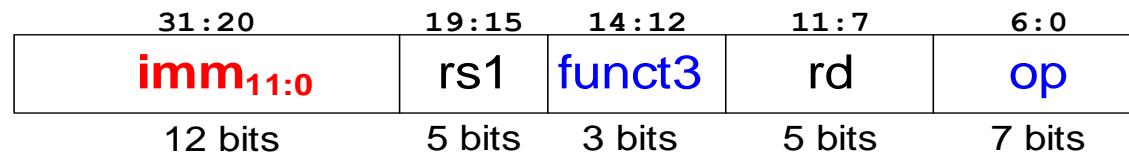


Address	Instruction	Type	Fields							Machine Language	
0x1004	sw x6, 8 (x9)	S	$\text{imm}_{11:5}$ 0000000	rs2 00110	rs1 01001	f3 010	$\text{imm}_{4:0}$ 01000	op 0100011		0064A423	

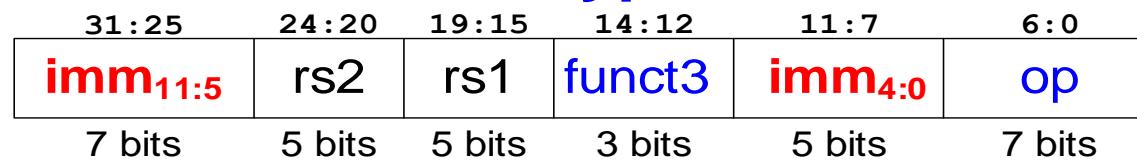
Single-Cycle Datapath: Immediate

ImmSrc	ImmExt	Instruction Type
0	$\{\{20\{\text{instr}[31]\}\}, \text{instr}[31:20]\}$	I-Type
1	$\{\{20\{\text{instr}[31]\}\}, \text{instr}[31:25], \text{instr}[11:7]\}$	S-Type

I-Type

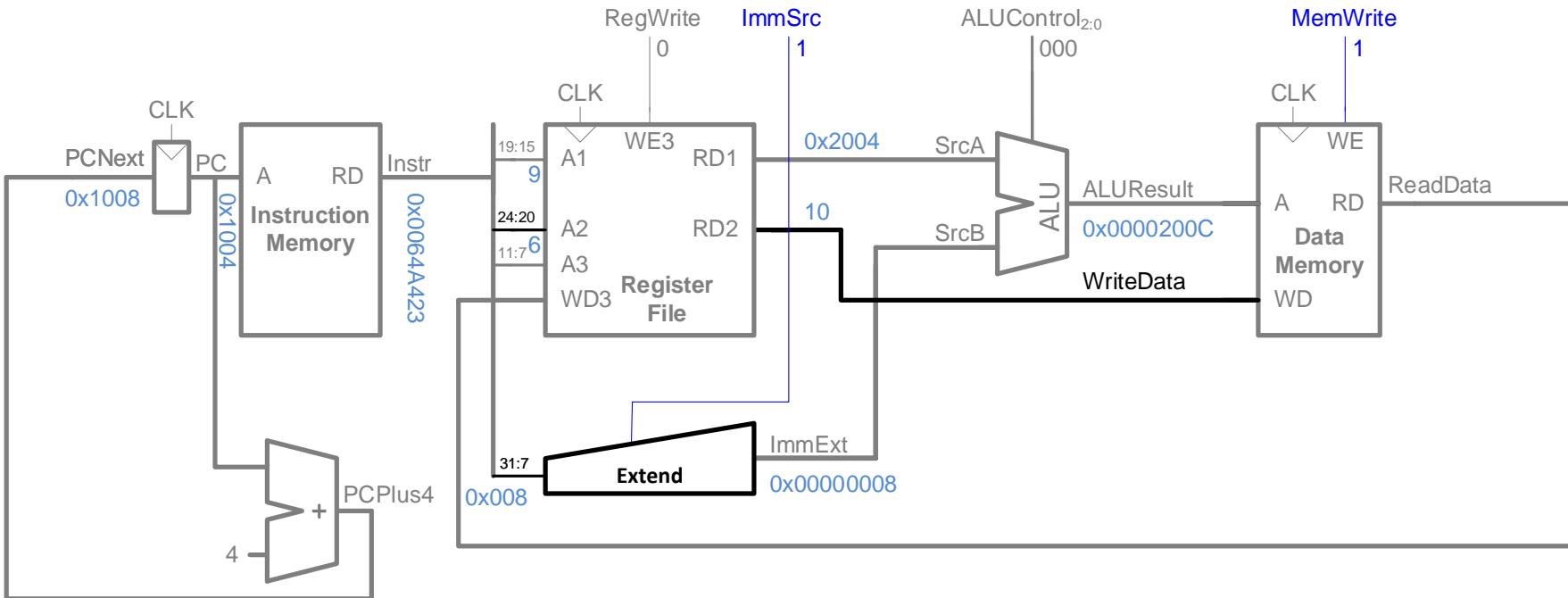


S-Type



funct7	4 3 2 1 0	rs1	funct3	rd	R
11 10 9 8 7 6 5	4 3 2 1 0	rs1	funct3	rd	I
11 10 9 8 7 6 5	rs2	rs1	funct3	4 3 2 1 0	S
12 10 9 8 7 6 5	rs2	rs1	funct3	4 3 2 1 11	B
31 30 29 28 27 26 25	24 23 22 21 20 19 18 17 16 15 14 13 12		rd		U
20 10 9 8 7 6 5 4 3 2 1 11	19 18 17 16 15 14 13 12		rd		J
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8 7				

Single-Cycle Datapath: sw

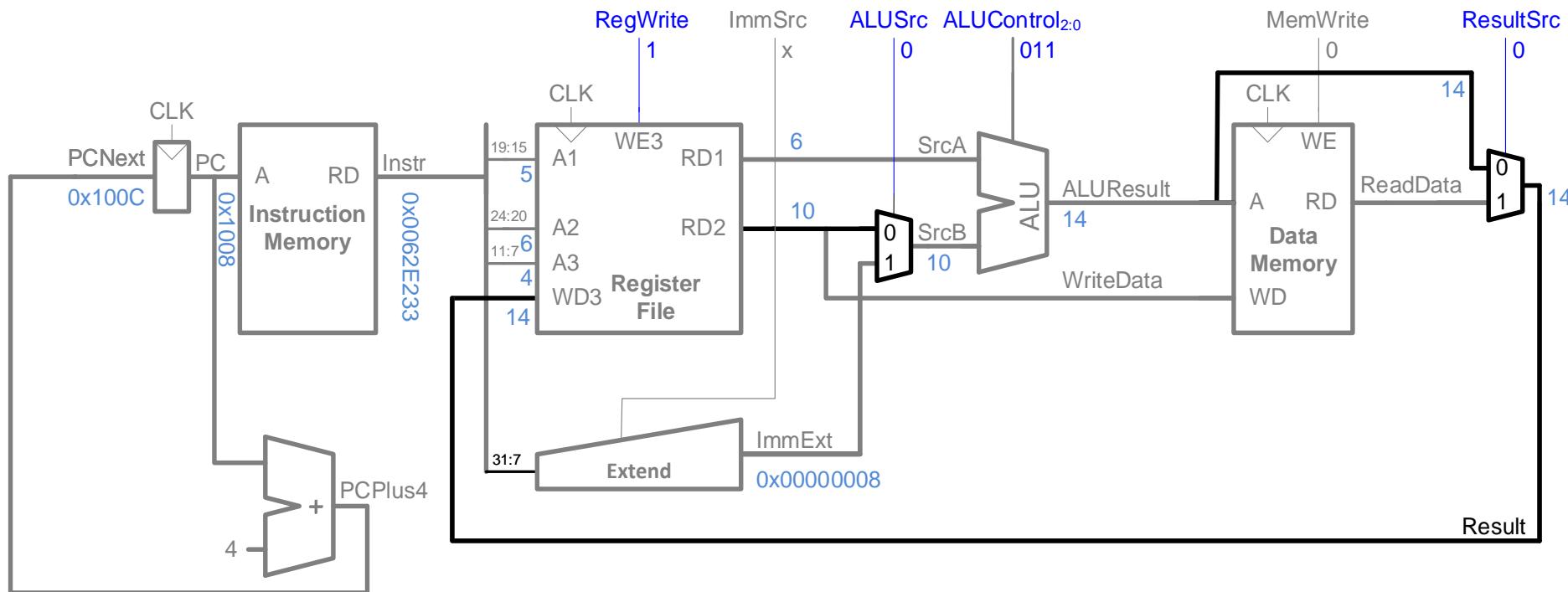


Address	Instruction	Type	Fields							Machine Language
0x1004	sw x6, 8 (x9)	S	<code>imm_{11:5}</code> 0000000	<code>rs2</code> 00110	<code>rs1</code> 01001	<code>f3</code> 010	<code>imm_{4:0}</code> 01000	<code>op</code> 0100011		0064A423

In our example, the PC is `0x1004`. Thus, the instruction memory reads out the `sw` instruction, `0x0064A423`. The register file reads `0x2004` (the base address) from `x9` and `10` from `x6` while the Extend unit extends the immediate offset `8` from 12 to 32 bits. The ALU computes $0x2004 + 8 = 0x200C$. The data memory writes `10` to address `0x200C`. Meanwhile, the PC is incremented to `0x1008`.

Single-Cycle Datapath: R-type

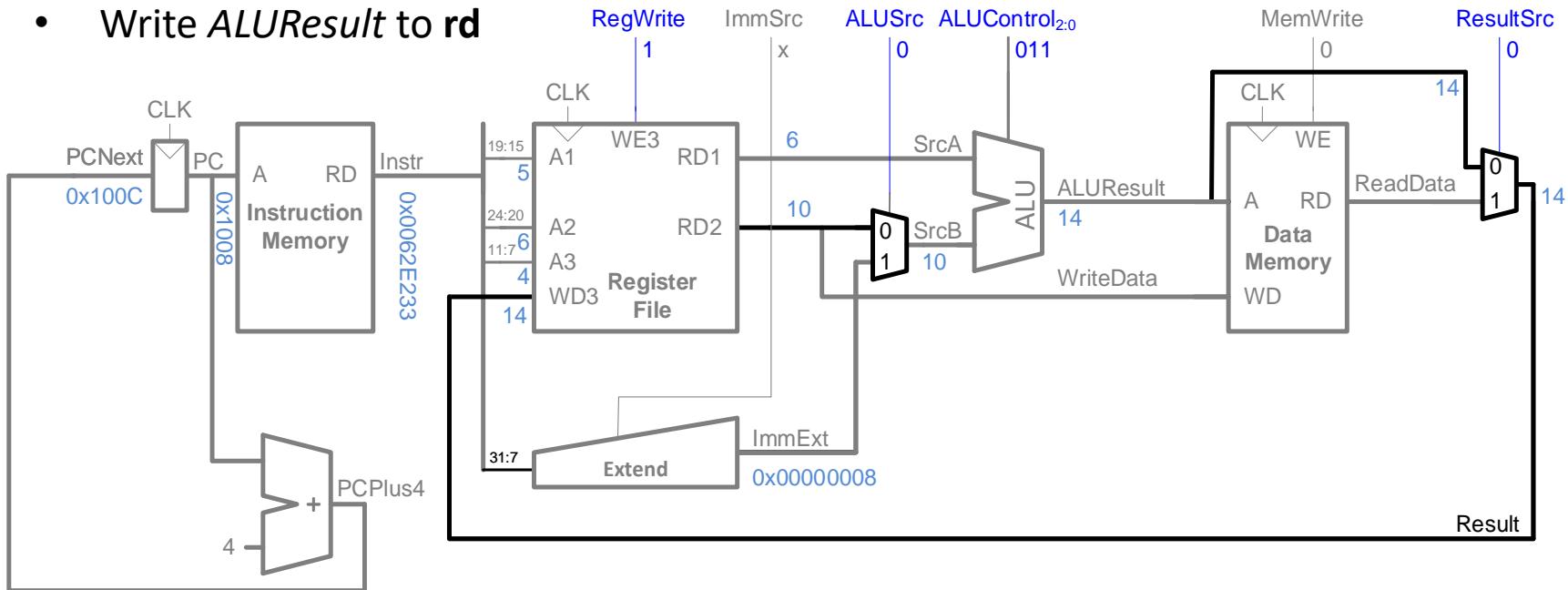
- Read from **rs1** and **rs2** (instead of **imm**)
- Write **ALUResult** to **rd**



Address	Instruction	Type	Fields	Machine Language
0x1008	or x4, x5, x6	R	funct7 0000000 rs2 00110 rs1 00101 f3 110 rd 00100 op 0110011	0062E233

Single-Cycle Datapath: R-type

- Read from **rs1** and **rs2** (instead of **imm**)
- Write **ALUResult** to **rd**

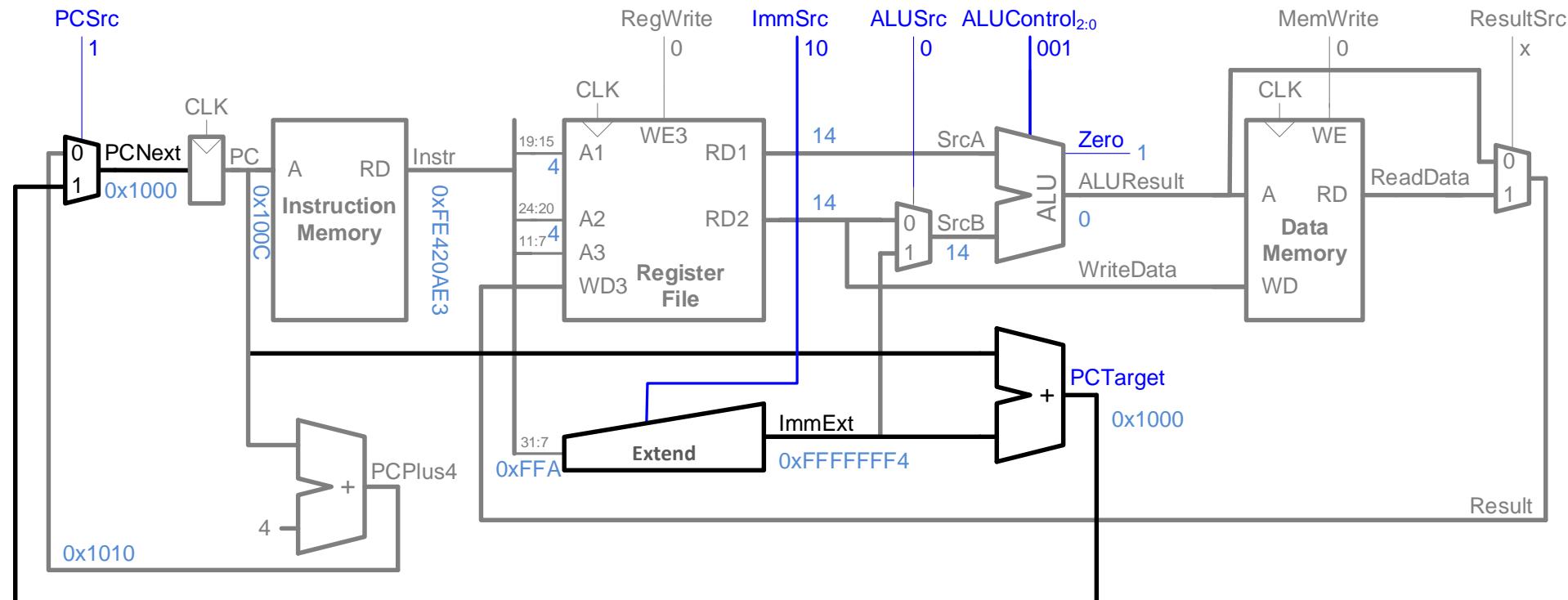


Address	Instruction	Type	Fields	Machine Language
0x1008	or x4, x5, x6	R	funct7: 0000000 rs2: 00110 rs1: 00101 f3: 110 rd: 00100 op: 0110011	0062E233

In our example, the PC is 0x1008. Thus, the instruction memory reads out the or instruction 0x0062E233. The register file reads source operands 6 from x5 and 10 from x6. ALUControl is 011, so the ALU computes $6 \mid 10 = 0110_2 \mid 1010_2 = 1110_2 = 14$. The result is written back to x4. Meanwhile, the PC is incremented to 0x100C.

Single-Cycle Datapath: beq

Calculate **target address**: PCTarget = PC + imm



Address	Instruction	Type	Fields								Machine Language	
0x100C	beq x4, x4, L7	B	imm _{12,10:5}	rs2	rs1	f3	imm _{4:1,11}	op	1100011	1100011	FE420AE3	

Single-Cycle Datapath: ImmExt

ImmSrc _{1:0}	ImmExt	Instruction Type
00	$\{{\text{20}\{\text{instr}[31]\}}, \text{instr}[31:20]\}$	I-Type
01	$\{{\text{20}\{\text{instr}[31]\}}, \text{instr}[31:25], \text{instr}[11:7]\}$	S-Type
10	$\{{\text{19}\{\text{instr}[31]\}}, \text{instr}[31], \text{instr}[7], \text{instr}[30:25], \text{instr}[11:8], 1'b0\}$	B-Type

I-Type

31:20	19:15	14:12	11:7	6:0
imm _{11:0}	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

S-Type

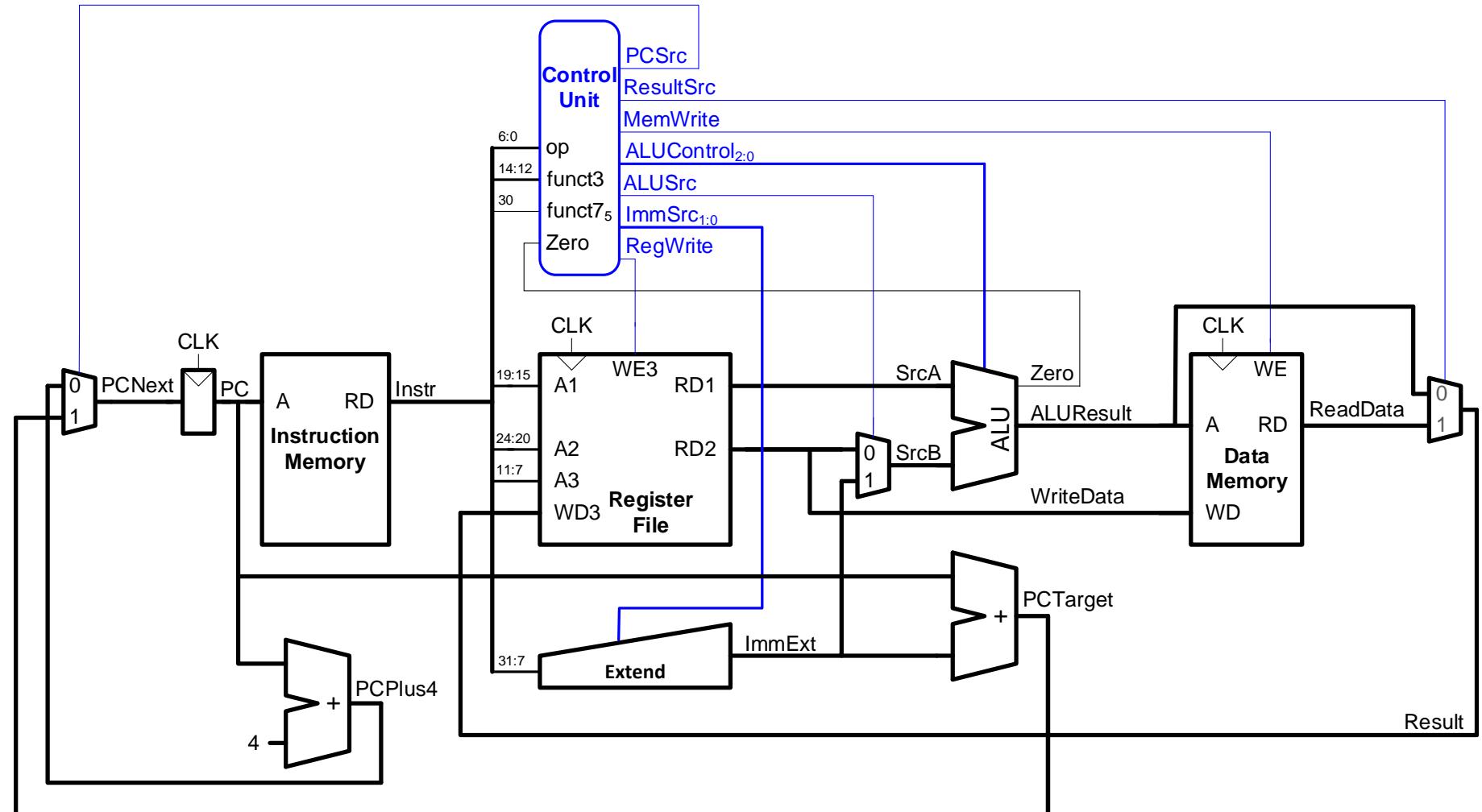
31:25	24:20	19:15	14:12	11:7	6:0
imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

funct7	4	3	2	1	0	rs1	funct3	rd
11 10 9 8 7 6 5	4	3	2	1	0	rs1	funct3	rd
11 10 9 8 7 6 5		rs2		rs1	funct3	4 3 2 1 0		
12 10 9 8 7 6 5		rs2		rs1	funct3	4 3 2 1 11		
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12								rd
20 10 9 8 7 6 5 4 3 2 1 11 19 18 17 16 15 14 13 12								rd
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7								

Single-Cycle RISC-V Processor

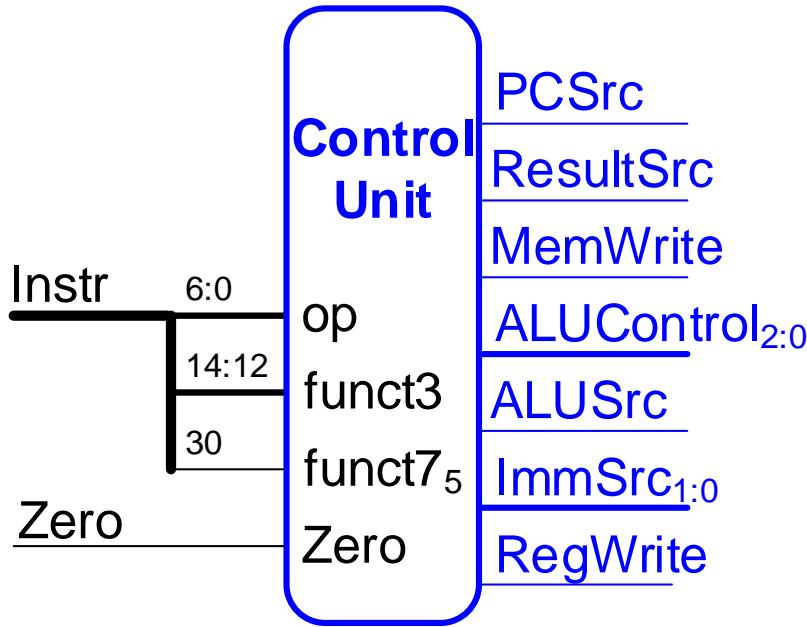


Chapter 7: Microarchitecture

Single-Cycle Control

Single-Cycle Control

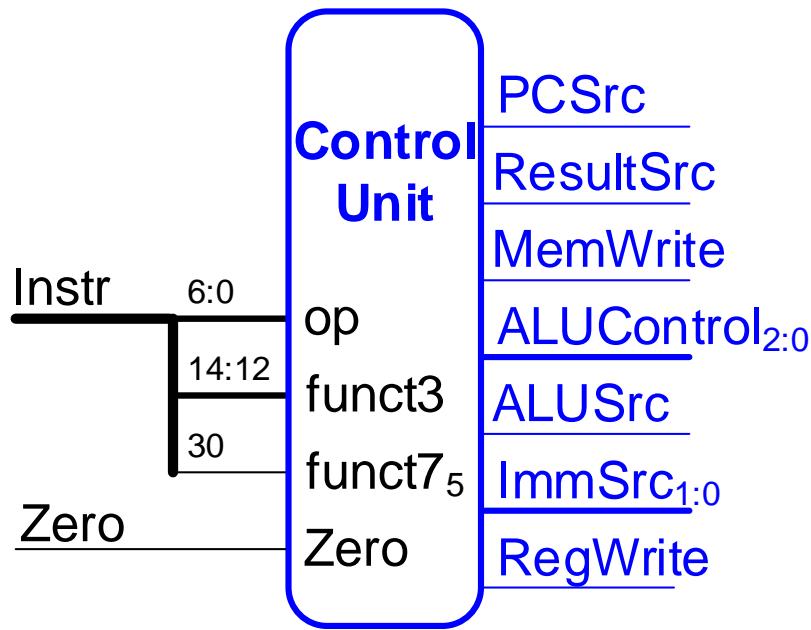
High-Level View



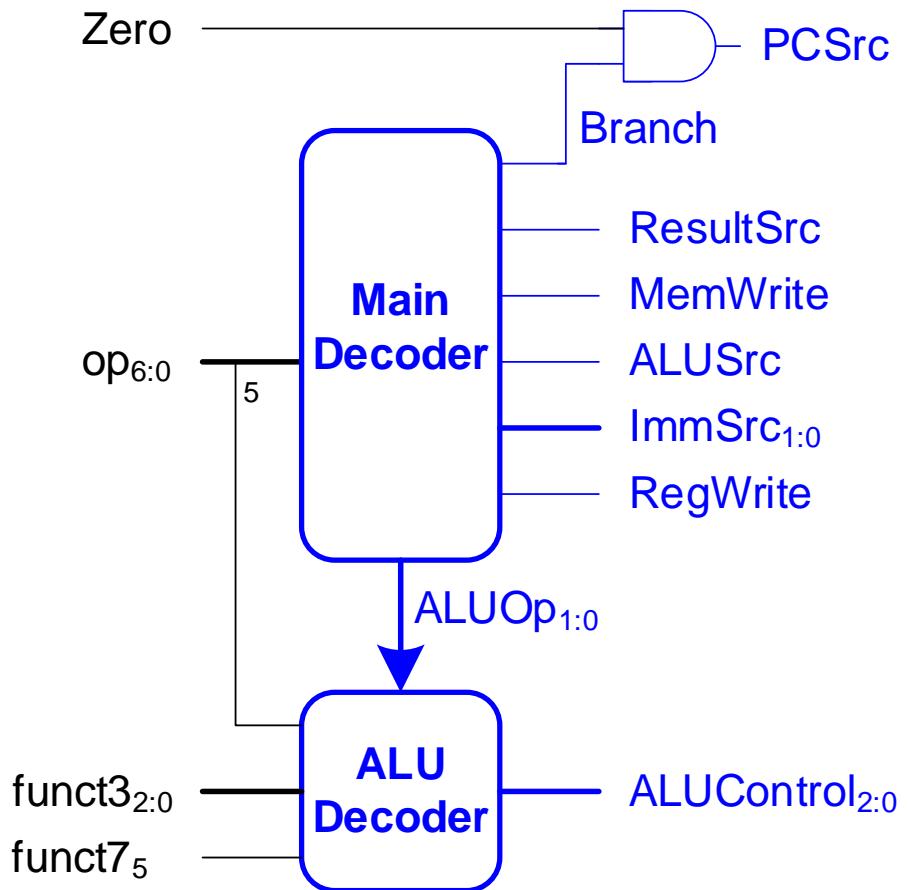
- The single-cycle processor's control unit computes the control signals based on **op**, **funct3**, and **funct7**.
- For the RV32I instruction set, only bit 5 of funct7 is used.
- So, we only need to consider
 - **Op** (Instr6:0)
 - **funct3** (Instr14:12)
 - **funct7₅** (Instr30)

Single-Cycle Control

High-Level View

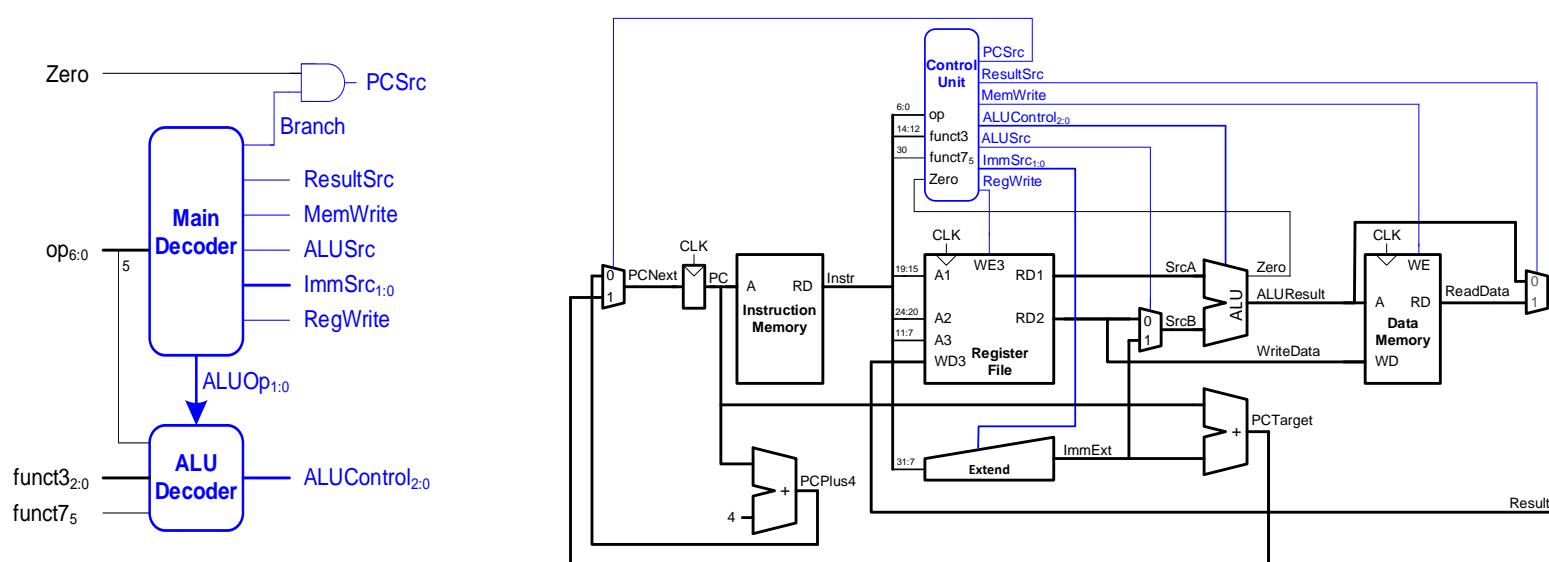


Low-Level View



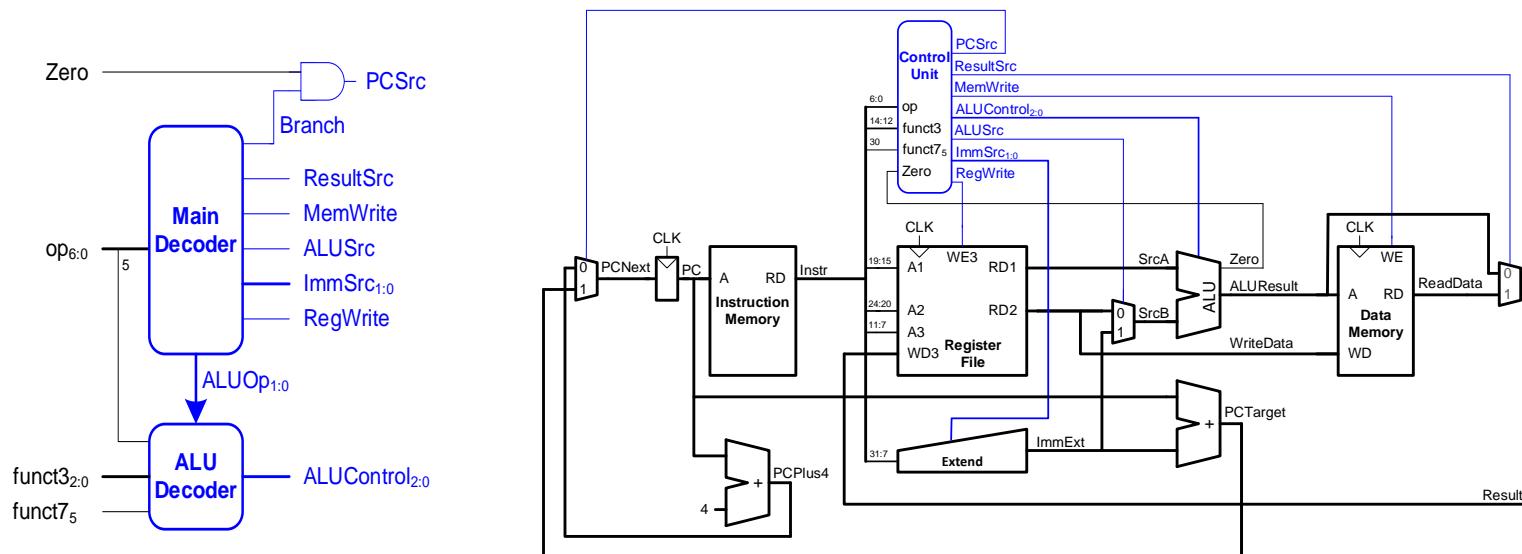
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw							
35	sw							
51	R-type							
99	beq							



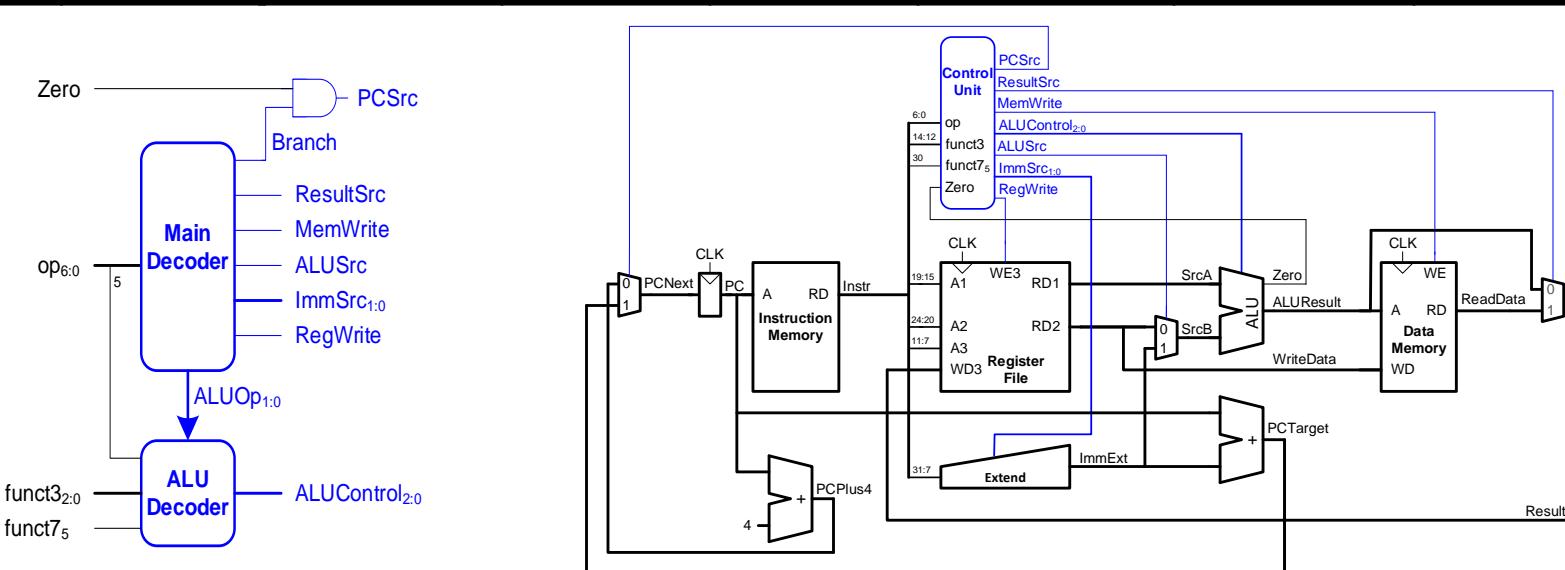
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1						
35	sw							
51	R-type							
99	beq							



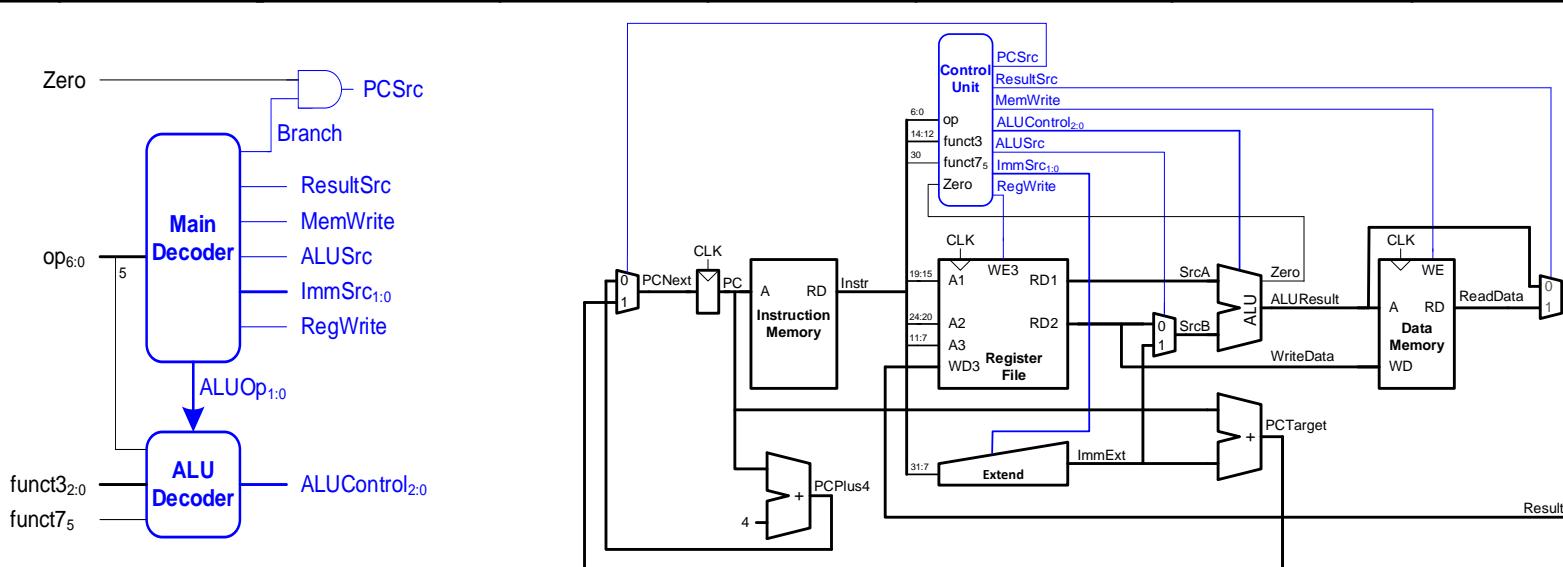
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00					
35	sw		ImmSrc_{1:0}	ImmExt		Instruction Type		
51	R-type		00	{}{20{instr[31]}}, instr[31:20]	I-Type			
51	R-type		01	{}{20{instr[31]}}, instr[31:25], instr[11:7]	S-Type			
99	beq		10	{}{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0	B-Type			



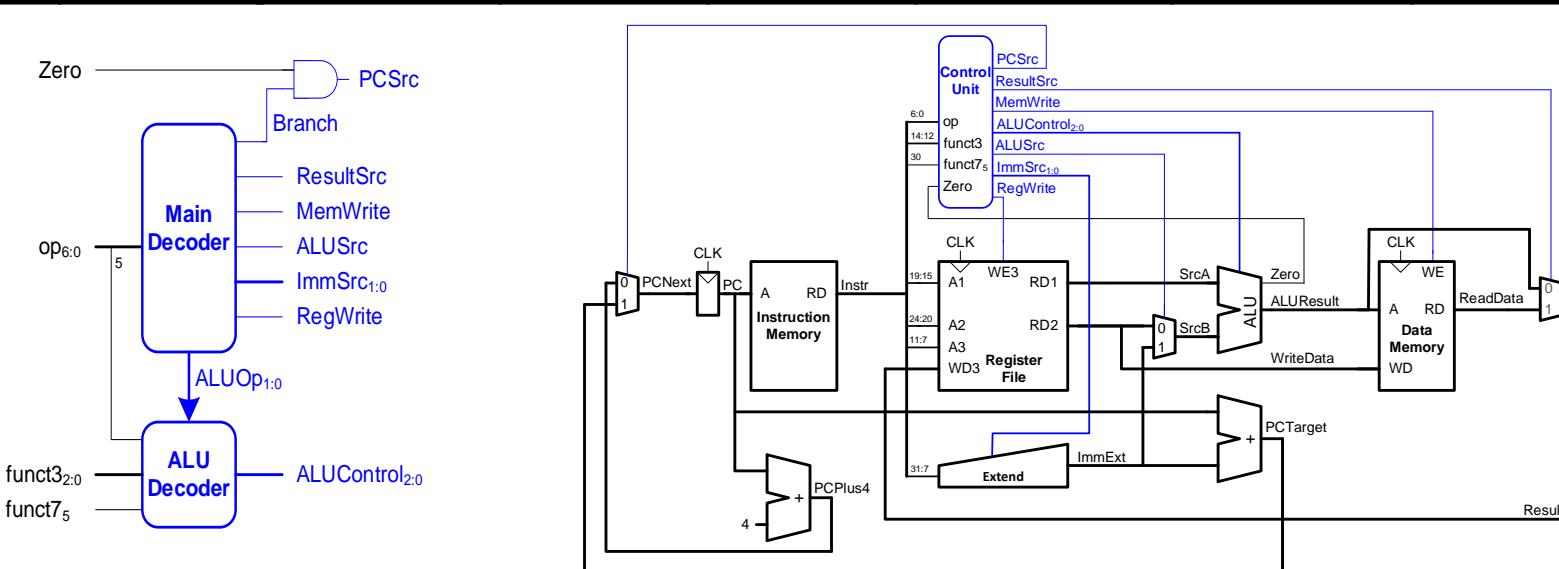
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1				
35	sw							
51	R-type							
99	beq							



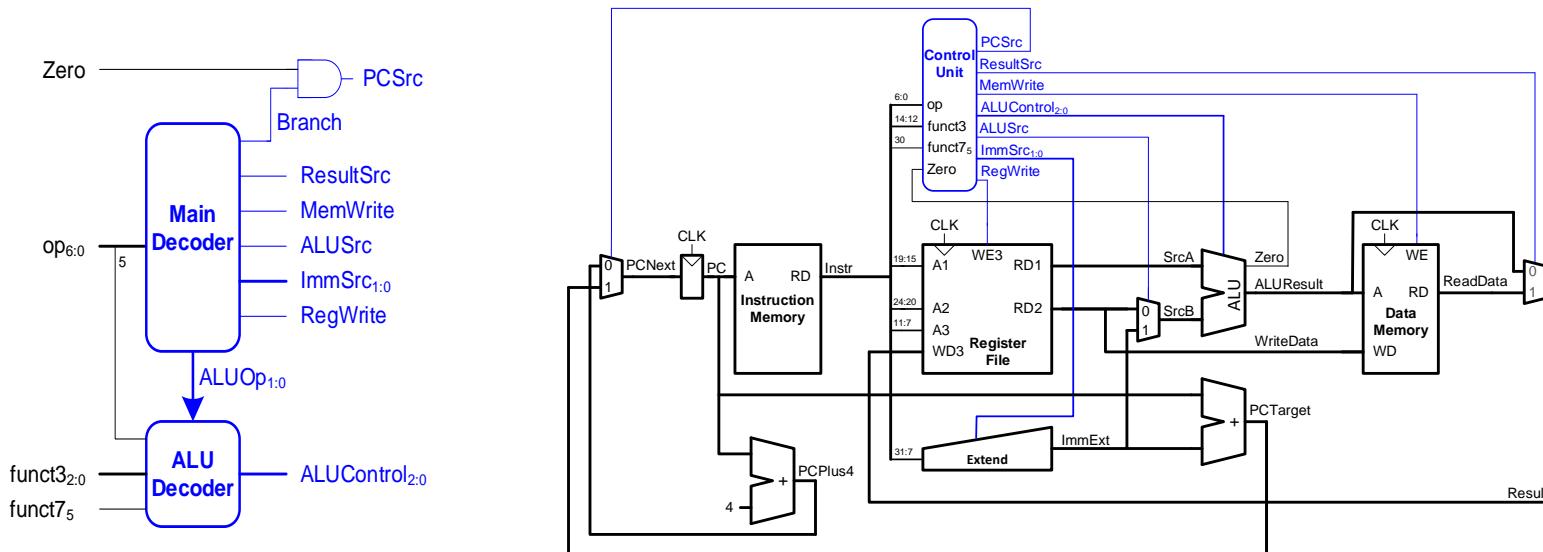
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0			
35	sw							
51	R-type							
99	beq							



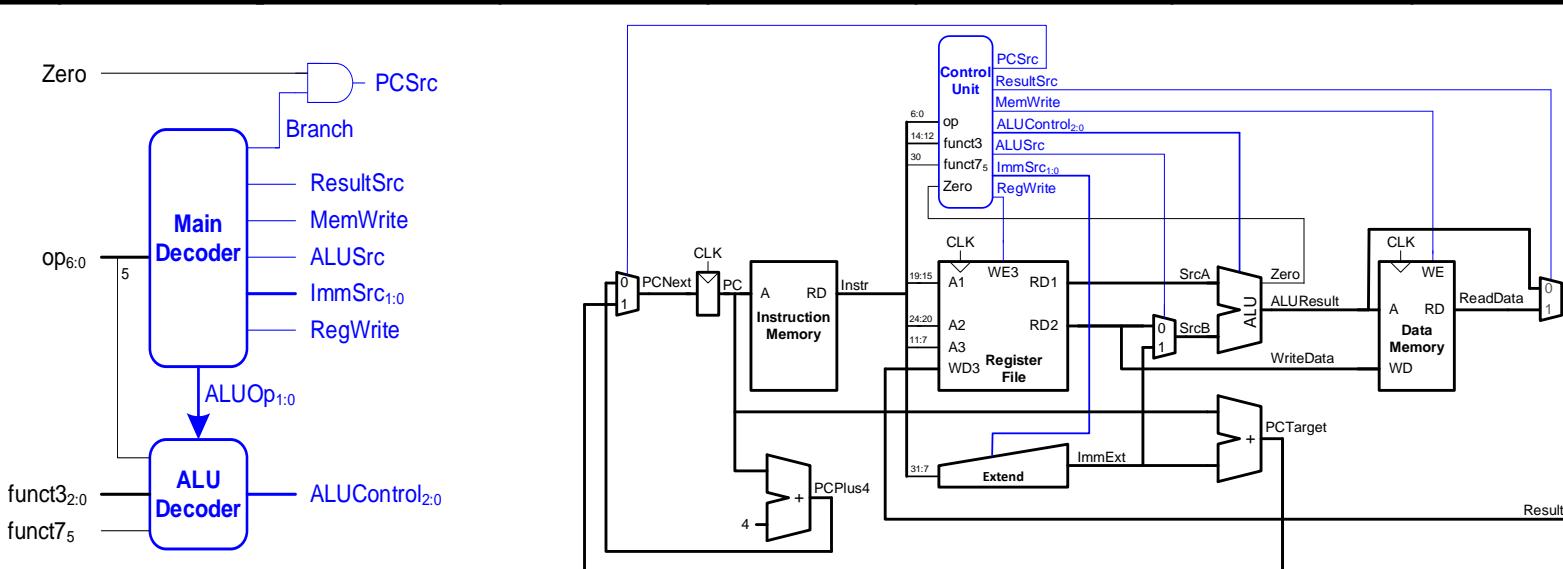
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1		
35	sw							
51	R-type							
99	beq							



Single-Cycle Control: Main Decoder

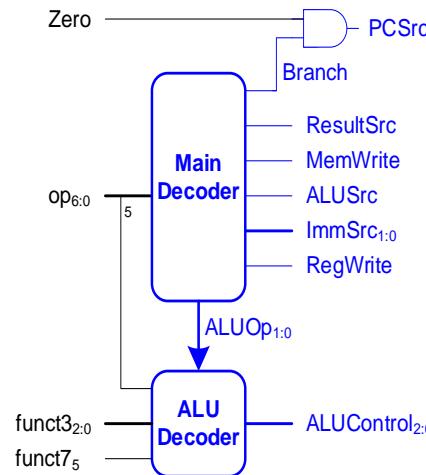
op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	
35	sw							
51	R-type							
99	beq							



Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw							
51	R-type							
99	beq							

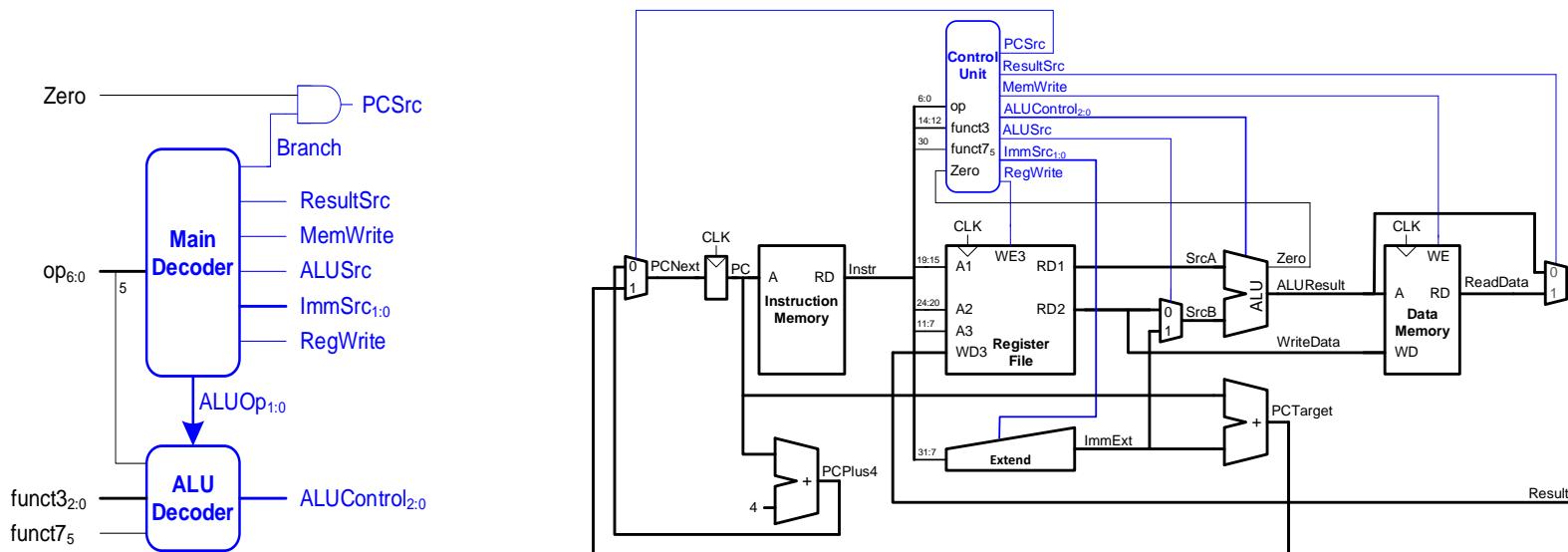
Table 7.3 ALU Decoder truth table



ALUOp	funct3	{op ₅ , funct7 ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
010	x	101 (set less than)	slt	
110	x	011 (or)	or	
111	x	010 (and)	and	

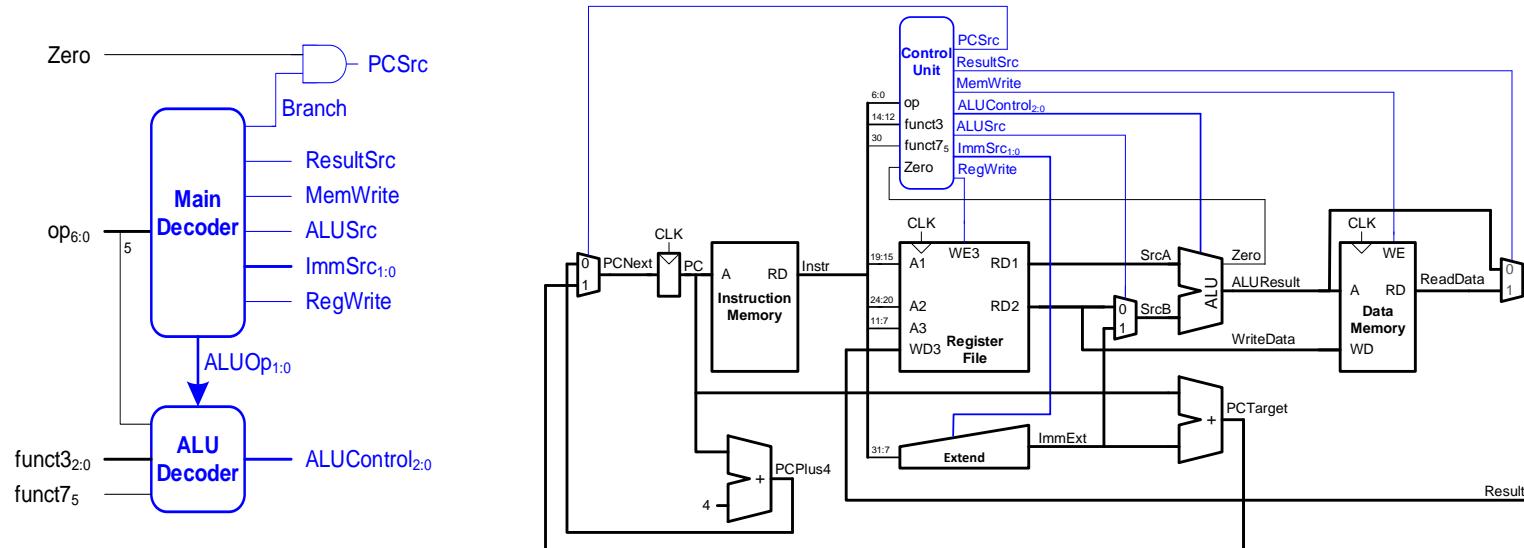
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0						
51	R-type							
99	beq							



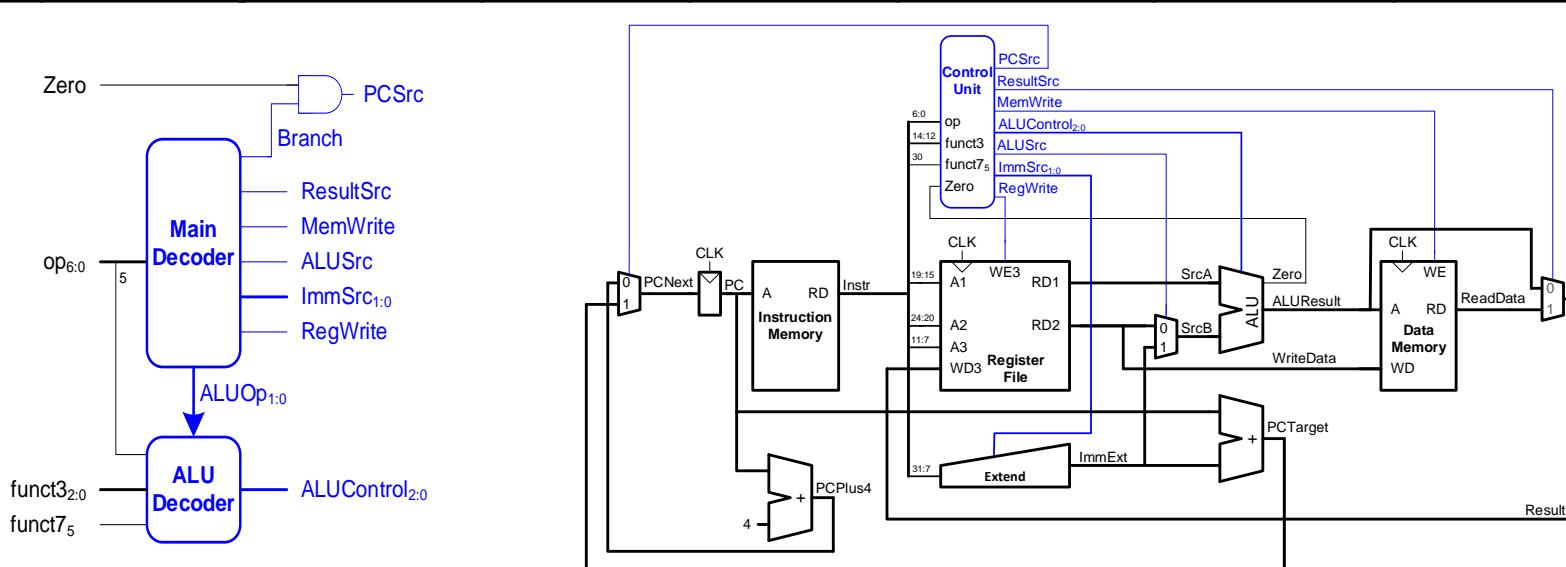
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	ImmSrc_{1:0}	ImmExt	Instruction Type		
51	R-type			00	{20{instr[31]}}, instr[31:20]	I-Type		
51	R-type			01	{20{instr[31]}}, instr[31:25], instr[11:7]	S-Type		
99	beq			10	{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0	B-Type		



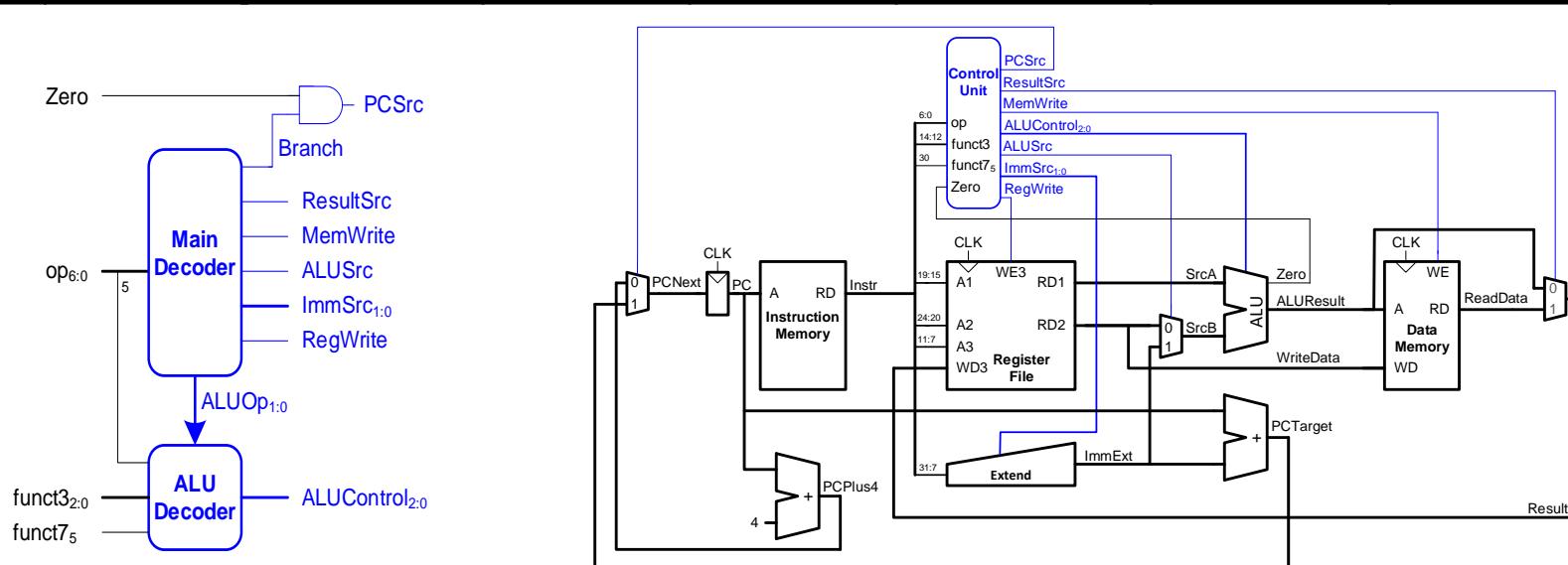
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1				
51	R-type							
99	beq							



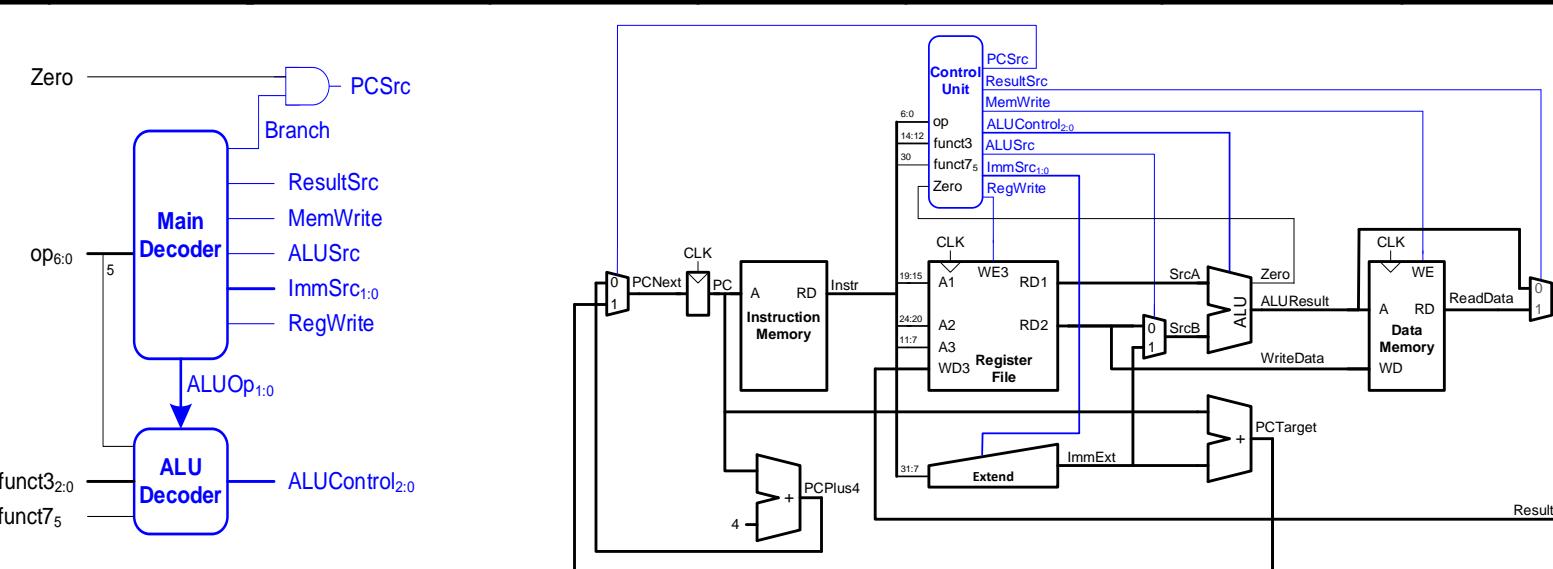
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1			
51	R-type							
99	beq							



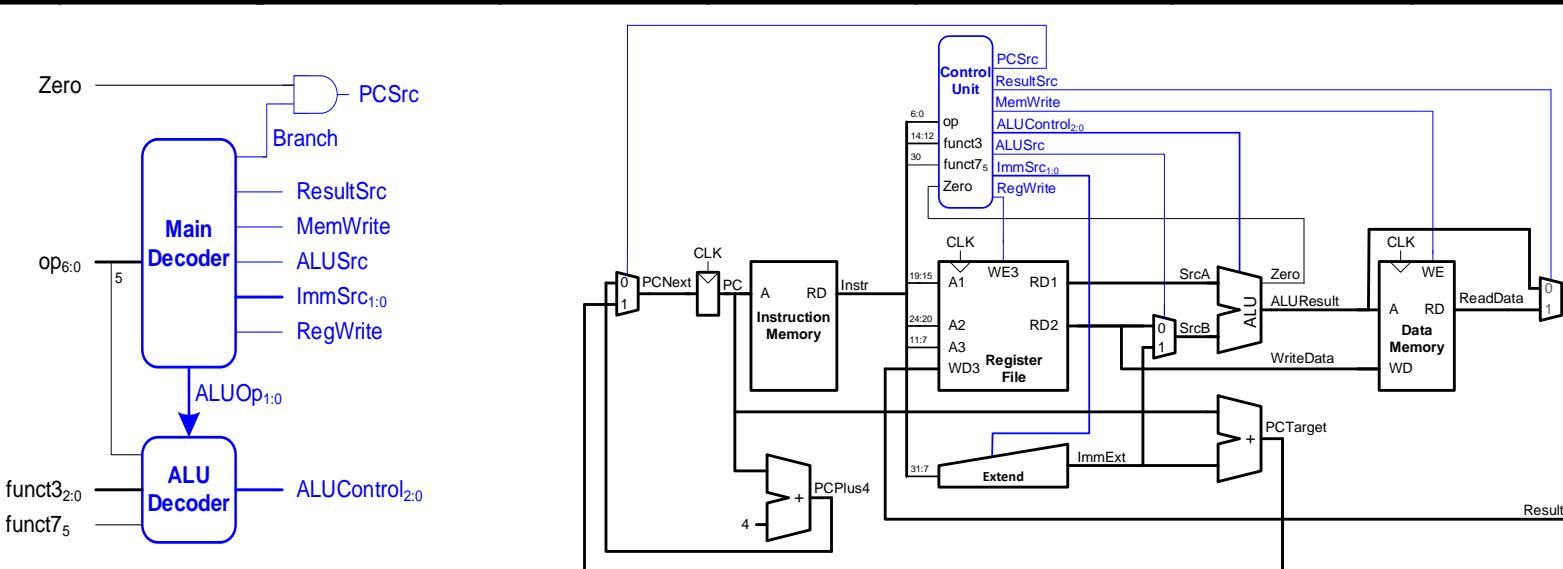
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X		
51	R-type							
99	beq							



Single-Cycle Control: Main Decoder

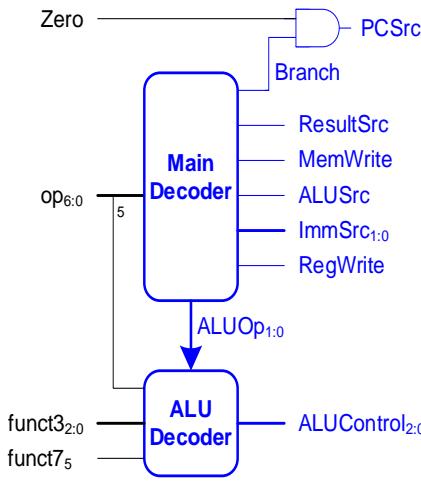
op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	
51	R-type							
99	beq							



Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type							
99	beq							

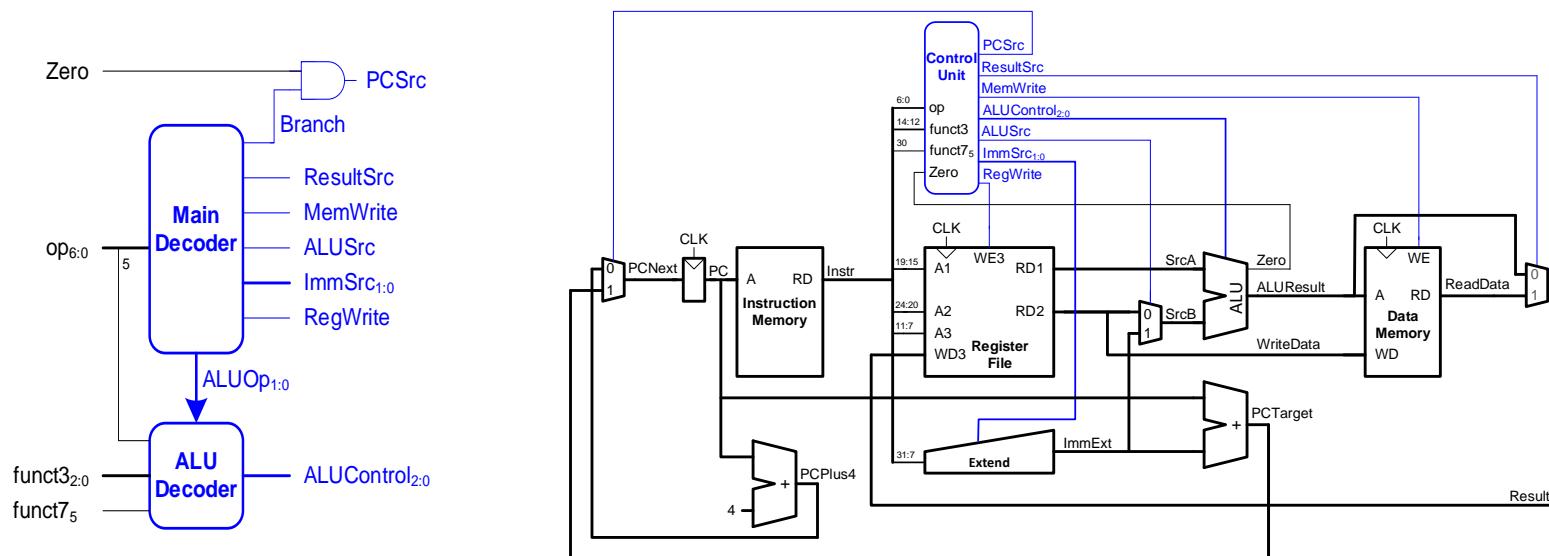
Table 7.3 ALU Decoder truth table



ALUOp	funct3	{op ₅ , funct7 ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
010	x	101 (set less than)	slt	
110	x	011 (or)	or	
111	x	010 (and)	and	

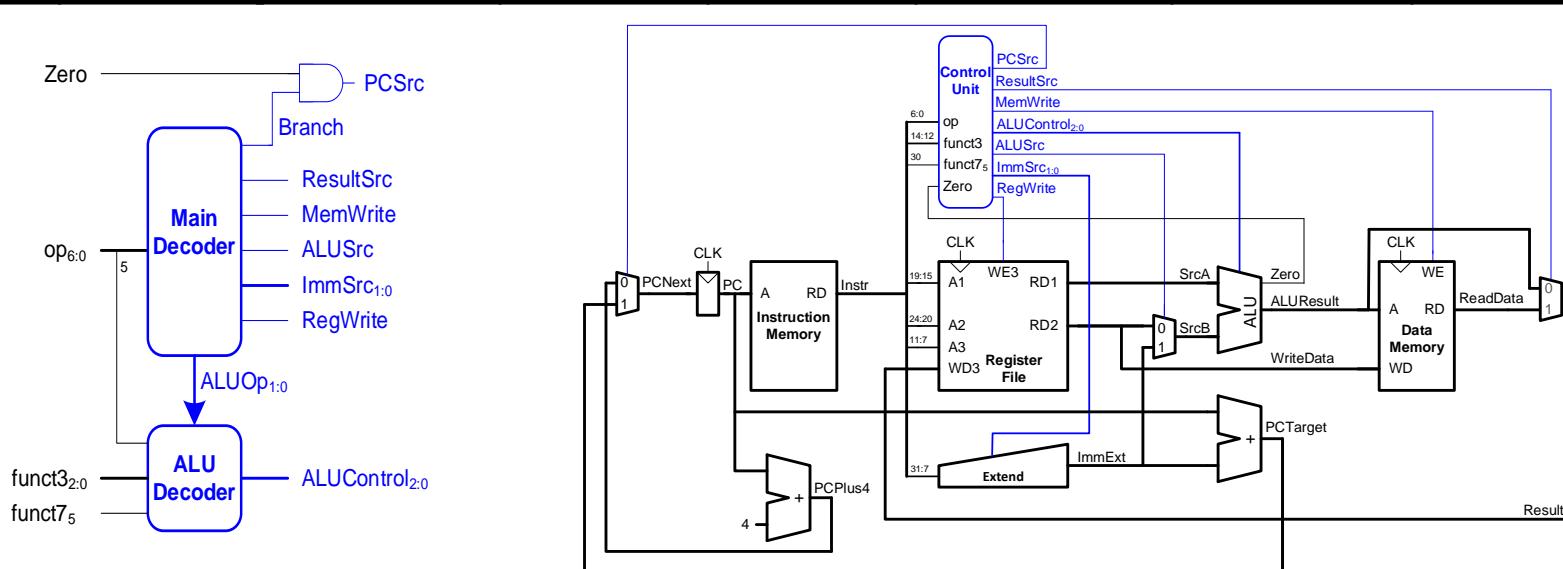
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1						
99	beq							



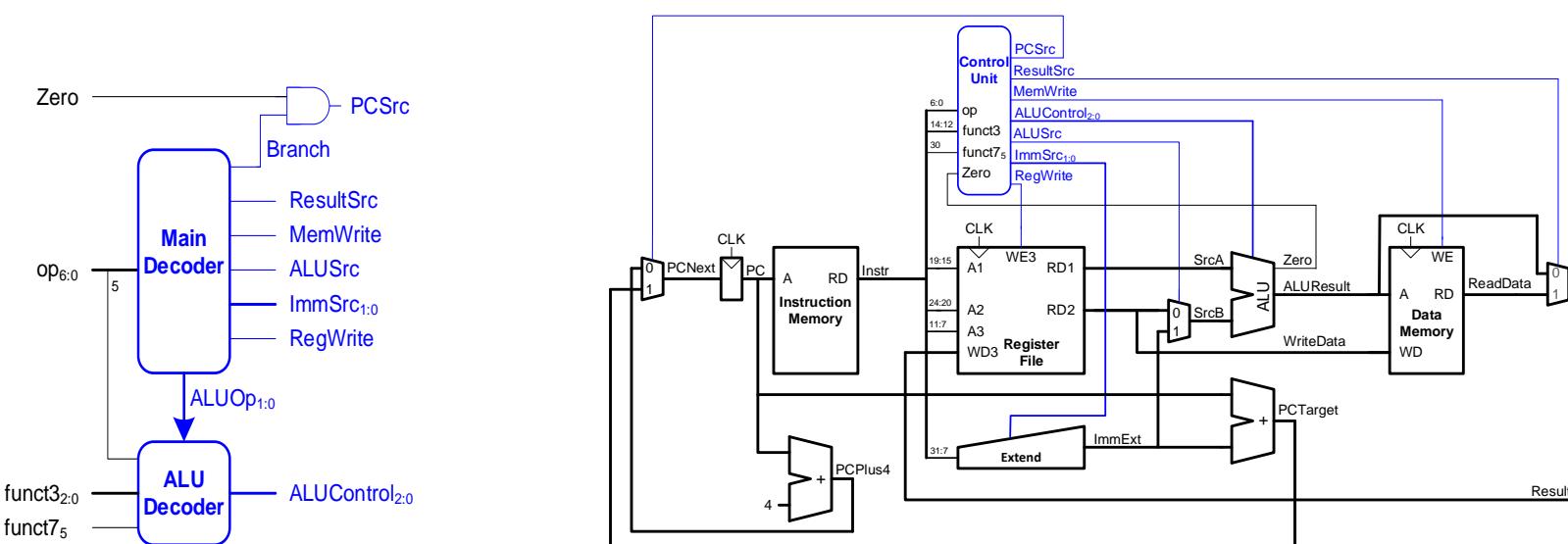
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX					
99	beq							



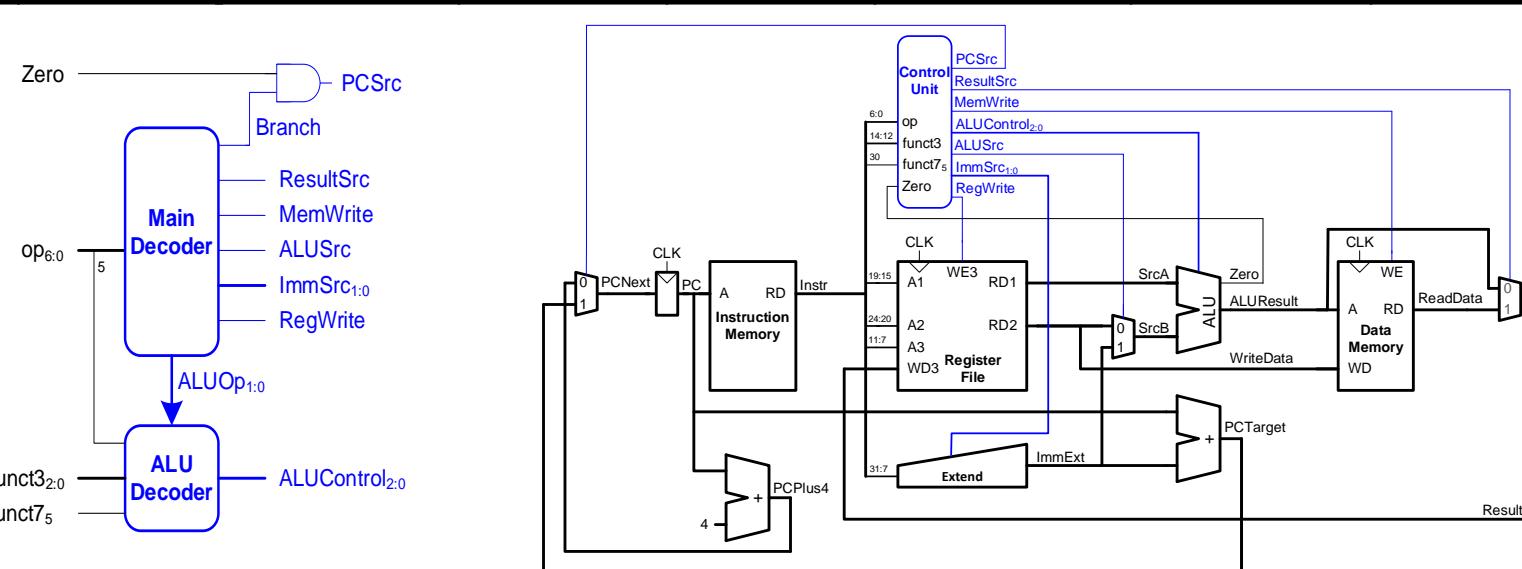
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0				
99	beq							



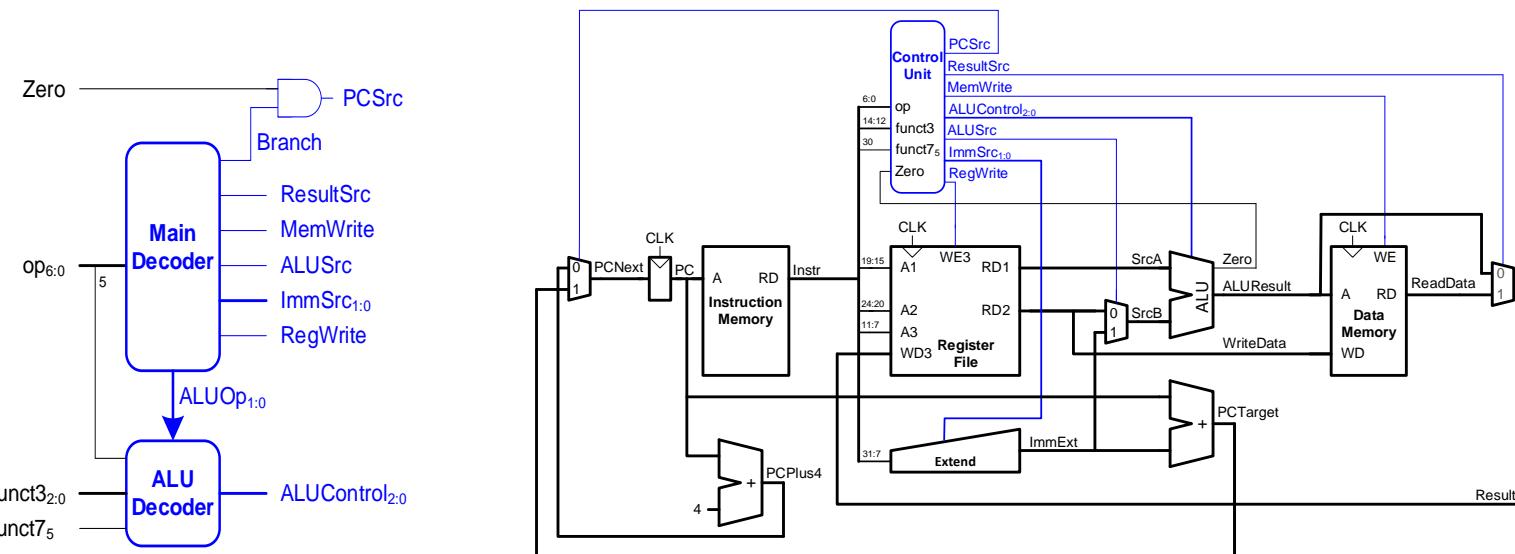
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0			
99	beq							



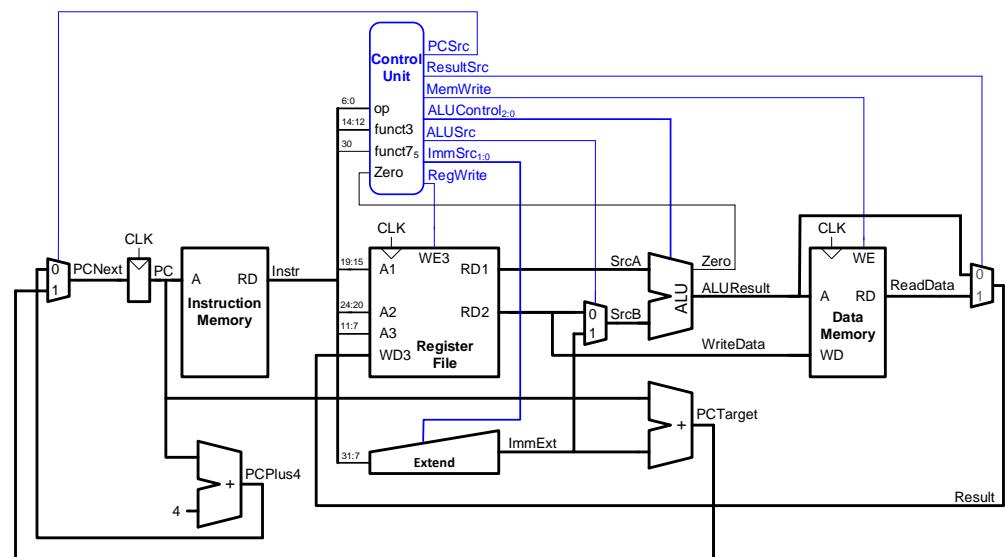
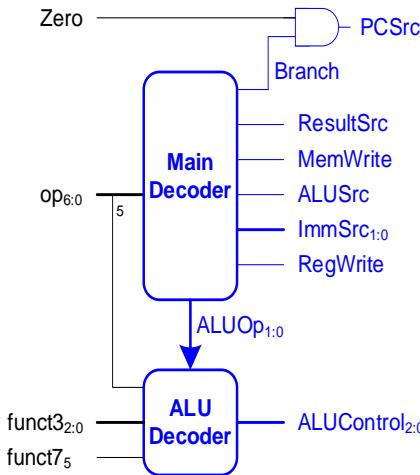
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0		
99	beq							



Single-Cycle Control: Main Decoder

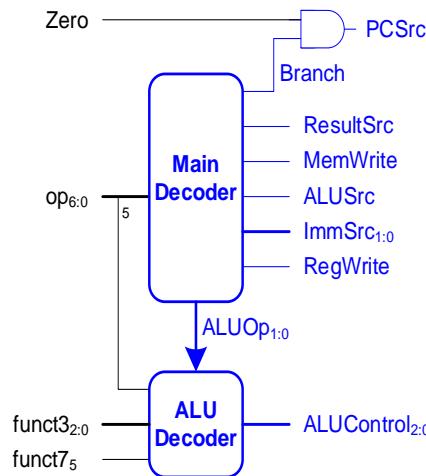
op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	
99	beq							



Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq							

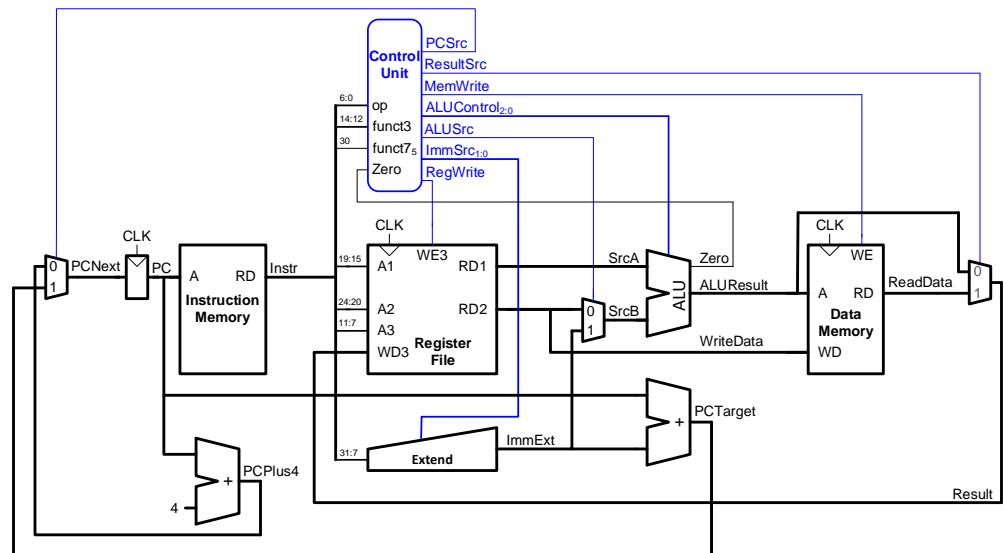
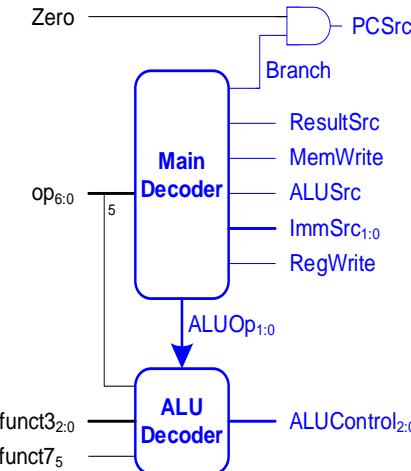
Table 7.3 ALU Decoder truth table



ALUOp	funct3	{op ₅ , funct7 ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
010	x	101 (set less than)	slt	
110	x	011 (or)	or	
111	x	010 (and)	and	

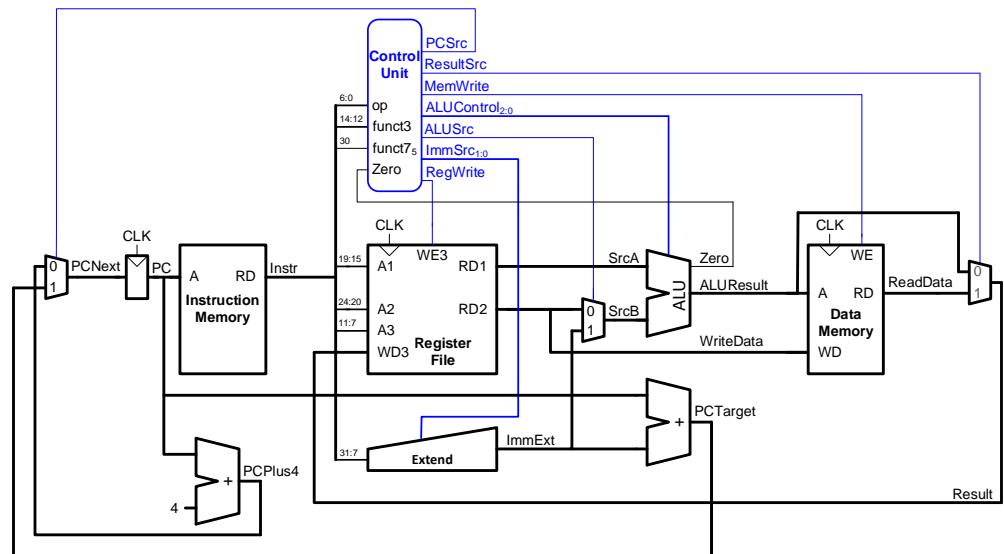
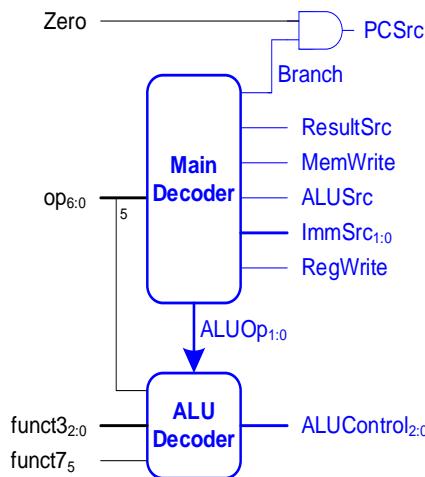
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0						



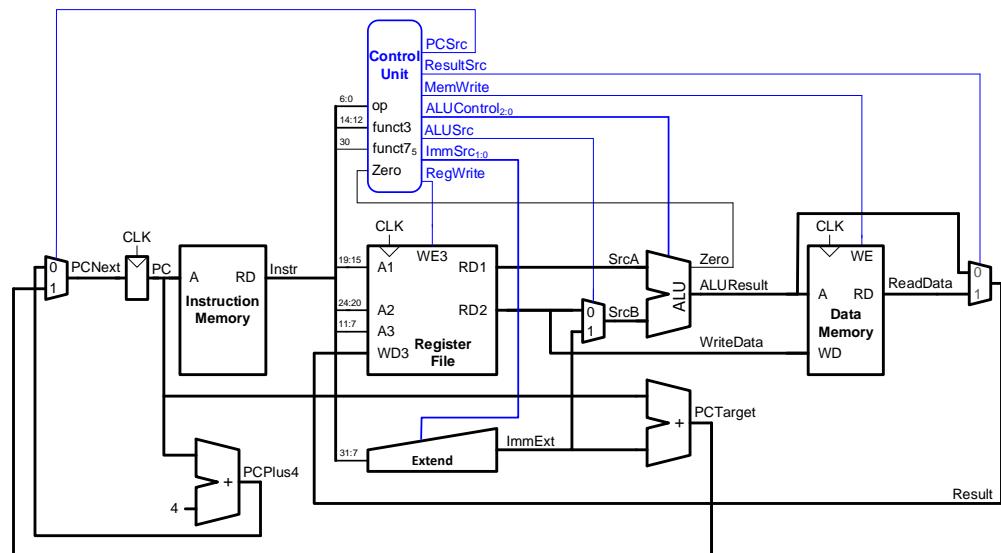
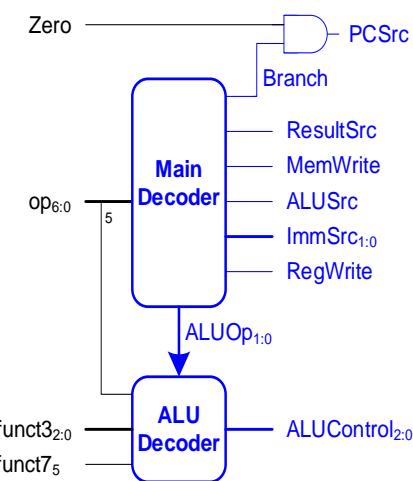
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp									
3	lw	1	00	1	0	1	0	00									
35	sw	0	01	1	1	X	0	00									
51	R-type	1	XX	<table border="1"> <tr> <th>ImmSrc_{1:0}</th> <th>ImmExt</th> <th>Instruction Type</th> </tr> <tr> <td>00</td> <td>{20[instr[31]}, instr[31:20]}</td> <td>I-Type</td> </tr> <tr> <td>01</td> <td>{20[instr[31]}, instr[31:25], instr[11:7]}</td> <td>S-Type</td> </tr> <tr> <td>10</td> <td>{19[instr[31]}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}</td> <td>B-Type</td> </tr> </table>	ImmSrc _{1:0}	ImmExt	Instruction Type	00	{20[instr[31]}, instr[31:20]}	I-Type	01	{20[instr[31]}, instr[31:25], instr[11:7]}	S-Type	10	{19[instr[31]}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type	10
ImmSrc _{1:0}	ImmExt	Instruction Type															
00	{20[instr[31]}, instr[31:20]}	I-Type															
01	{20[instr[31]}, instr[31:25], instr[11:7]}	S-Type															
10	{19[instr[31]}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type															
99	beq	0	10														



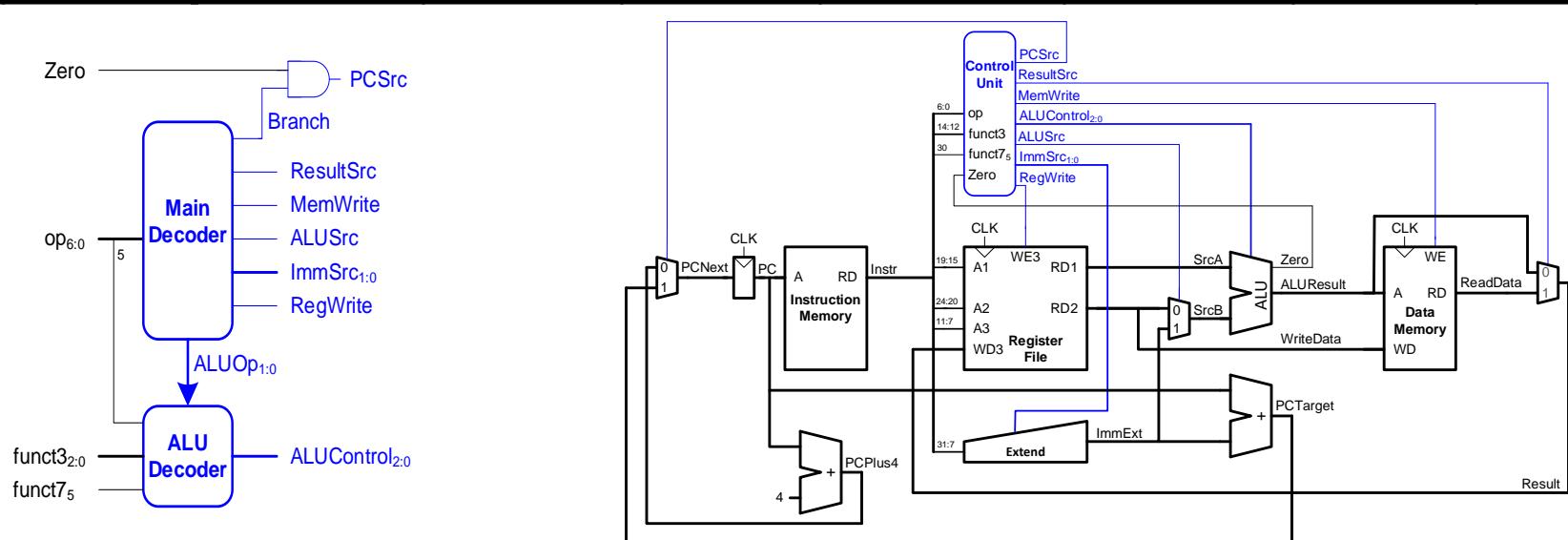
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0				



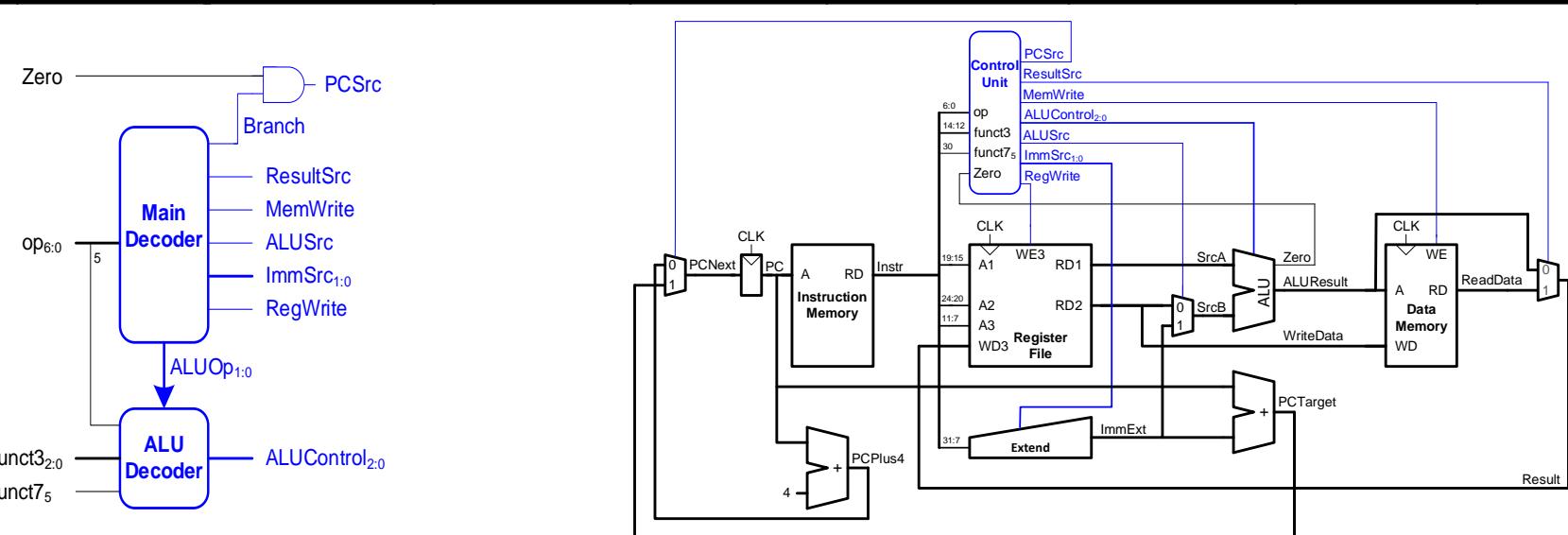
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0	0			



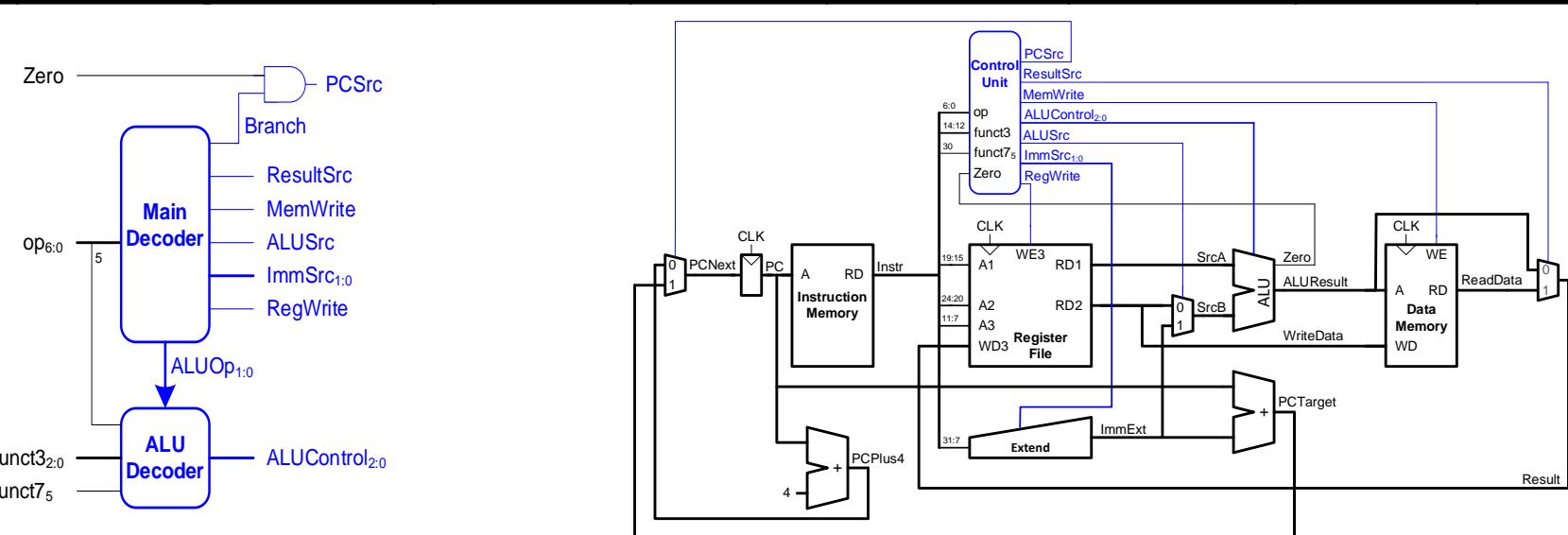
Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0	0	X		



Single-Cycle Control: Main Decoder

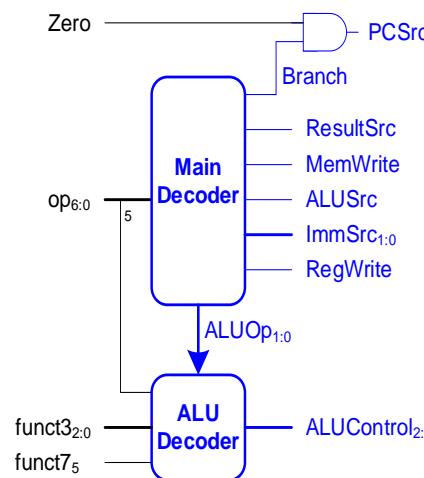
op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0	0	X	1	



Single-Cycle Control: Main Decoder

op	Instr.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0	0	X	1	01

Table 7.3 ALU Decoder truth table



ALUOp	funct3	{op ₅ , funct7 ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
010	x	101 (set less than)	slt	
110	x	011 (or)	or	
111	x	010 (and)	and	

Review: ALU

ALUControl _{2:0}	Function
000	add
001	subtract
010	and
011	or
101	SLT

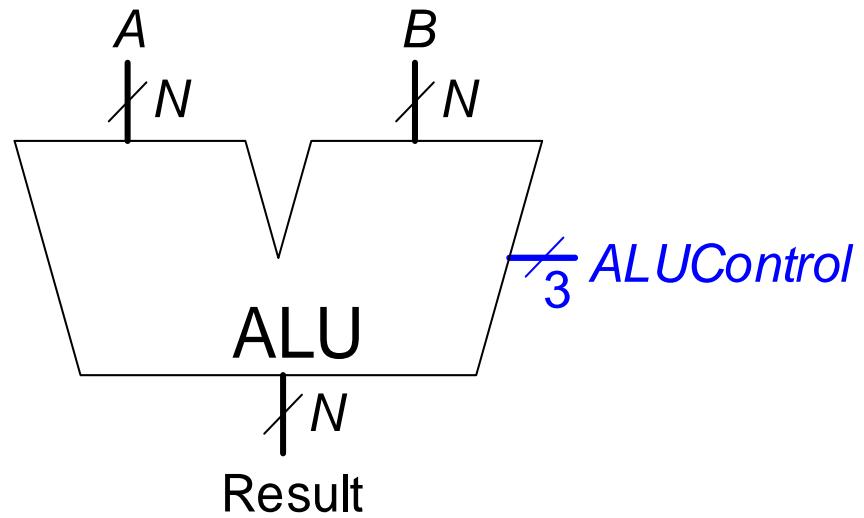


Table 7.3 ALU Decoder truth table

ALUOp	funct3	{op ₅ , funct7 ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

Review: ALU

ALUControl _{2:0}	Function
000	add
001	subtract
010	and
011	or
101	SLT

- We use $ALUControl = 101$ for the SLT operation and take advantage of the fact that making $ALUControl/0 = 1$ causes the adder to perform $A - B$.
- When $Sum_{N-1} = 1$, the result of $A - B$ is negative, and A is less than B . So, we zero-extend Sum_{N-1} and feed it into the 101 multiplexer input to complete the SLT operation.
- This implementation does not account for overflow.

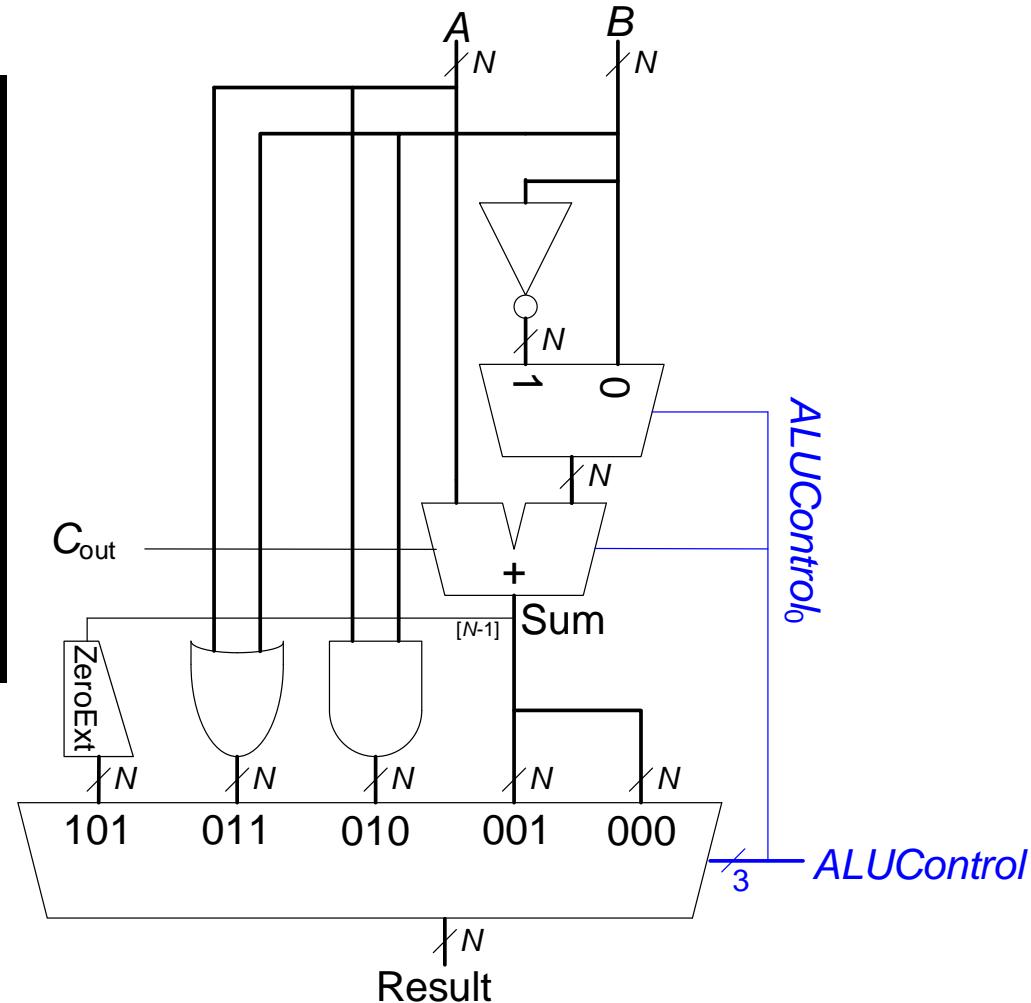
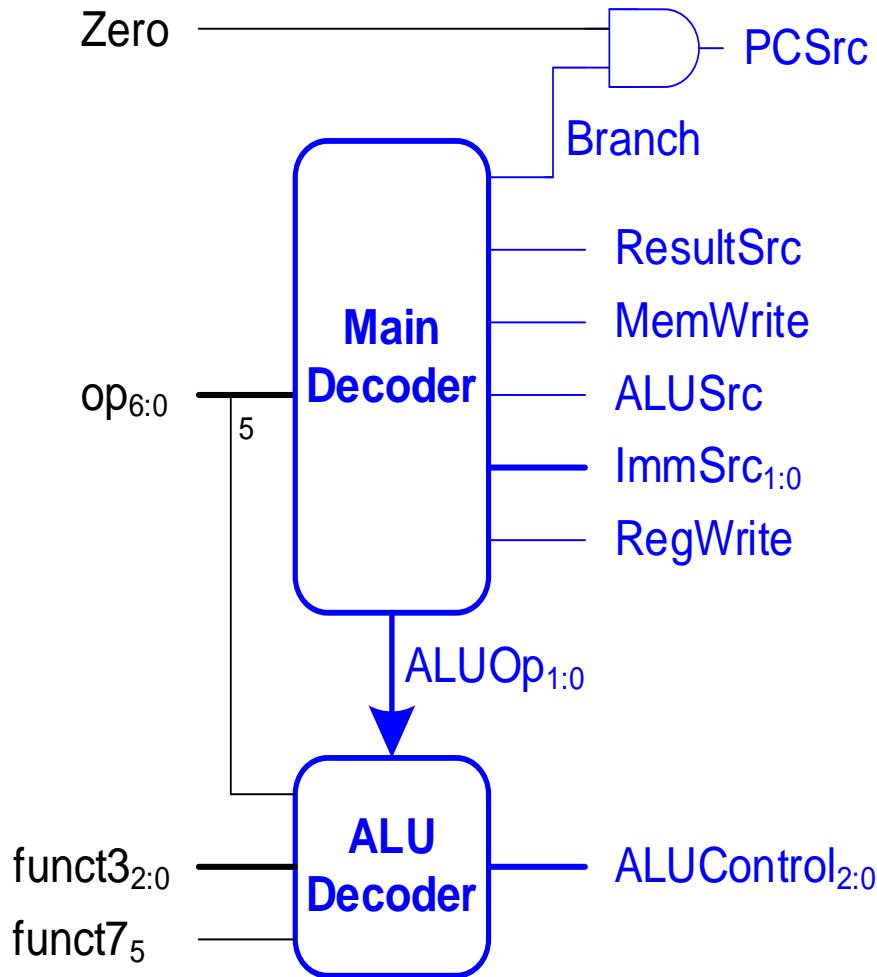


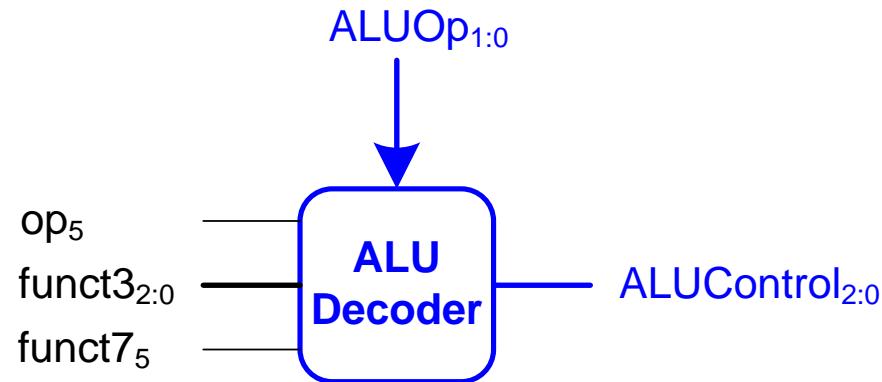
Figure 5.18 ALU (not accounting for overflow)

Single-Cycle Control: ALU Decoder



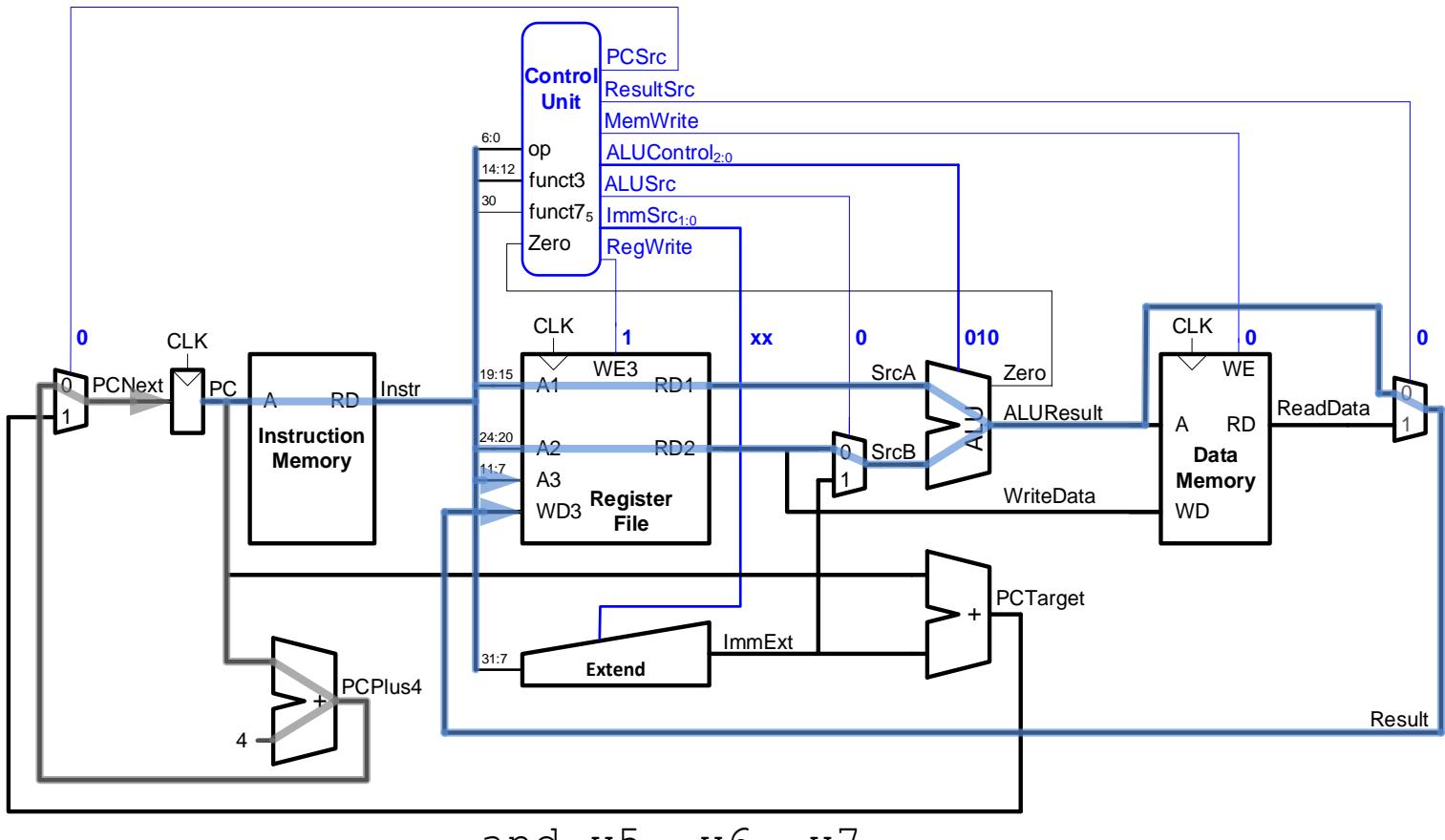
Single-Cycle Control: ALU Decoder

<i>ALUOp</i>	<i>funct3</i>	<i>op₅</i> , <i>funct7₅</i>	<i>Instruction</i>	<i>ALUControl_{2:0}</i>
00	x	x	lw, sw	000 (add)
01	x	x	beq	001 (subtract)
10	000	00, 01, 10	add	000 (add)
	000	11	sub	001 (subtract)
	010	x	slt	101 (set less than)
	110	x	or	011 (or)
	111	x	and	010 (and)



Example: and

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
51	R-type	1	XX	0	0	0	0	10



and x5, x6, x7

Chapter 7: Microarchitecture

**Extending the
Single-Cycle
Processor**

Extended Functionality: I-Type ALU

Enhance the single-cycle processor to handle **I-Type ALU instructions**: addi, andi, ori, and slti

- **Similar to R-type** instructions
- But **second source** comes from **immediate**
- Change ***ALUSrc*** to select the immediate
- And ***ImmSrc*** to pick the correct immediate

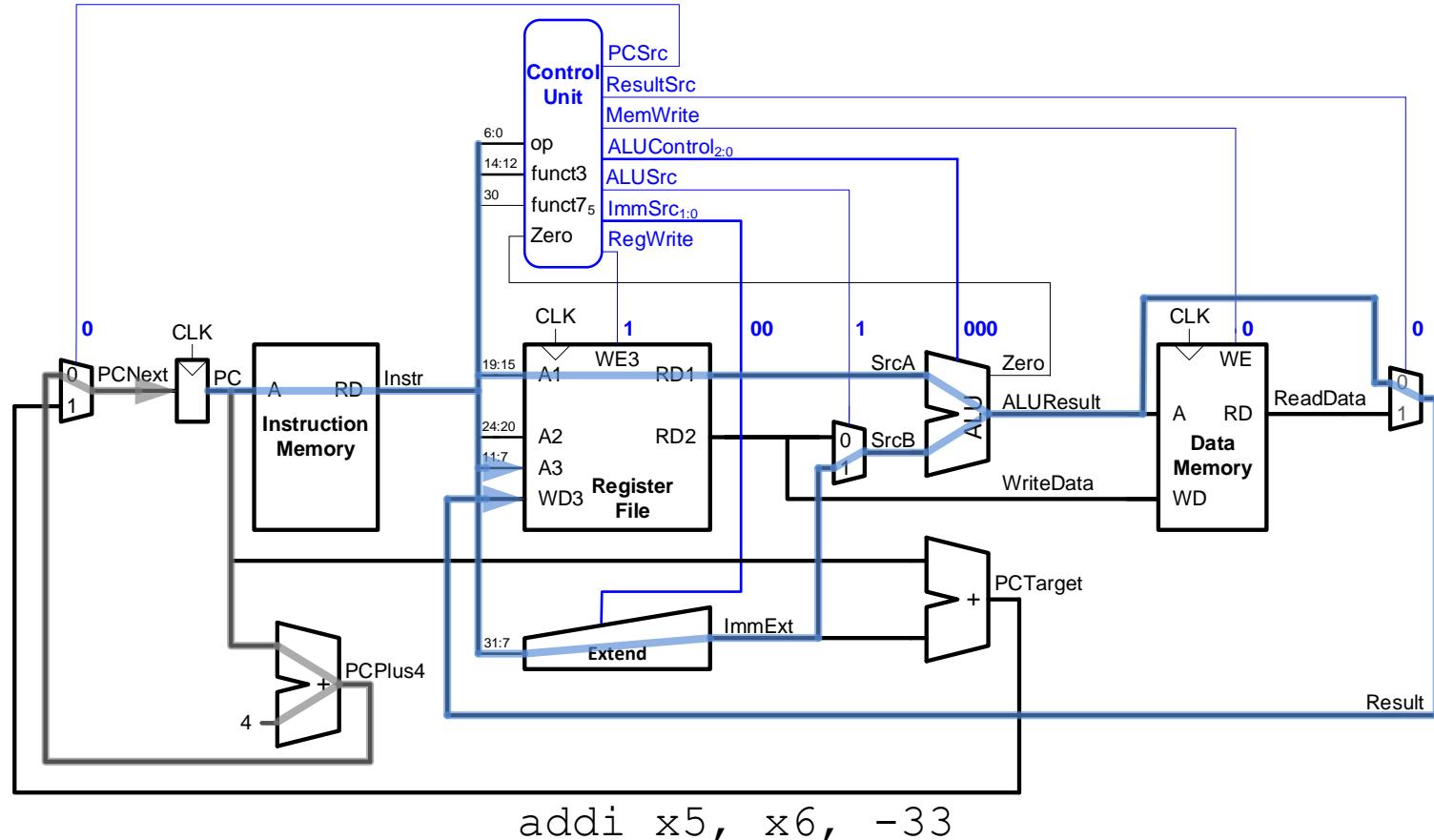
Extended Functionality: I-Type ALU

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	lw	1	00	1	0	1	0	00
35	sw	0	01	1	1	X	0	00
51	R-type	1	XX	0	0	0	0	10
99	beq	0	10	0	0	X	1	01
19	I-type	1	00	1	0	0	0	10

- This change also provides the other I-type ALU instructions: andi, ori, and slti.
- These other instructions share the same **op** value of 0010011_2 , need the same control signals, and only differ in the **funct3** field, which the ALU Decoder already uses to determine *ALUControl* and, thus, the ALU operation.

Extended Functionality: addi

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
19	I-type	1	00	1	0	0	0	10



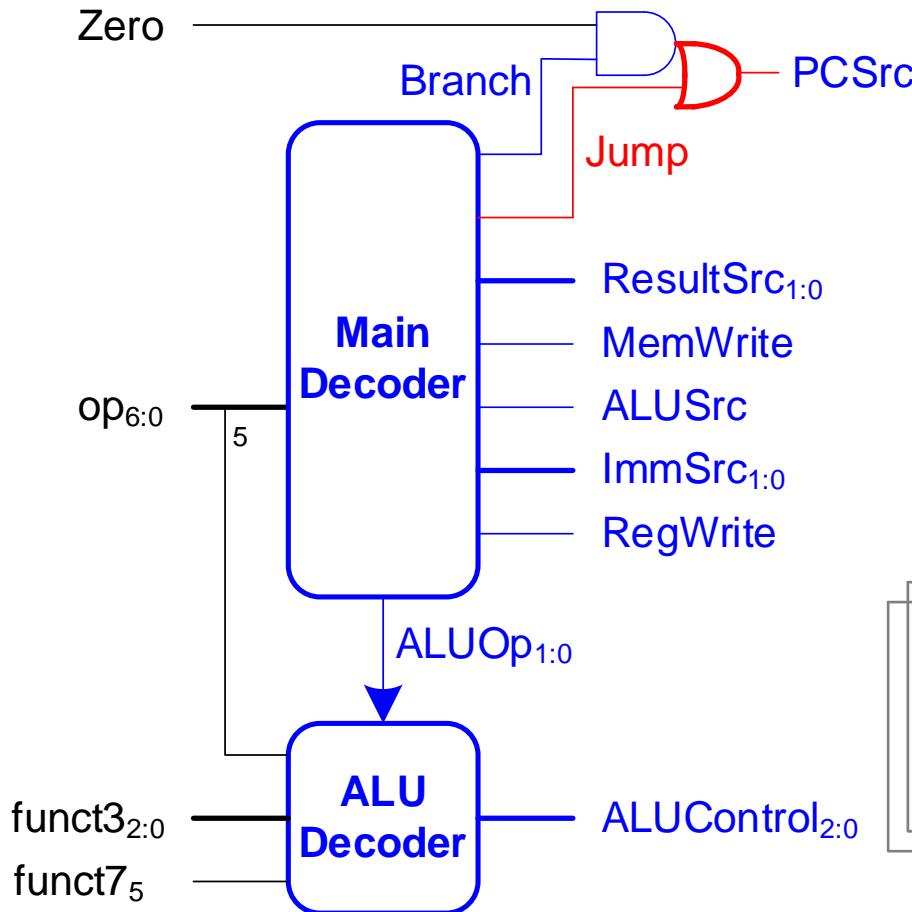
Extended Functionality: jal

Enhance the single-cycle processor to handle jal

- **Similar to beq**
- But jump is **always taken**
 - PCSrc should be 1
- **Immediate format** is different
 - Need a new ImmSrc of 11
- And jal must **compute PC+4** and **store in rd**
 - Take PC+4 from adder through ResultMux

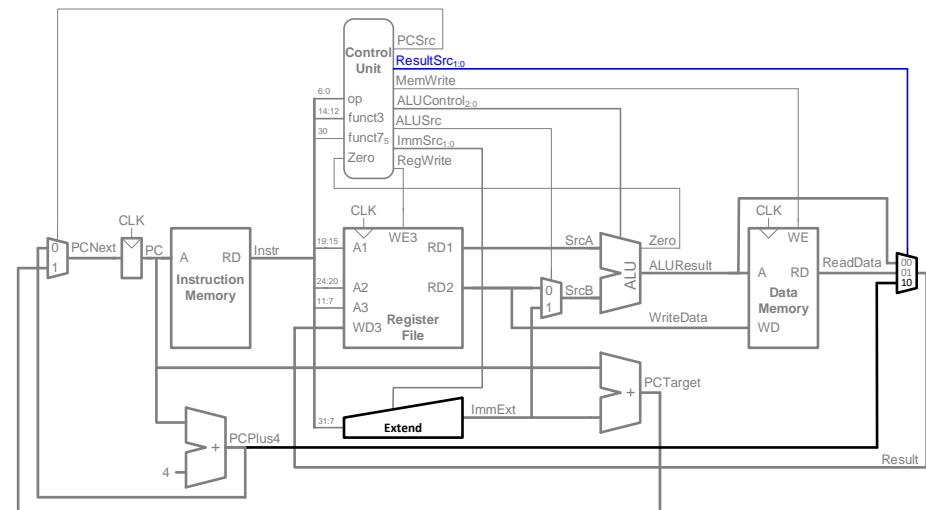
The jump and link (jal) instruction writes PC+4 to **rd** and changes PC to the jump target address, PC + **imm**.

Extended Functionality: jal



- Similar to beq
- But jump is always taken

PCSrc should be 1



Extended Functionality: *ImmExt*

ImmSrc_{1:0}	ImmExt	Instruction Type
00	$\{\{20\{\text{instr}[31]\}\}, \text{instr}[31:20]\}$	I-Type
01	$\{\{20\{\text{instr}[31]\}\}, \text{instr}[31:25], \text{instr}[11:7]\}$	S-Type
10	$\{\{19\{\text{instr}[31]\}\}, \text{instr}[31], \text{instr}[7], \text{instr}[30:25], \text{instr}[11:8], 1'b0\}$	B-Type
11	$\{\{12\{\text{instr}[31]\}\}, \text{instr}[19:12], \text{instr}[20], \text{instr}[30:21], 1'b0\}$	J-Type

Immediate format is different, need a new *ImmSrc* of 11

I-Type

31:20	19:15	14:12	11:7	6:0
$\text{imm}_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$\text{imm}_{12,10:5}$	rs2	rs1	funct3	$\text{imm}_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$\text{imm}_{11:5}$	rs2	rs1	funct3	$\text{imm}_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

J-Type

31:12	11:7	6:0
$\text{imm}_{20,10:1,11,19:12}$	rd	op
20 bits	5 bits	7 bits

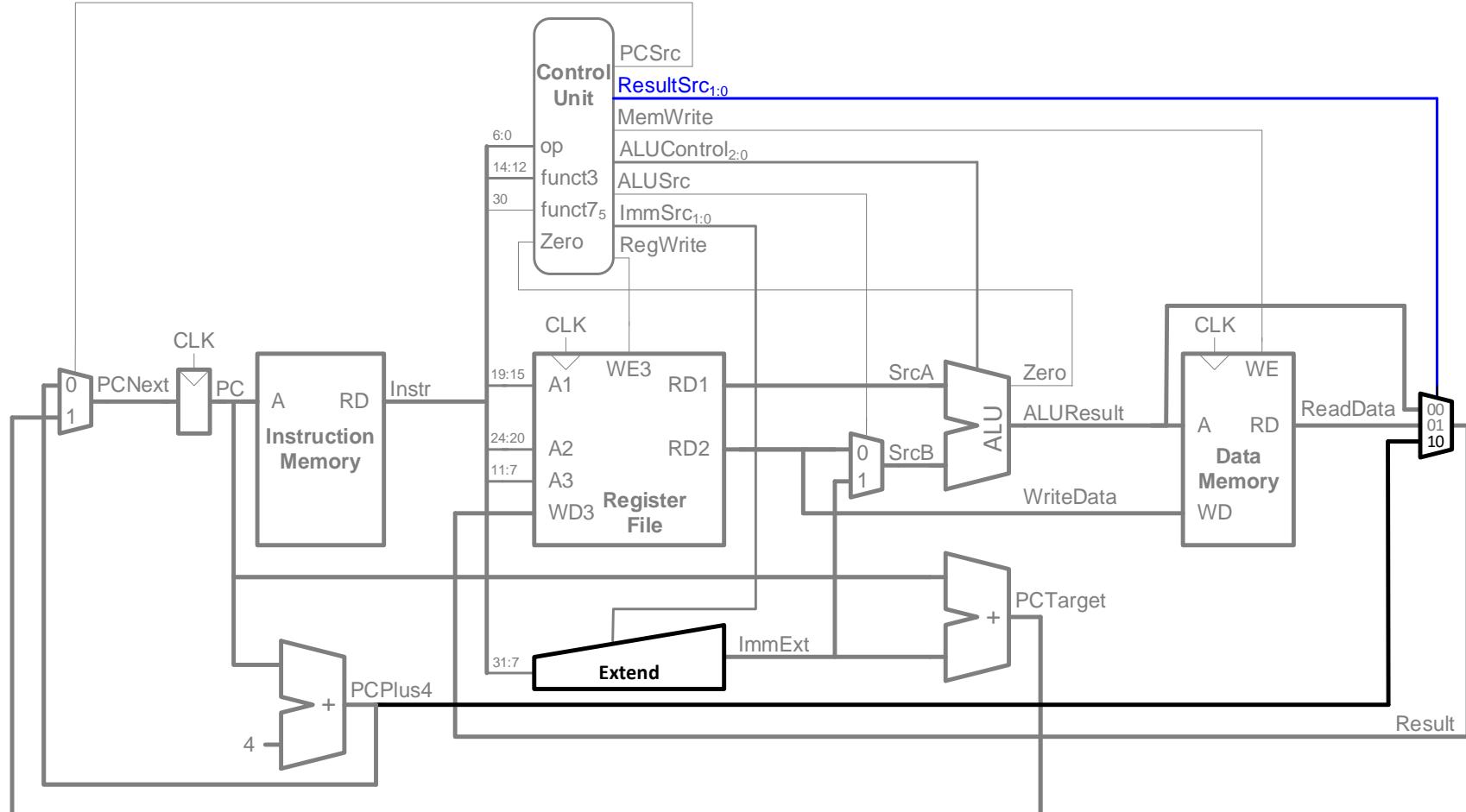
Extended Functionality: jal

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
3	lw	1	00	1	0	01	0	00	0
35	sw	0	01	1	1	XX	0	00	0
51	R-type	1	XX	0	0	00	0	10	0
99	beq	0	10	0	0	XX	1	01	0
19	I-type	1	00	1	0	00	0	10	0
111	jal	1	11	X	0	10	0	XX	1

- The processor calculates the jump target address, the value of *PCNext*, by adding PC to the 21-bit signed immediate encoded in the instruction.
 - The least significant bit of the immediate is always 0 and the next 20 most significant bits come from *Instr31:12*.
 - This 21-bit immediate is then sign-extended.
- The datapath already has hardware for adding PC to a sign-extended immediate, selecting this as the next PC, computing PC+4, and writing a value to the register file.
- Hence, in the datapath, we must only modify the Extend unit to
 - sign-extend the 21-bit immediate and encoding *ImmSrc* to support the long immediate for *jal*.
 - expand the Result multiplexer to choose PC+4 (i.e., *PCPlus4*) as shown above

Extended Functionality: jal

op	Instruct.	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
111	jal	1	11	X	0	10	0	XX	1



Chapter 7: Microarchitecture

Single-Cycle Performance

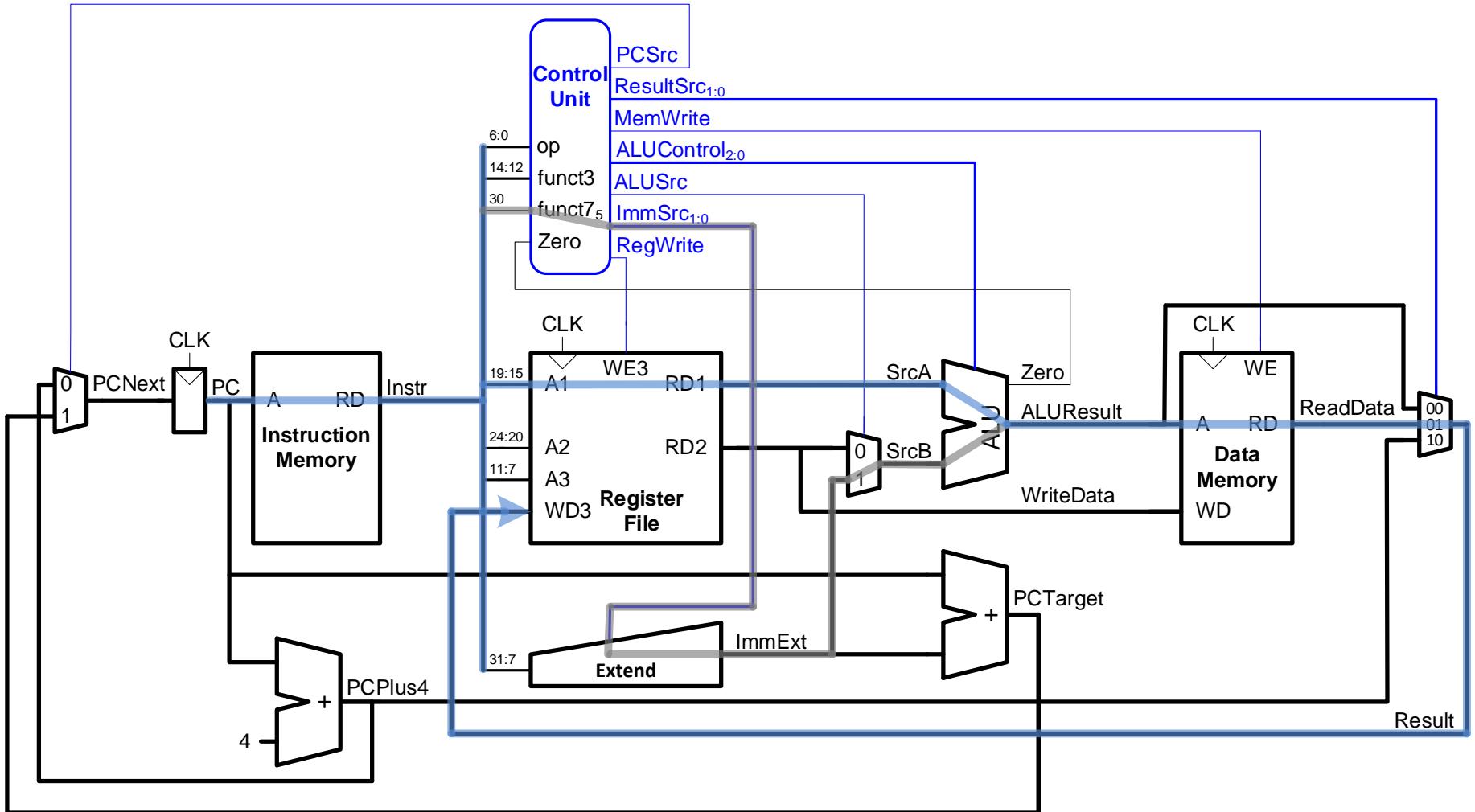
Processor Performance

Program Execution Time

$$= (\# \text{instructions})(\text{cycles/instruction})(\text{seconds/cycle})$$

$$= \# \text{ instructions} \times \text{CPI} \times T_C$$

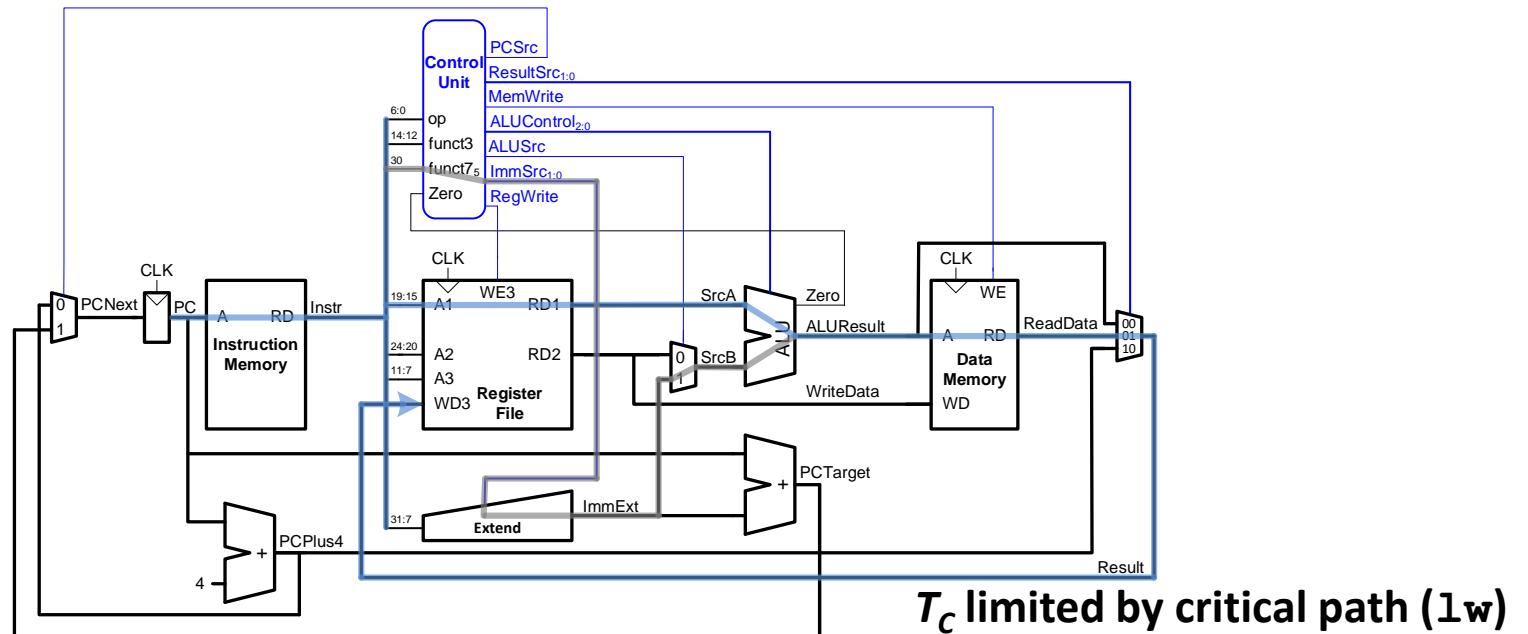
Single-Cycle Processor Performance



T_C limited by critical path (1w)

Single-Cycle Processor Performance

- In our processor, the **lw** instruction is the most time-consuming and involves the critical path shown in the Figure below.
- As indicated by heavy blue lines, the critical path starts with the PC loading a new address on the rising edge of the clock.
- The instruction memory then reads the new instruction, and the register file reads **rs1** as **SrcA**.
- While the register file is reading, the immediate field is sign-extended based on *ImmSrc* and selected at the **SrcB** multiplexer (path highlighted in gray).
- The ALU adds **SrcA** and **SrcB** to find the memory address.
- The data memory reads from this address, and the Result multiplexer selects **ReadData** as **Result**.
- Finally, **Result** must set up at the register file before the next rising clock edge so that it can be properly written.



Single-Cycle Processor Performance

- **Single-cycle critical path:**

$$T_{c_single} = t_{pcq_PC} + t_{mem} + \max[t_{RFread}, t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- **Typically, limiting paths are:**

- memory, ALU, register file

- So,
$$\begin{aligned} T_{c_single} &= t_{pcq_PC} + t_{mem} + t_{RFread} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} \\ &= t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup} \end{aligned}$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$T_{c_single} = t_{pcq_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$$
$$=$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$\begin{aligned}T_{c_single} &= t_{pcq_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}} \\&= (40 + 2*200 + 100 + 120 + 30 + 60) \text{ ps} = \mathbf{750 \text{ ps}}\end{aligned}$$

Single-Cycle Performance Example

Program with 100 billion instructions:

$$\text{Execution Time} = \# \text{ instructions} \times \text{CPI} \times T_C$$

Single-Cycle Performance Example

Program with 100 billion instructions:

$$\begin{aligned}\text{Execution Time} &= \# \text{ instructions} \times \text{CPI} \times T_C \\ &= (100 \times 10^9)(1)(750 \times 10^{-12} \text{ s}) \\ &= \mathbf{75 \text{ seconds}}\end{aligned}$$

Chapter 7: Microarchitecture

Multicycle RISC-V Processor

Single- vs. Multicycle Processor

- **Single-cycle:**
 - + simple
 - cycle time limited by longest instruction (l_w)
 - separate memories for instruction and data
 - 3 adders/ALUs
- **Multicycle processor** addresses these issues by breaking instruction into **shorter steps**
 - shorter instructions take fewer steps
 - can re-use hardware
 - cycle time is faster

Single- vs. Multicycle Processor

- **Single-cycle:**

- + simple
 - cycle time limited by longest instruction (l_w)
 - separate memories for instruction and data
 - 3 adders/ALUs

- **Multicycle:**

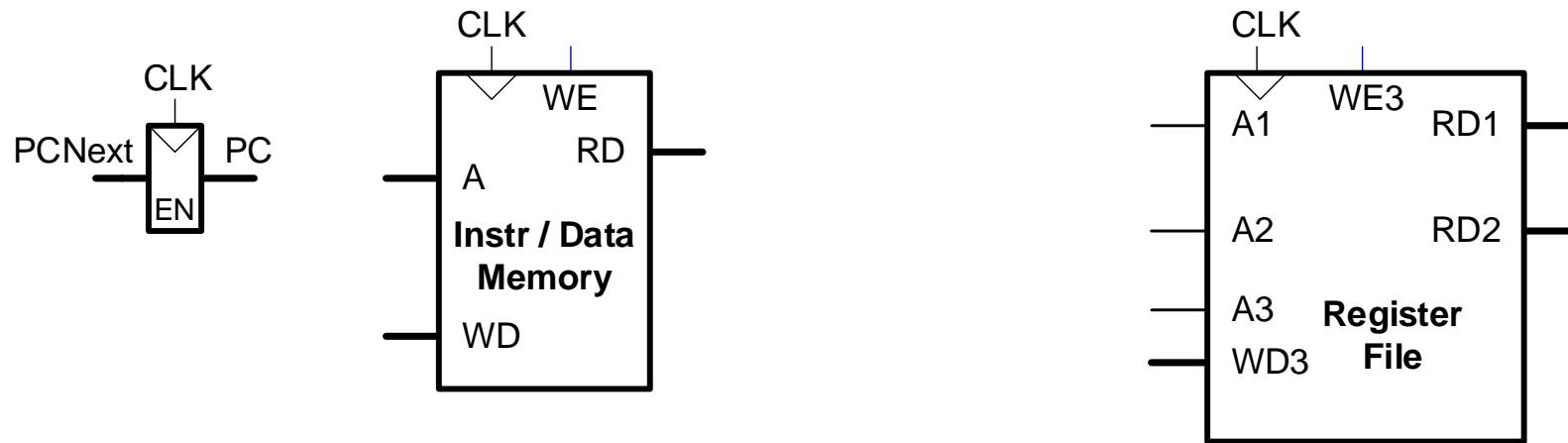
- + higher clock speed
- + simpler instructions run faster
- + reuse expensive hardware on multiple cycles
- sequencing overhead paid many times

Same design steps
as single-cycle:

- **first datapath**
- **then control**

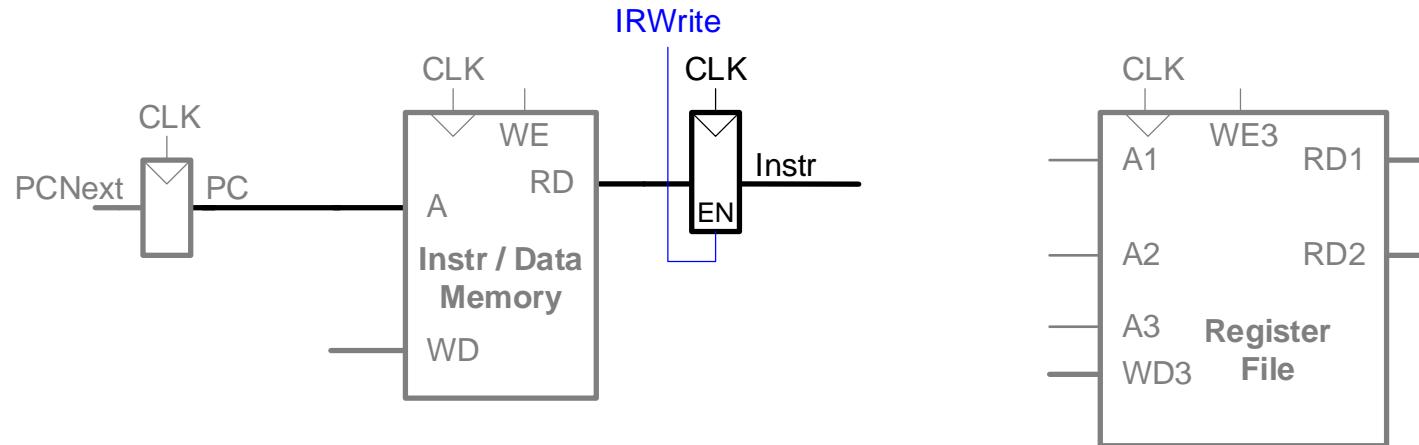
Multicycle State Elements

- Replace separate Instruction and Data memories with a **single unified memory** – more realistic
 - We can read the instruction in one cycle, then read or write the data in another cycle
- The PC and Register File remain unchanged



Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction

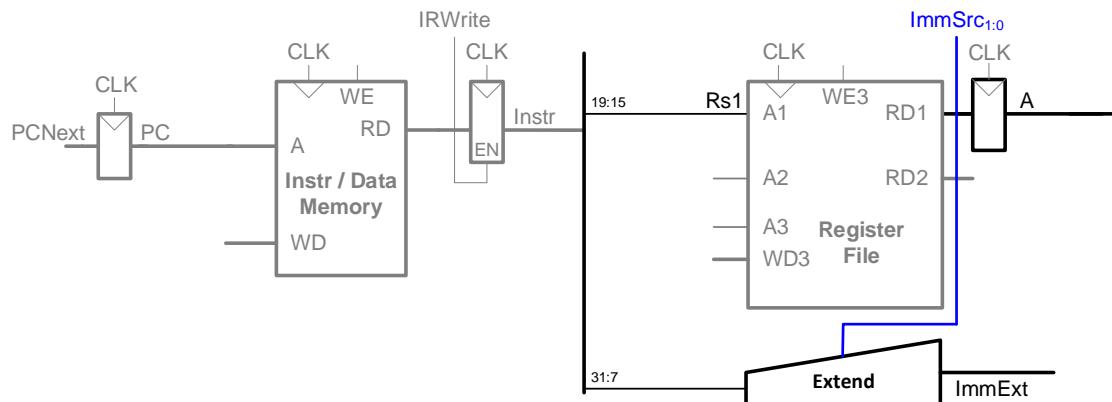


- As with the single-cycle processor, gradually build the data path by adding components to handle each instruction step.
- The PC contains the address of the instruction (l_w) to execute.
- The first step is to read this instruction from memory.
 - The PC is connected to the address input of the memory
 - The instruction is read and stored in a new non-architectural instruction register (IR), so it is available for future cycles.
 - The IR write-enable signal, *IRWrite*, is asserted when the IR should be loaded with a new instruction.

Multicycle Datapath: lw Get Sources

STEP 2: Read source operand from RF and extend immediate

First, build the datapath for the lw instruction.

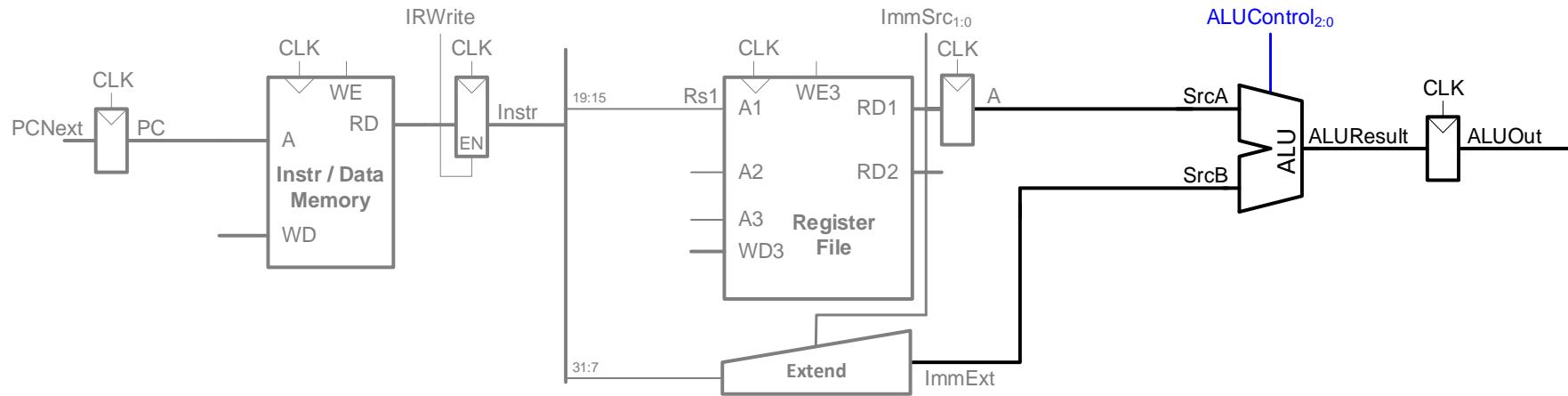


- After fetching lw , the 2nd step is to read the source register containing the base address.
- The source register is specified in the *rs1* field, $Instr_{19:15}$.
- The $Instr_{19:15}$ bits are connected to the Register file input address A1.
- The Register File reads the register onto RD1.
- The value from RD1 is stored in another non-architectural register, A.
- The lw instruction also requires a 12-bit offset in the immediate field of the instruction, $Instr_{31:20}$, which must be sign-extended to 32 bits.
- The Extend unit takes a 2-bit *ImmSrc* control signal to specify a 12-, 13-, or 21-bit immediate to extend for various types of instructions.
- The *ImmExt* is a combinational function of *Instr* and will not change while the current instruction is being processed, so no need to have a non-architectural register to hold the constant value.

Address	Instruction	Type		Fields	Machine Language
0x1000	L7: lw x6, -4 (x9)	I	$imm_{11:0}$ 111111111110	rs1 01001 f3 010 rd 00110 op 0000011 FFC4A303	

Multicycle Datapath: lw Address

STEP 3: Compute the memory address

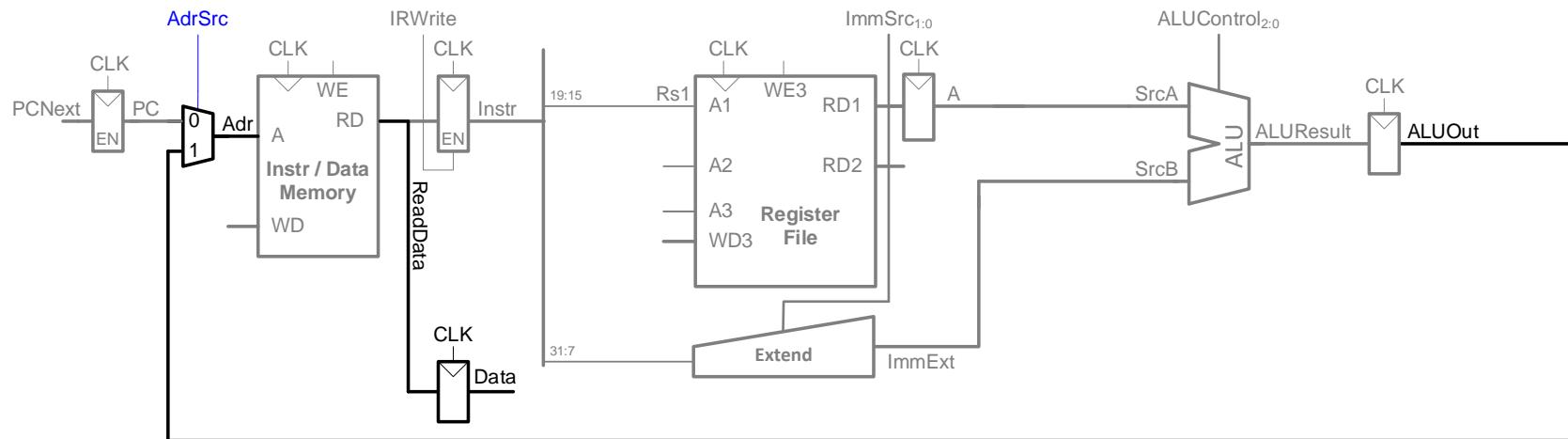


- In the third step, the ALU computes the memory address for the load instruction by adding the base address PC and the offset.
- The $ALUControl_{2:0}$ should be set to 000 to perform the addition.
- $ALUResult$ is stored in a non-architectural register, $ALUout$.

Address	Instruction	Type	Fields				Machine Language	
0x1000	L7: lw x6, -4(x9)	I	imm _{11:0}	rs1	f3	rd	op	FFC4A303

Multicycle Datapath: lw Memory Read

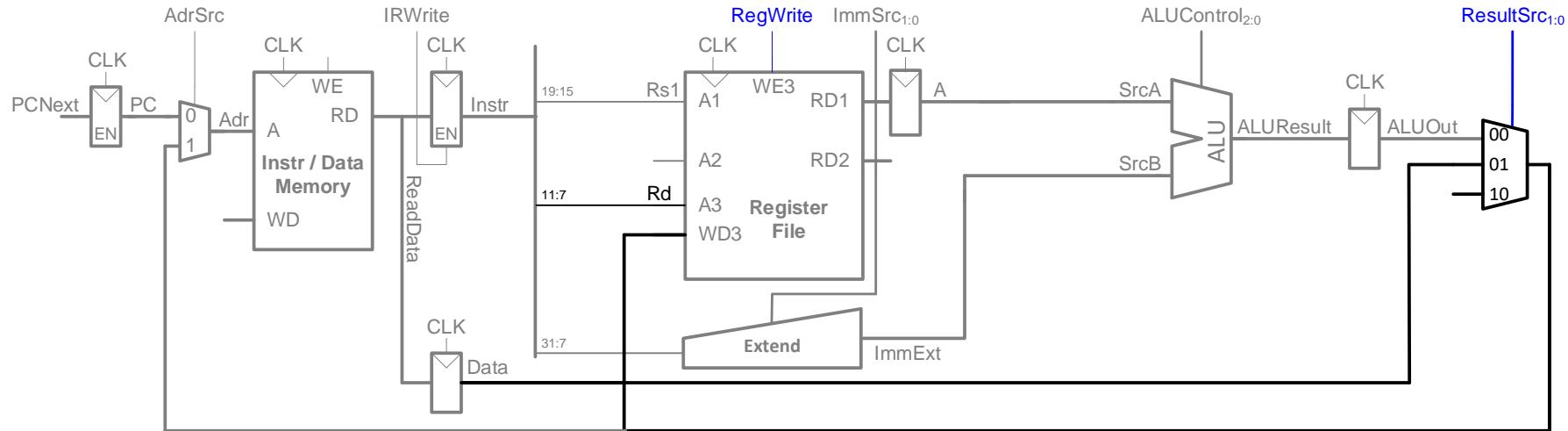
STEP 4: Read data from memory



- The fourth step is to load the data from the calculated address in the memory.
- We add a mux in front of the memory to choose the memory address, *Adr*, from either the *PC* or *ALUOut* based on the *AdrSrc* select signal.
- The data read from the memory is stored in another non-architectural register, *Data*.
- The address(*Adr*) multiplexer permits us to reuse the memory unit during the lw instruction.
 - In the first step, the address is taken from the *PC* to fetch the instruction.
 - In the fourth step, the address is taken from *ALUOut* to load the data.
 - Hence, *AdrSrc* must have different values during different steps of a single instruction.
 - We will develop a Finite-State-Machine (FSM) controller to generate these sequences of control signals.

Multicycle Datapath: 1w Write Register

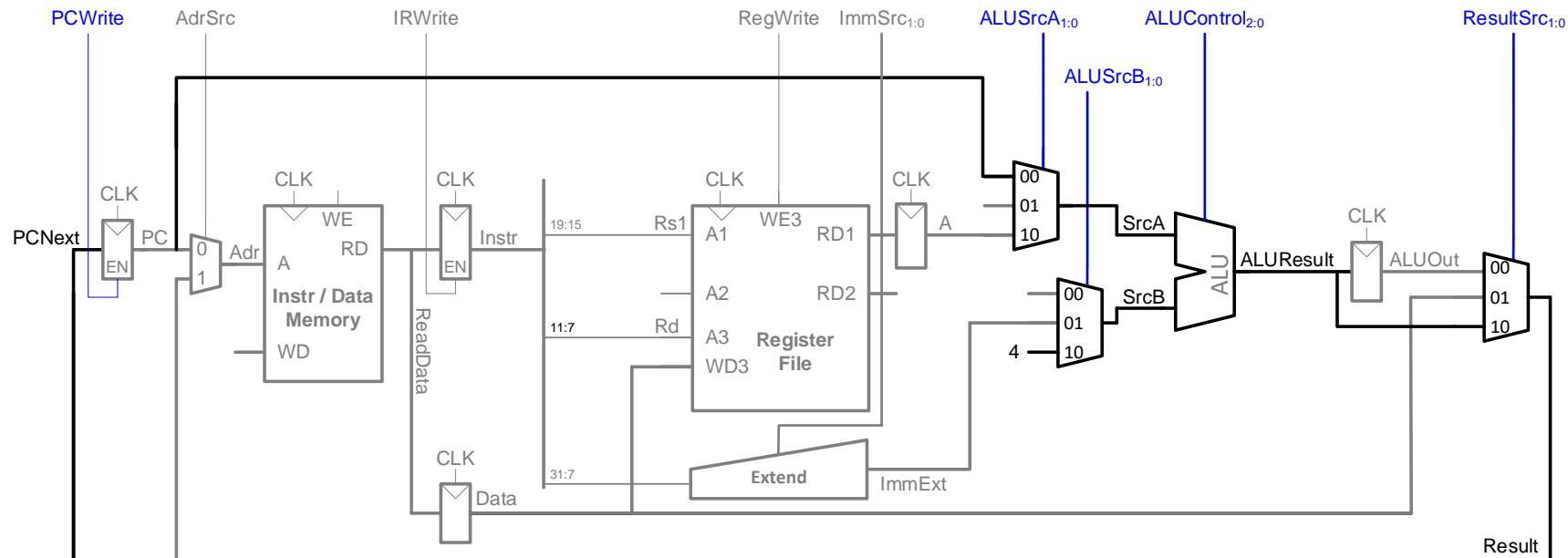
STEP 5: Write data back to register file



- Finally, the data is written back to the register file.
- The destination register is specified by the **rd** field of the instruction, $Instr_{11:7}$.
- The result comes from the non-architectural Data register.
 - Instead of connecting the Data register directly to the register file's **WD3** write port, we add a multiplexer on the Result bus to choose either **ALUOut** or **Data** before feeding Result back to the register file's write-data port (**WD3**).
 - This will be helpful because other instructions will need to write a result from the ALU to the register file.
 - The **RegWrite** signal is set to 1 to indicate that the register file should be updated.

Multicycle Datapath: Increment PC

STEP 6: Increment PC: $PC = PC + 4$



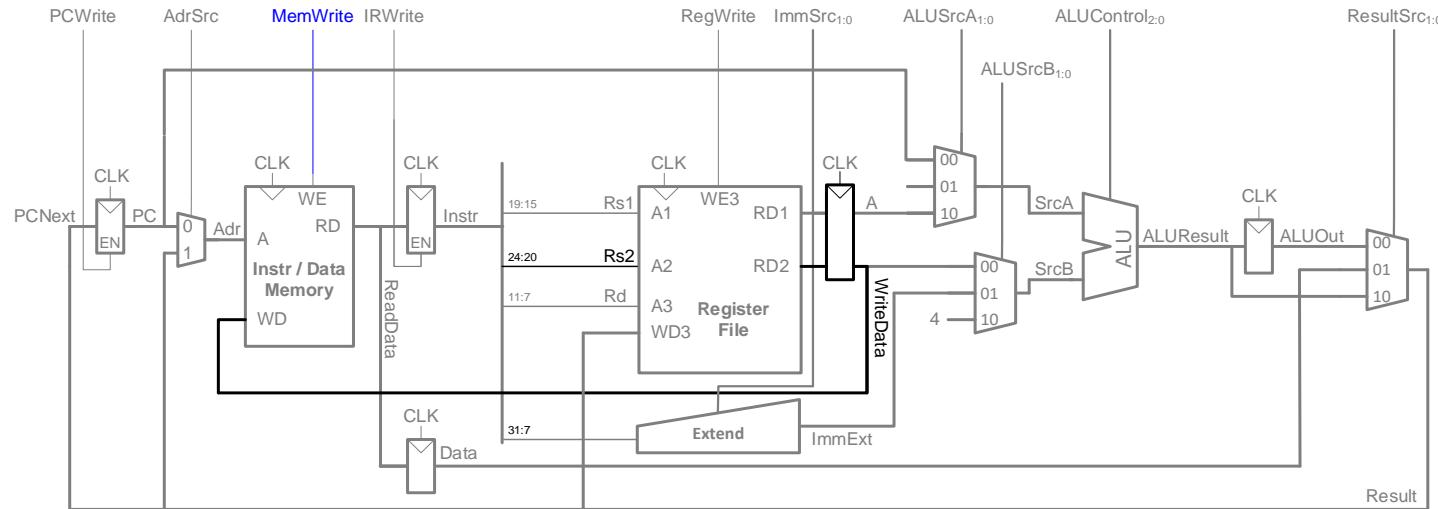
- While all this is happening, the processor must update the PC by adding 4 to it.
- We can use the existing ALU in the instruction **fetch step** (step 1) when it is not busy.
 - To do so, we must insert source multiplexers to choose PC and the constant 4 as ALU inputs.
 - A mux controlled by **ALUSrcA** chooses either **PC** or **A** as **SrcA**
 - A mux controlled by **ALUSrcB** chooses either **4** or **ImmExt** as **source B**.
 - A mux controlled by **ResultSrc** chooses the sum from the **ALUResult** ($PC + 4$) rather than **ALUOut**.
 - The **PCWrite** control signal enables the PC to be written only on certain cycles.

Chapter 7: Microarchitecture

Multicycle Datapath: Other Instructions

Multicycle Datapath: SW

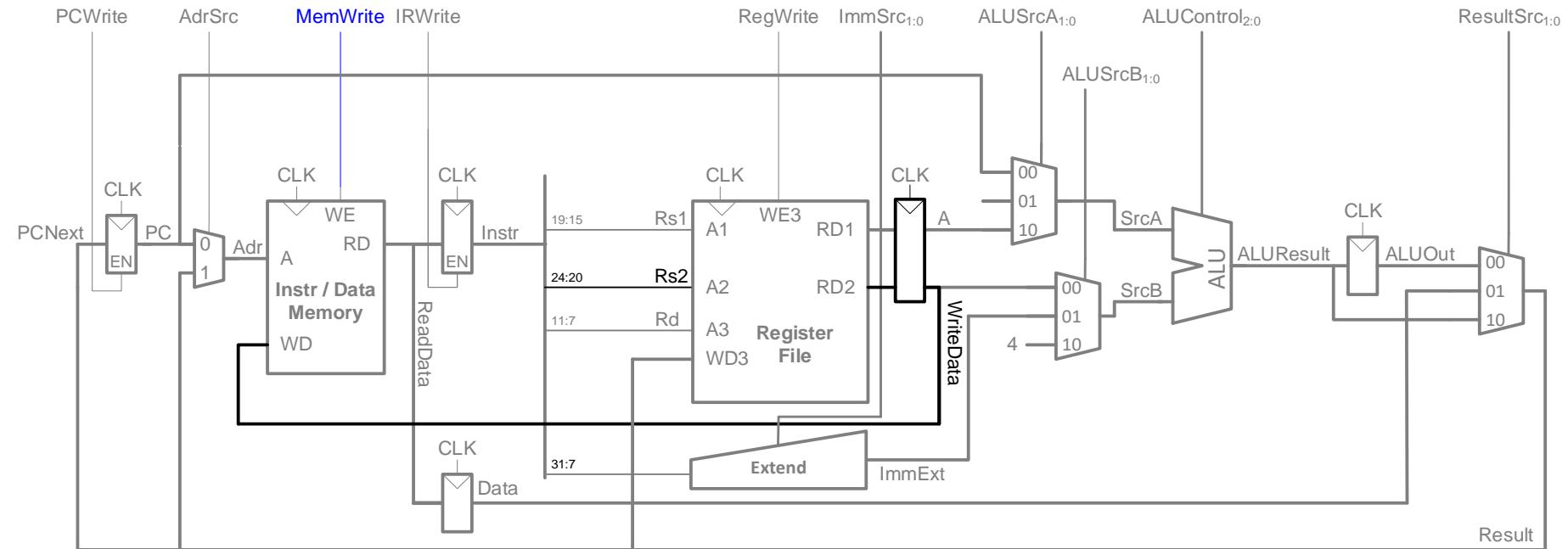
Write data in rs2 to memory



Like `lw`, `sw` reads a base address from port 1 of the register file and extends the immediate on the second step. Then, the ALU adds the base address to the immediate to find the memory address on the third step. The only new feature of `sw` is:

- Must read a second register from the register file and write its contents into memory.
- The register is specified in the *rs2* field of the instruction, $Instr_{24:20}$, which is connected to the second port of the register file (A2).
- After it is read on the second step, the register's contents are then stored in a non-architectural register, the *WriteData* register, just below the A register.
- It is then sent to the data memory's write data port (WD) to be written on the fourth step.
- The *MemWrite* control signal is asserted to enable memory write.

Multicycle Datapath: R-Type Instructions

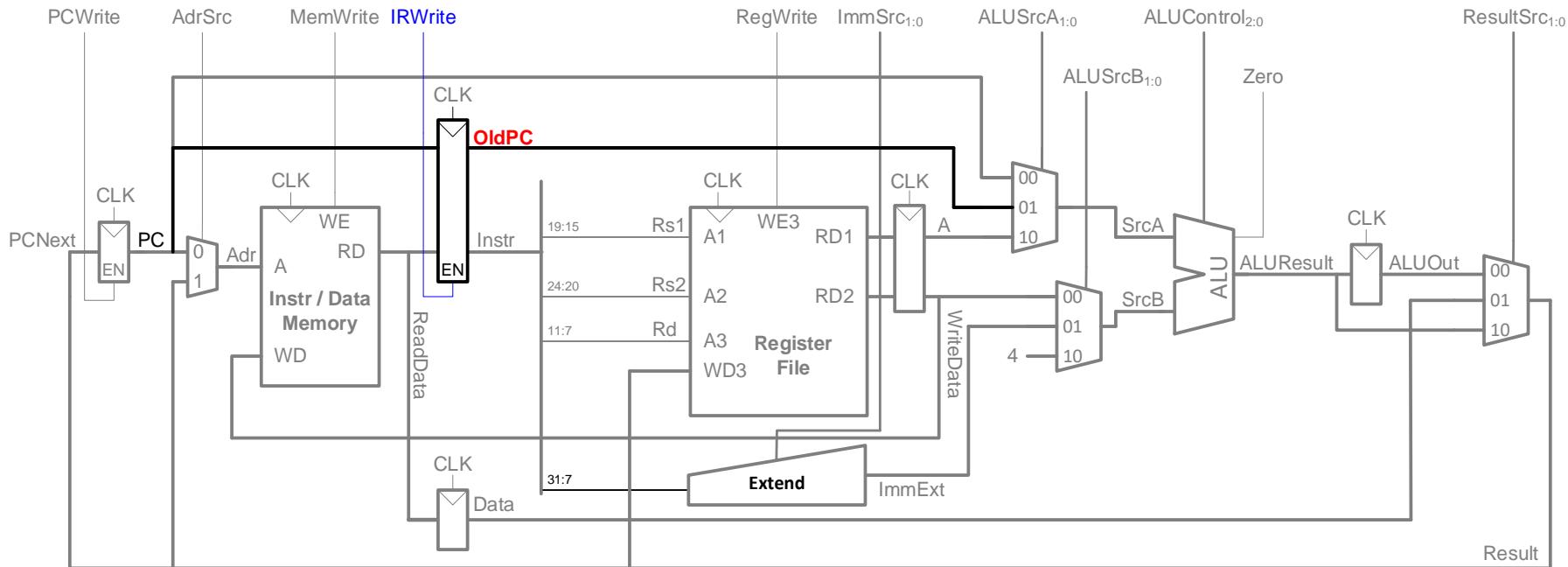


- R-type instructions operate on two source registers and write the result back to the register file.
- The datapath already contains all the connections necessary for these steps.

Multicycle Datapath: beq

Calculate branch target address:
 $BTA = PC + imm$

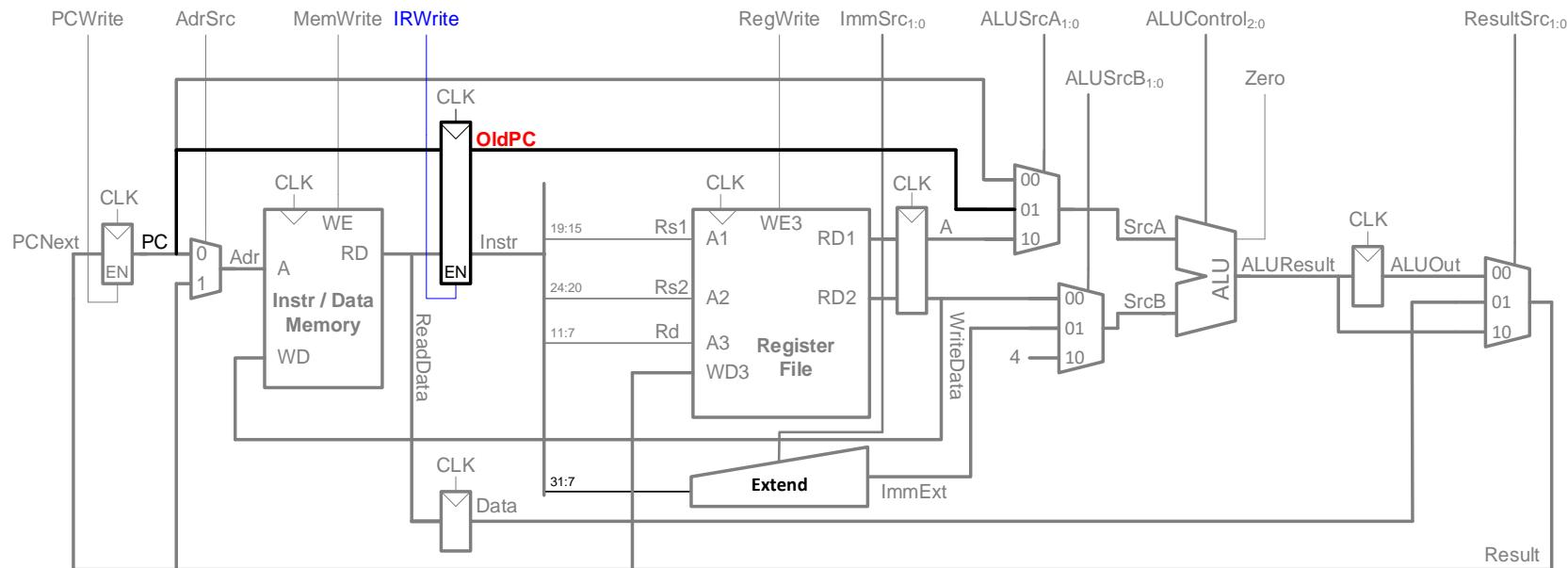
- beq checks whether two register contents are equal and compute the branch target address by adding the current PC to a 13-bit signed branch offset.
- The hardware to compare the registers using subtraction is already present in the datapath.



PC is updated in Fetch stage, so need to save **old (current) PC**

Multicycle Datapath: beq

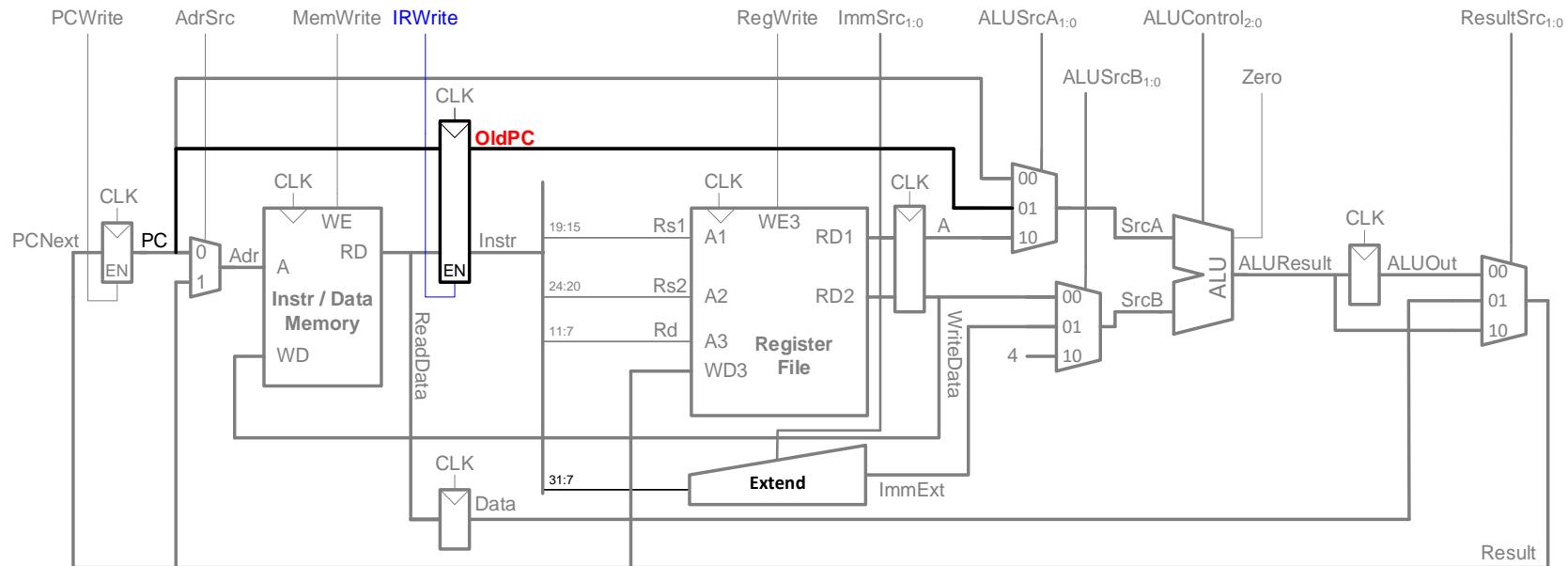
- Calculate branch target address: $\text{BTA} = \text{PC} + \text{imm}$
- PC is updated in the Fetch stage (i.e. step 1), so need to save **old (current) PC**



- Step 1: instruction fetched; PC updated to $\text{PC}+4$; the PC of the current instruction, OldPC , must be stored in a non-architectural register.
- Step 2: the source registers are fetched, ALU calculates BTA using OldPC for SrcA and ImmExt for SrcB , set $\text{ALUControl} = 000$ to perform addition, $\text{BTA} = \text{OldPC} + \text{immExt}$; stores this sum (**BTA**) in the ALUOut register.
- Step 3: ALU subtracts the two source registers and asserts the `Zero` flag output if the ALUResult is zero.
 - If the `Zero` flag is set, the control unit asserts `PCWrite` and the Result multiplexer selects ALUOut (that contains the BTA) to feed to the PC. No new hardware is needed.

Multicycle Datapath: beq

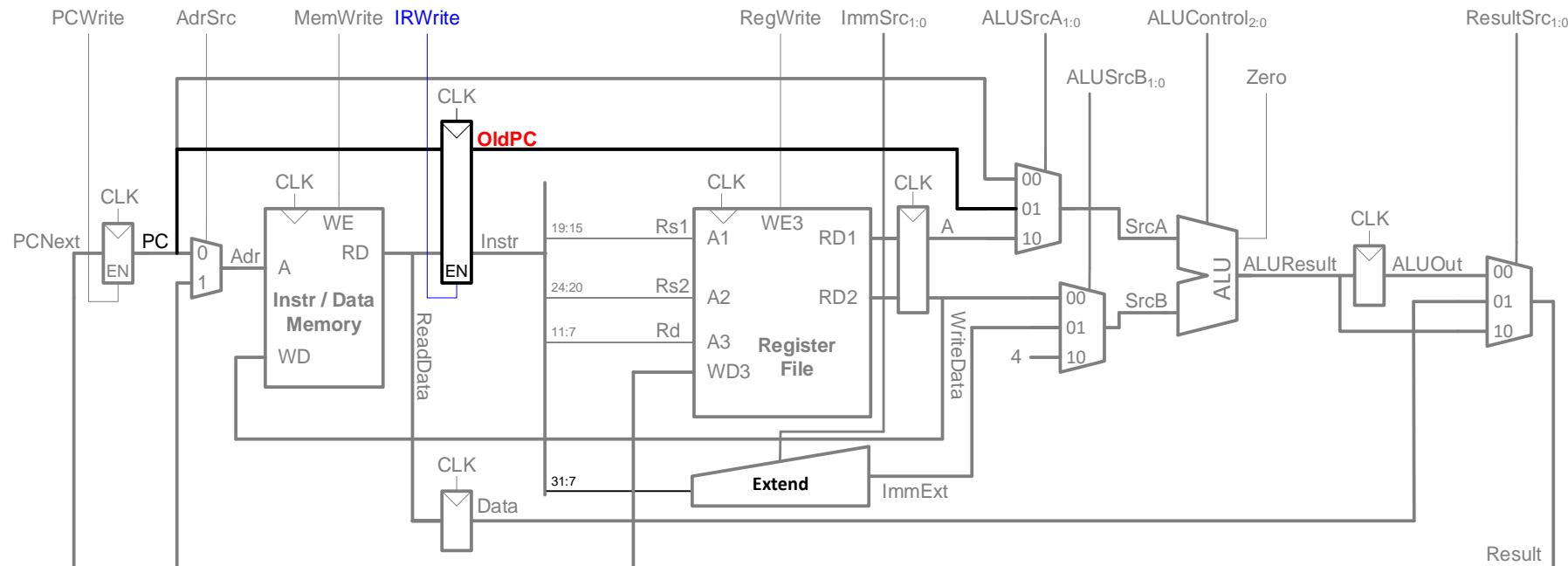
- Calculate branch target address: $\text{BTA} = \text{PC} + \text{imm}$
- PC is updated in the Fetch stage (i.e. step 1), so need to save **old (current) PC**



- The design of the multicycle processor is much like that of the single-cycle processor in that hardware is systematically connected between the state elements to handle each instruction.
- The main difference is that the instruction is executed in several steps.
- Non-architectural registers are inserted to hold the results of each step.
- Memory can be shared for instructions and data.
- ALU can be reused several times, reducing hardware costs.

Multicycle Datapath: beq

Calculate branch target address:
 $BTA = PC + imm$



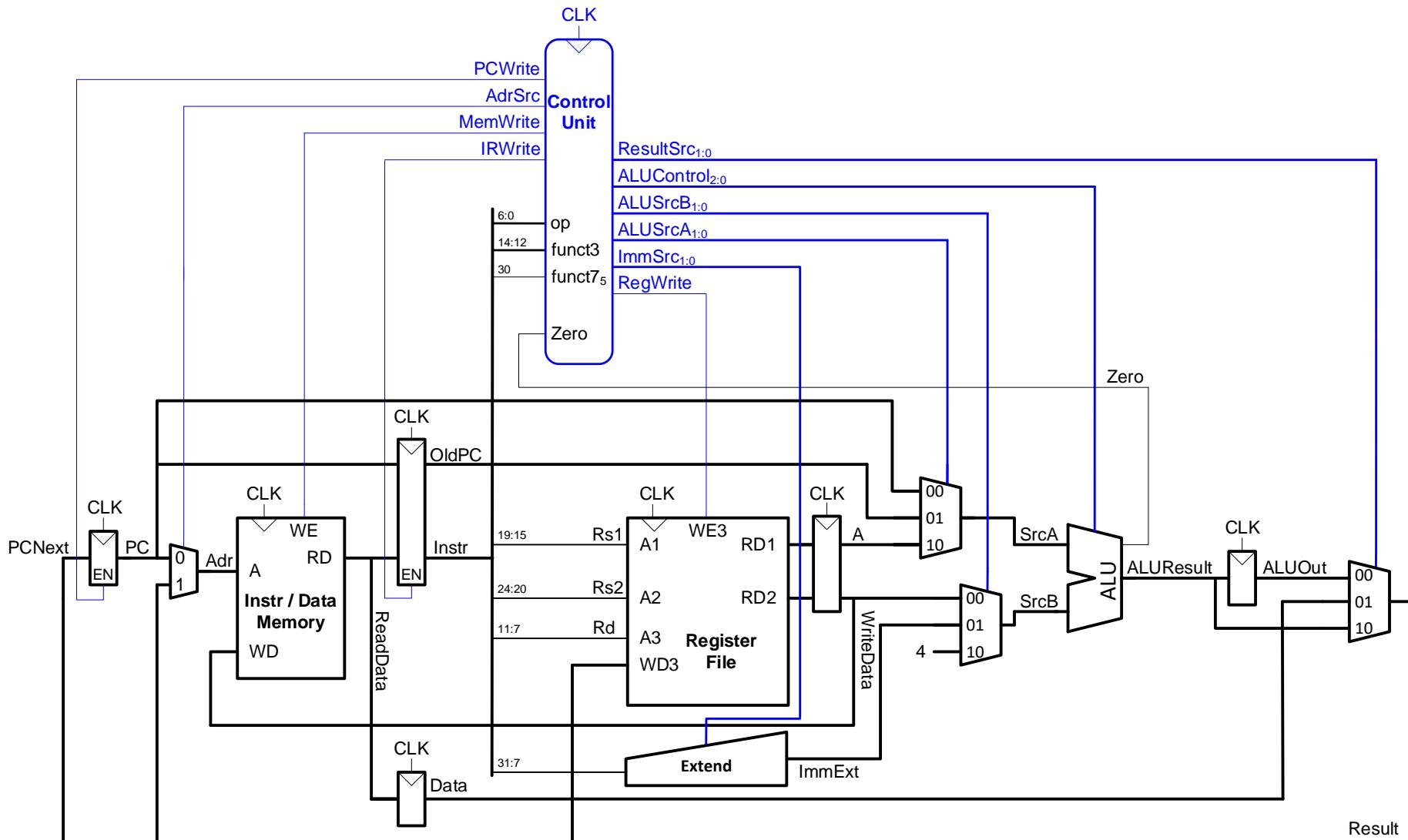
PC is updated in Fetch stage, so need to save **old (current) PC**

Multicycle Datapath

Example Program:

Address	Instruction	Type	Fields						Machine Language	
0x1000	L7: lw x6, -4(x9)	I	imm_{11:0} 111111111100	rs1 01001	f3 010	rd 00110	op 0000011	FFC4A303		
0x1004	sw x6, 8(x9)	S	imm_{11:5} 0000000	rs2 00110	rs1 01001	f3 010	imm_{4:0} 01000	op 0100011	0064A423	
0x1008	or x4, x5, x6	R	funct7 0000000	rs2 00110	rs1 00101	f3 110	rd 00100	op 0110011	0062E233	
0x100C	beq x4, x4, L7	B	imm_{12,10:5} 1111111	rs2 00100	rs1 00100	f3 000	imm_{4:1,11} 10101	op 1100011	FE420AE3	

Multicycle RISC-V Processor

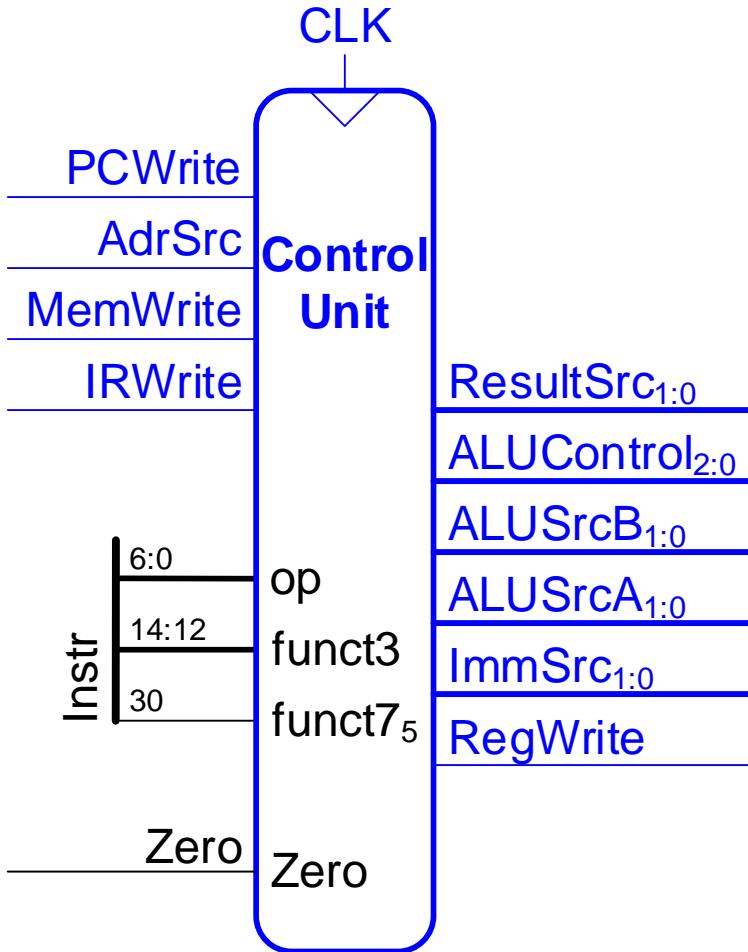


Chapter 7: Microarchitecture

Multicycle Control

Multicycle Control

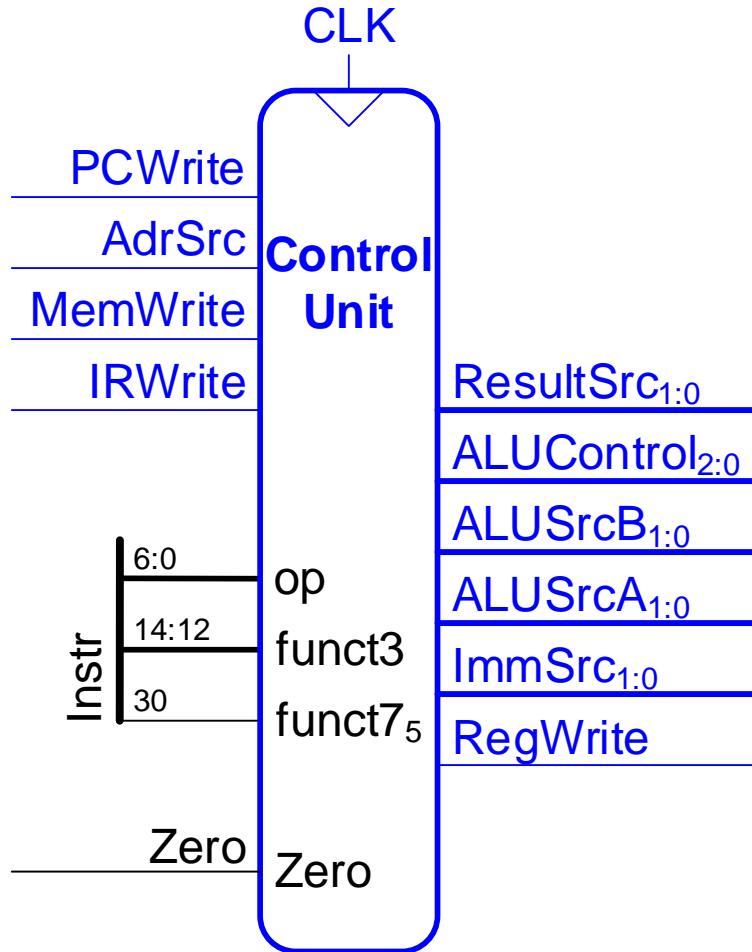
High-Level View



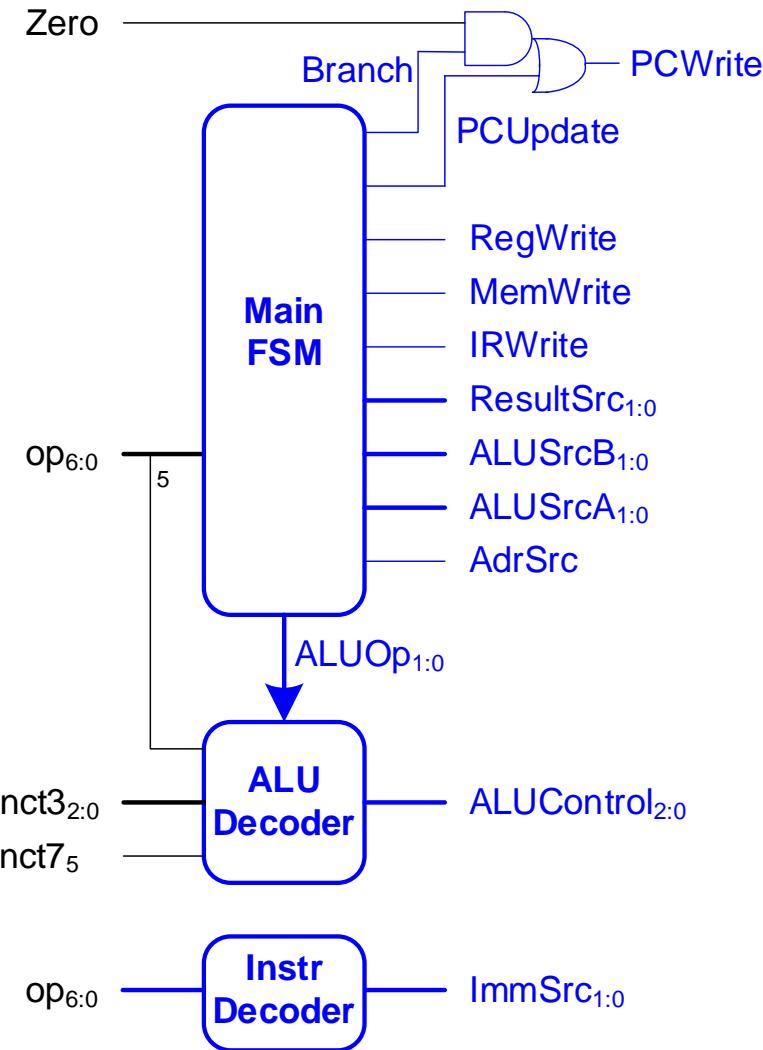
- As in the single-cycle processor, the control unit computes the control signals based on
 - op ($Instr_{6:0}$)
 - Funct3 ($Instr_{14:12}$), and
 - funct7₅ ($Instr_{30}$)

Multicycle Control

High-Level View

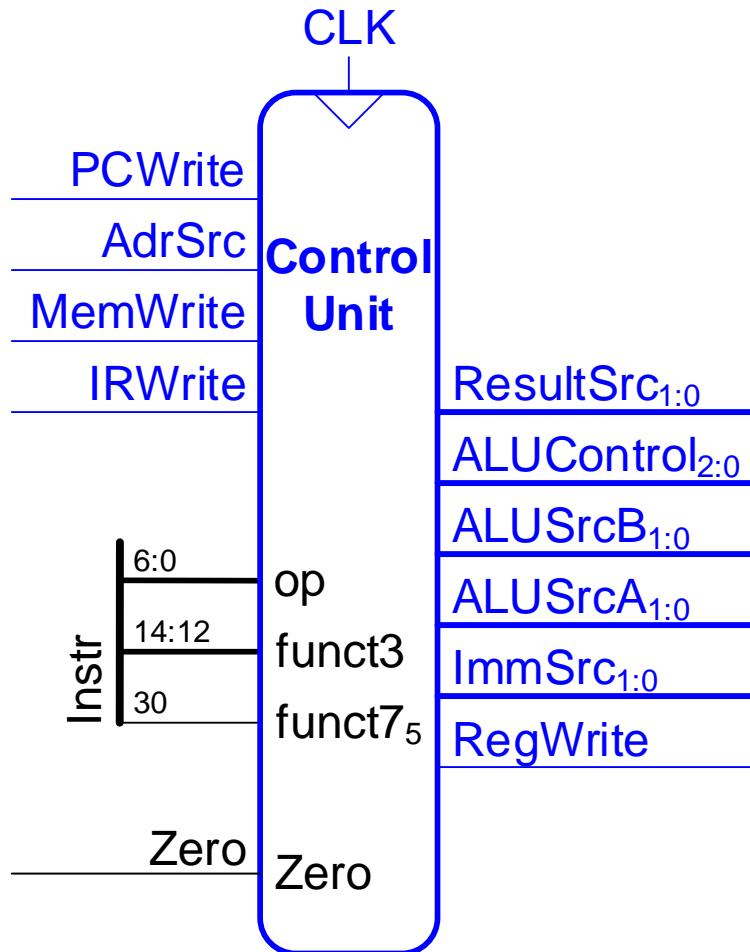


Low-Level View

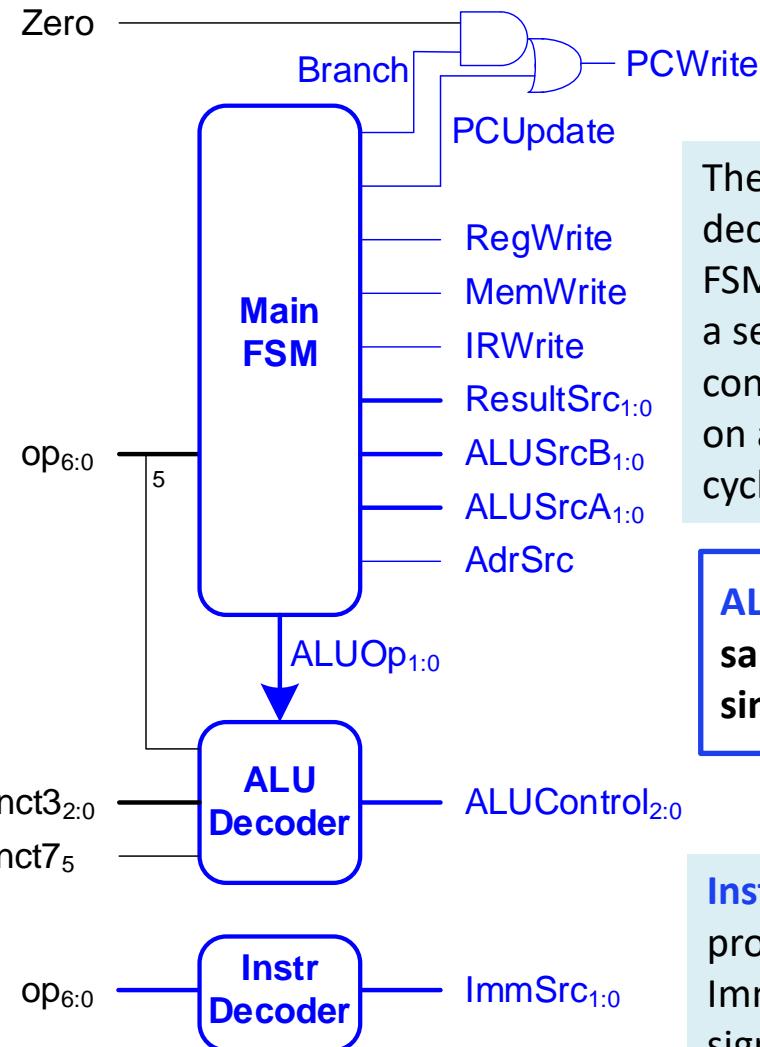


Multicycle Control

High-Level View



Low-Level View

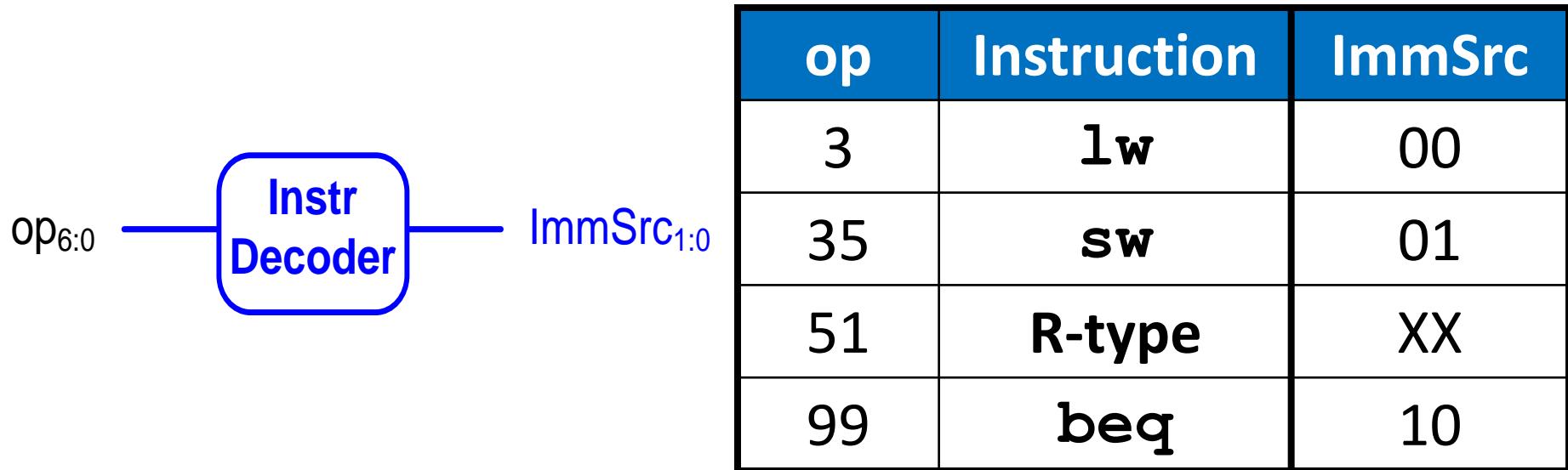


The main decoder is an FSM to produce a sequence of control signals on appropriate cycles.

ALU Decoder
same as single-cycle

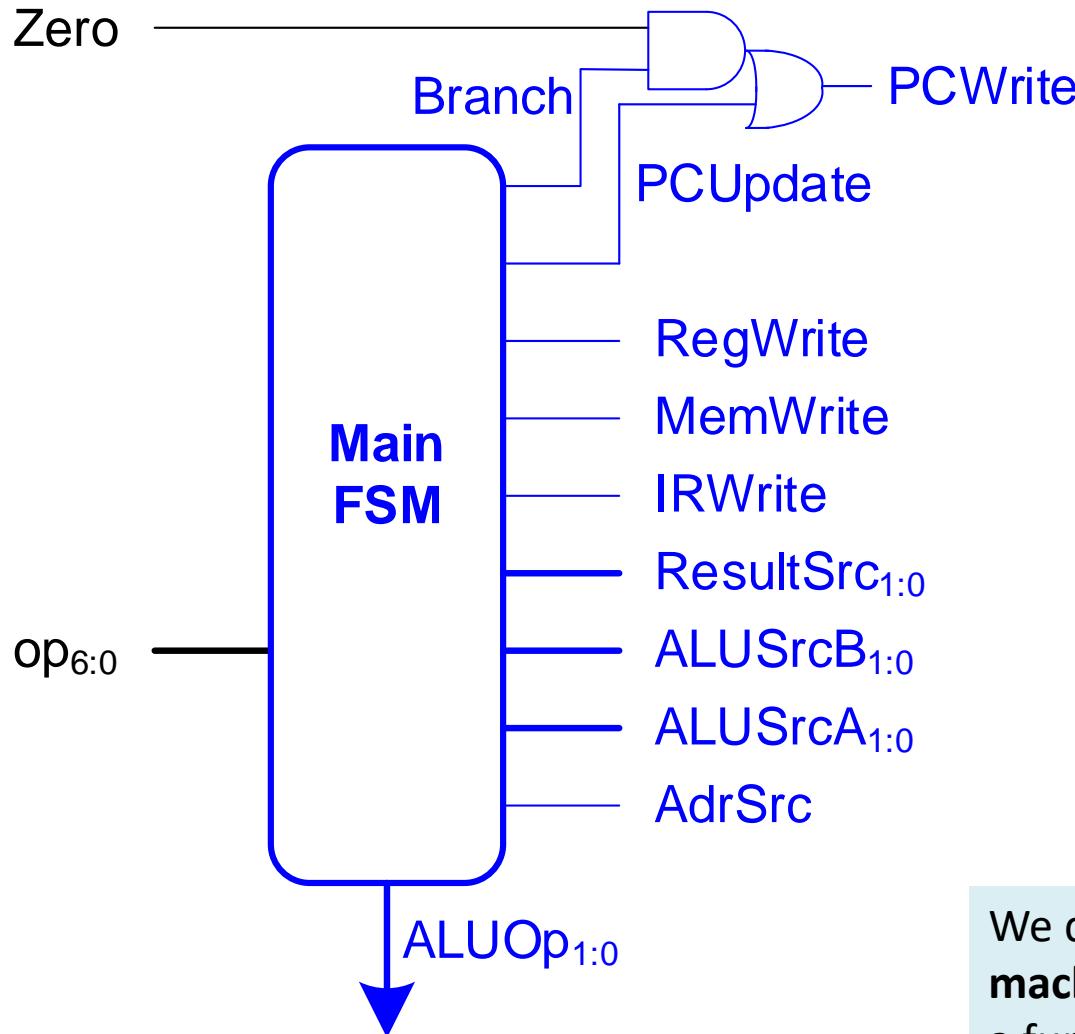
Instr Decoder
produces the ImmSrc select signal

Multicycle Control: Instruction Decoder



ImmSrc _{1:0}	ImmExt	Instruction Type
00	$\{{20\{instr[31]\}}, \text{instr[31:20]}$	I-Type
01	$\{{20\{instr[31]\}}, \text{instr[31:25]}, \text{instr[11:7]}$	S-Type
10	$\{{19\{instr[31]\}}, \text{instr[31]}, \text{instr[7]}, \text{instr[30:25]}, \text{instr[11:8]}, 1'b0\}$	B-Type

Multicycle Control: Main FSM

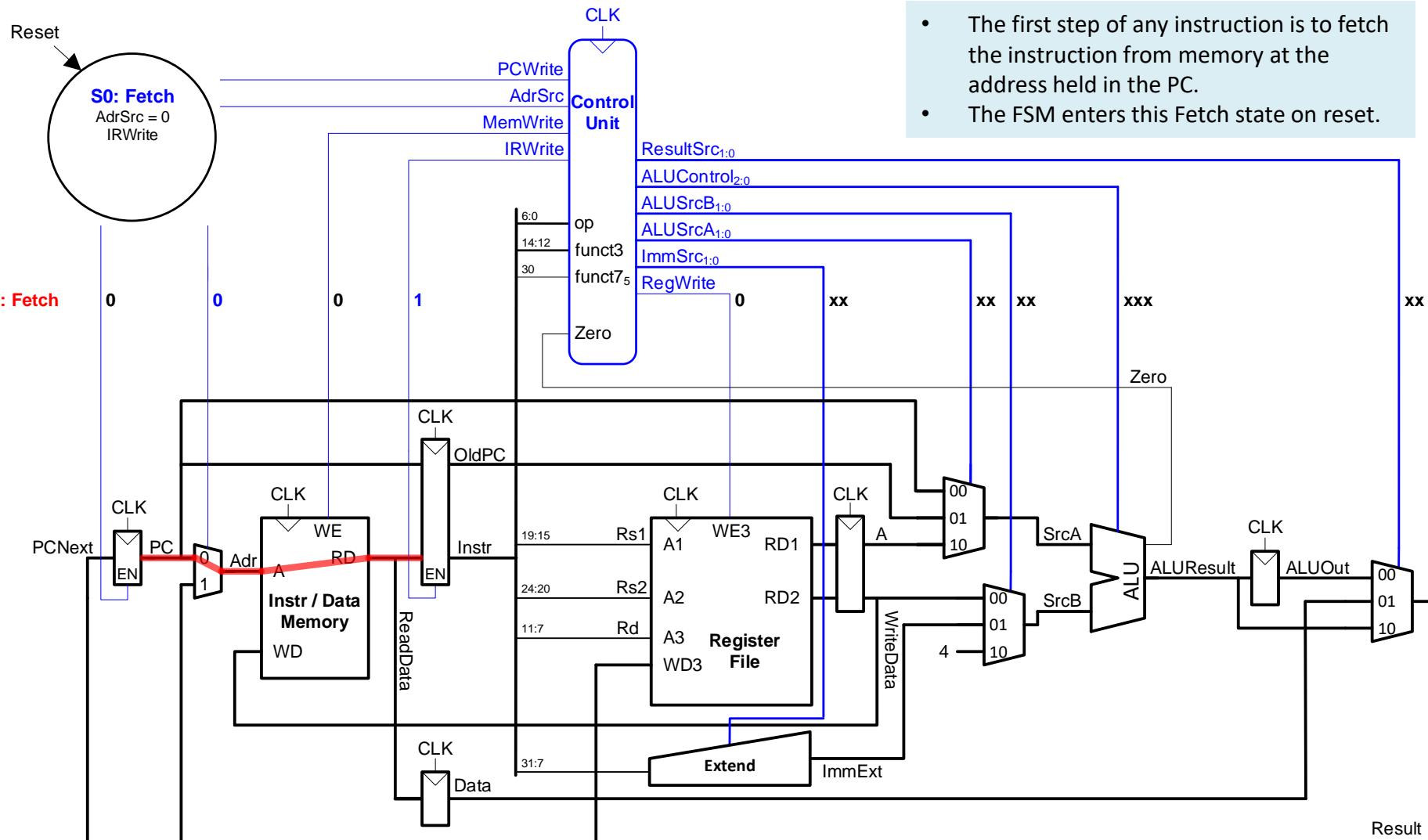


To declutter FSM:

- **Write enable signals** (`RegWrite`, `MemWrite`, `IRWrite`, `Pupate`, and `Branch`) are **0** if not listed in a state.
- **Other signals are don't care** if not listed in a state

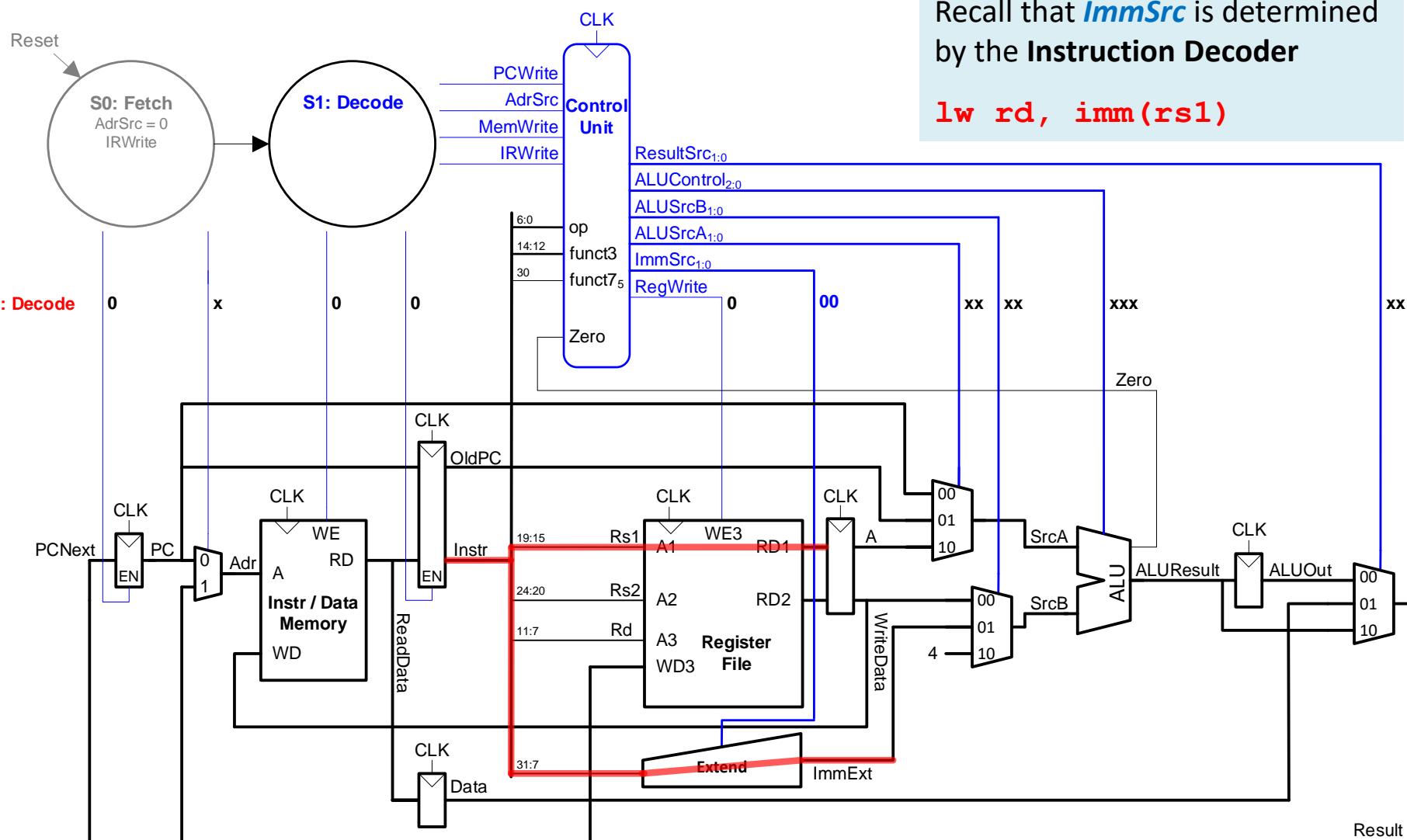
We design the Main FSM as a **Moore machine** so that the outputs are only a function of the current state.

Main FSM: Fetch

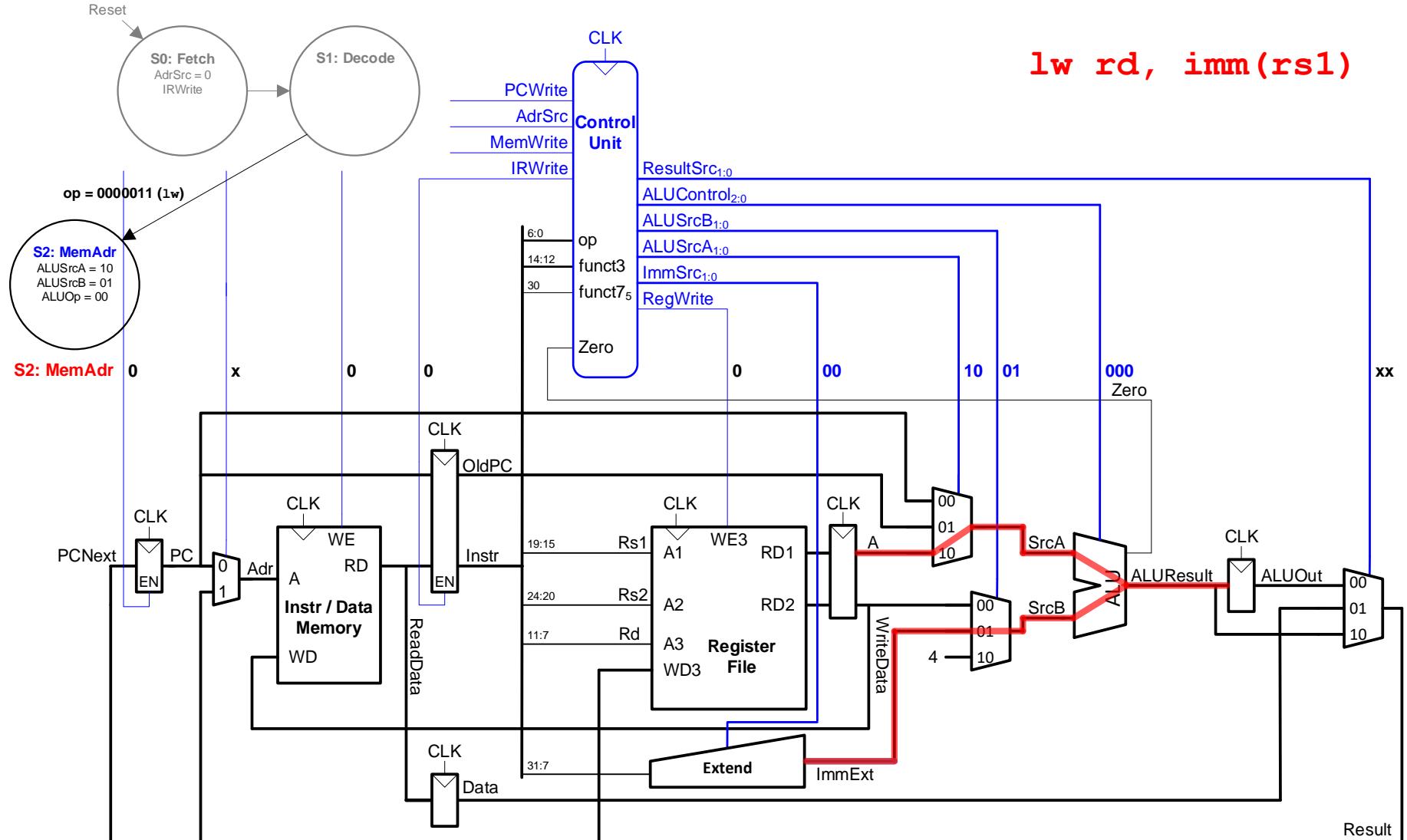


- The first step of any instruction is to fetch the instruction from memory at the address held in the PC.
- The FSM enters this Fetch state on reset.

Main FSM: Decode

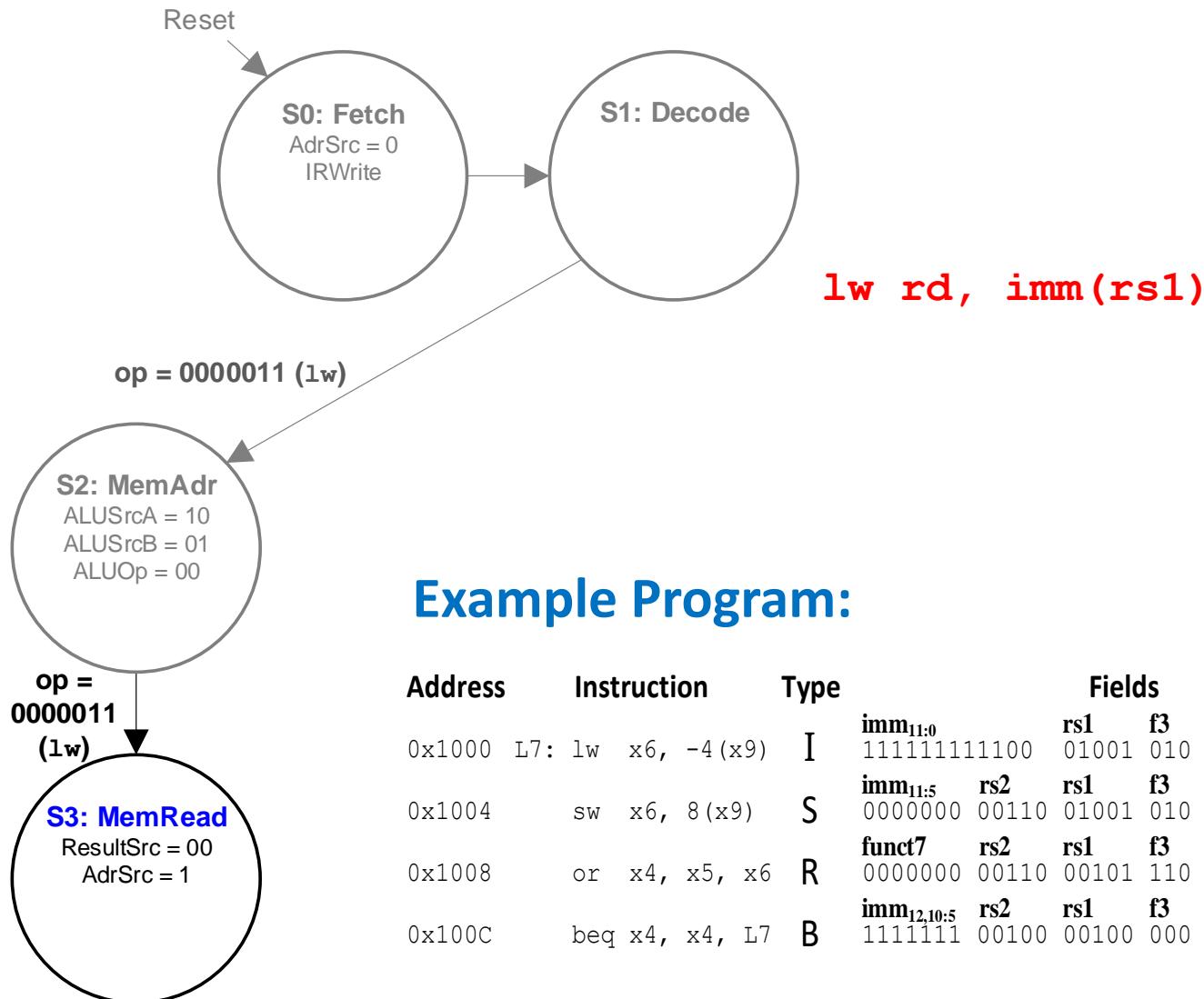


Main FSM: Address

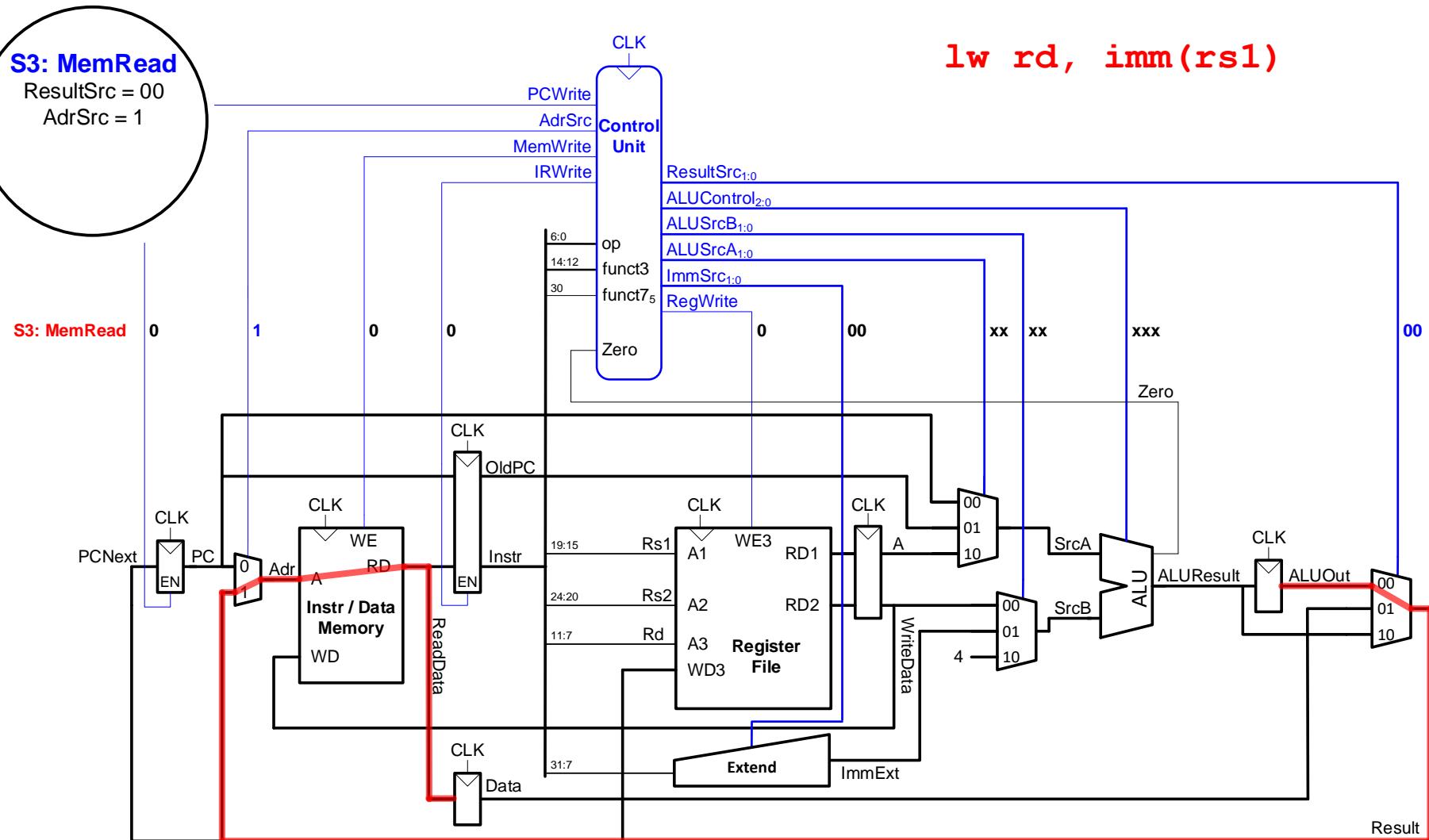


lw rd, imm(rs1)

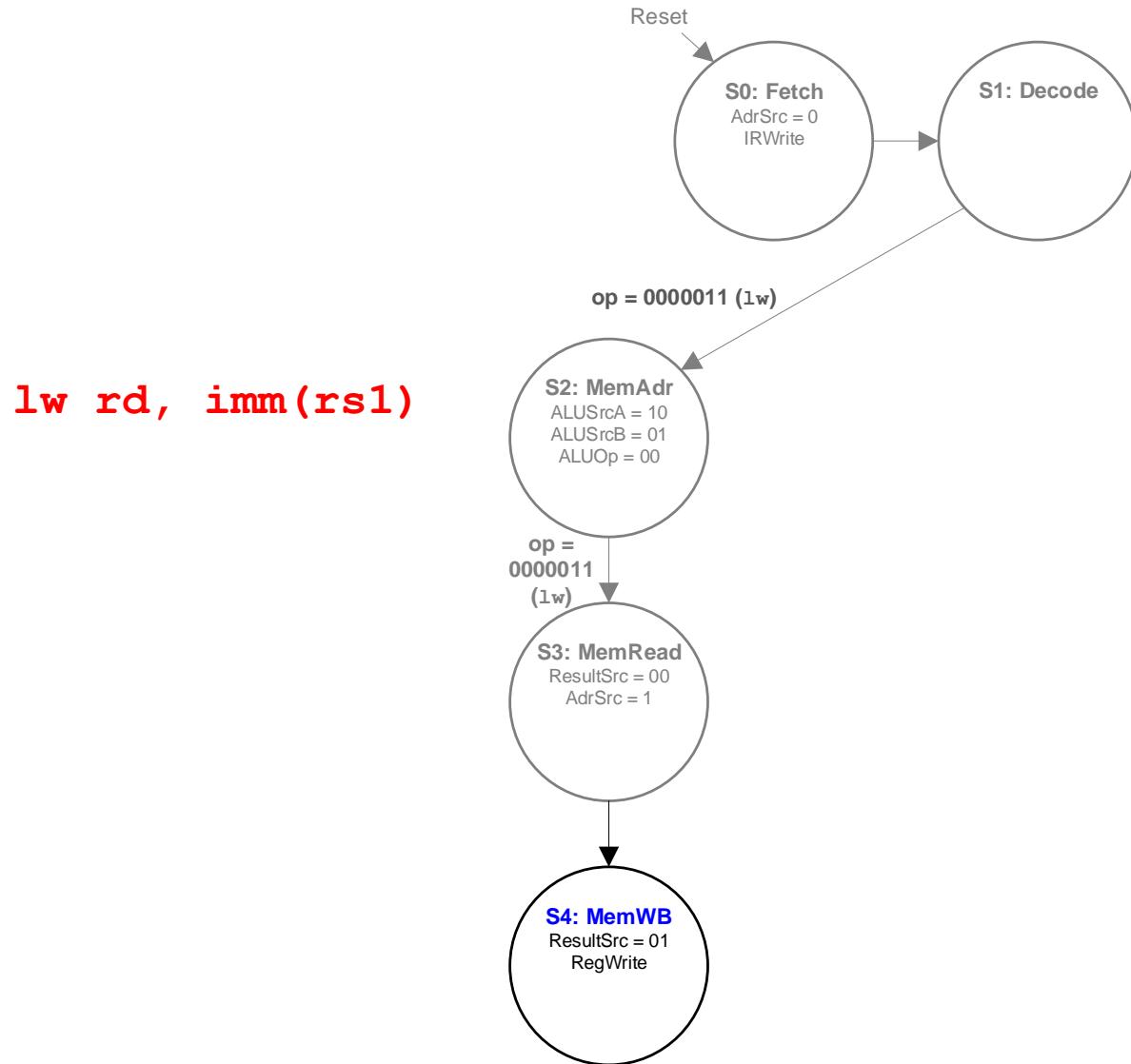
Main FSM: Read Memory



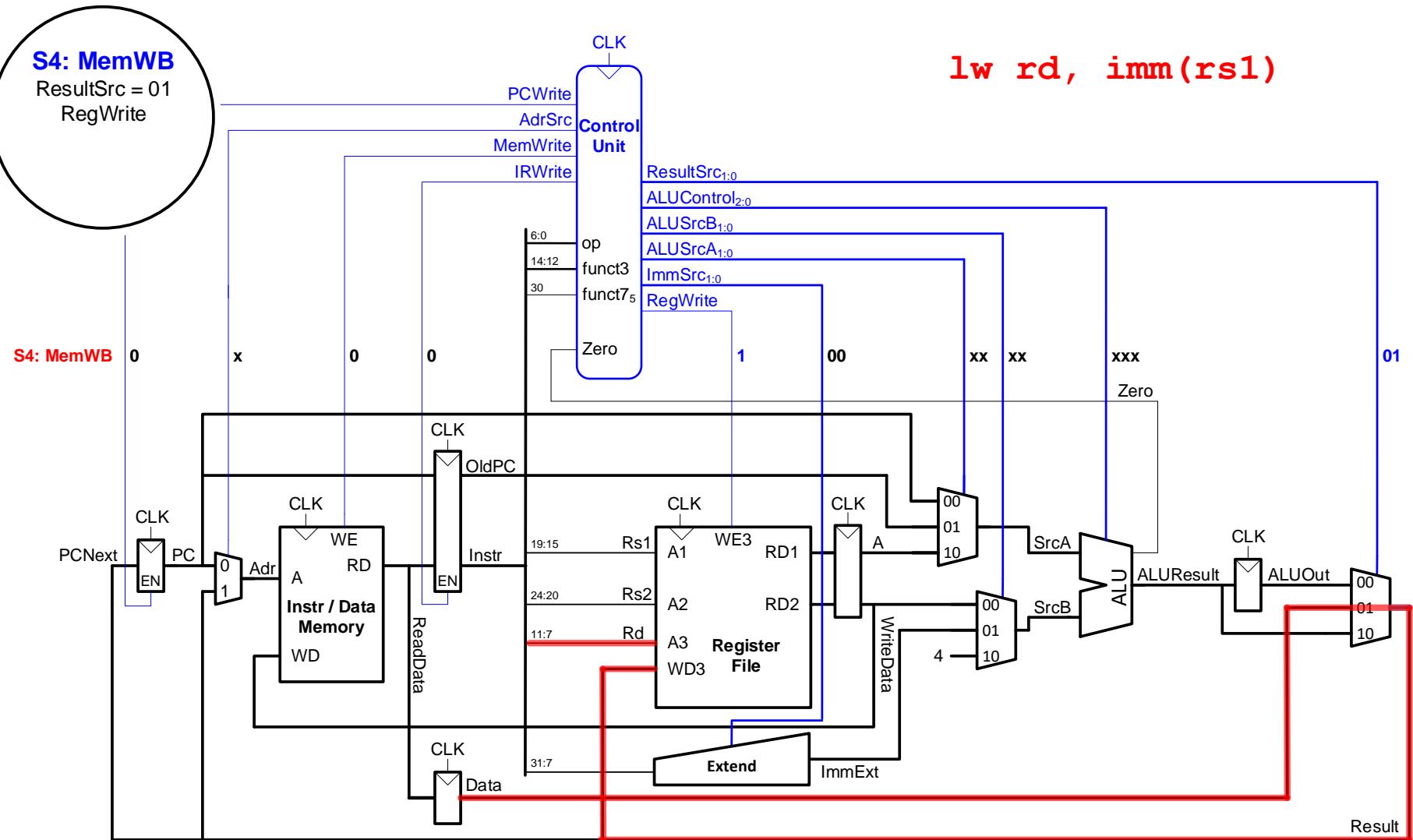
Main FSM: Read Memory Datapath



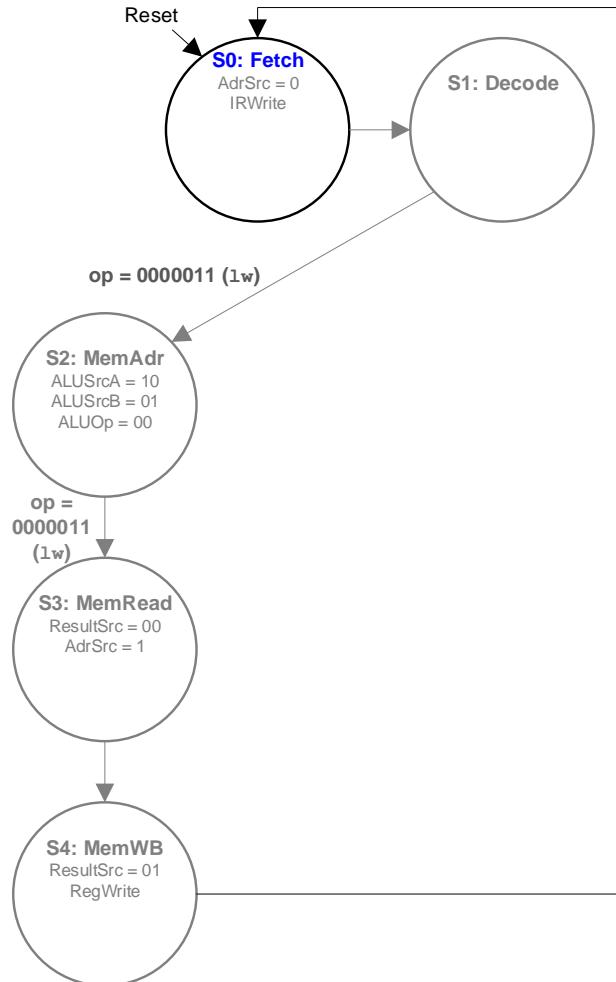
Main FSM: Write RF



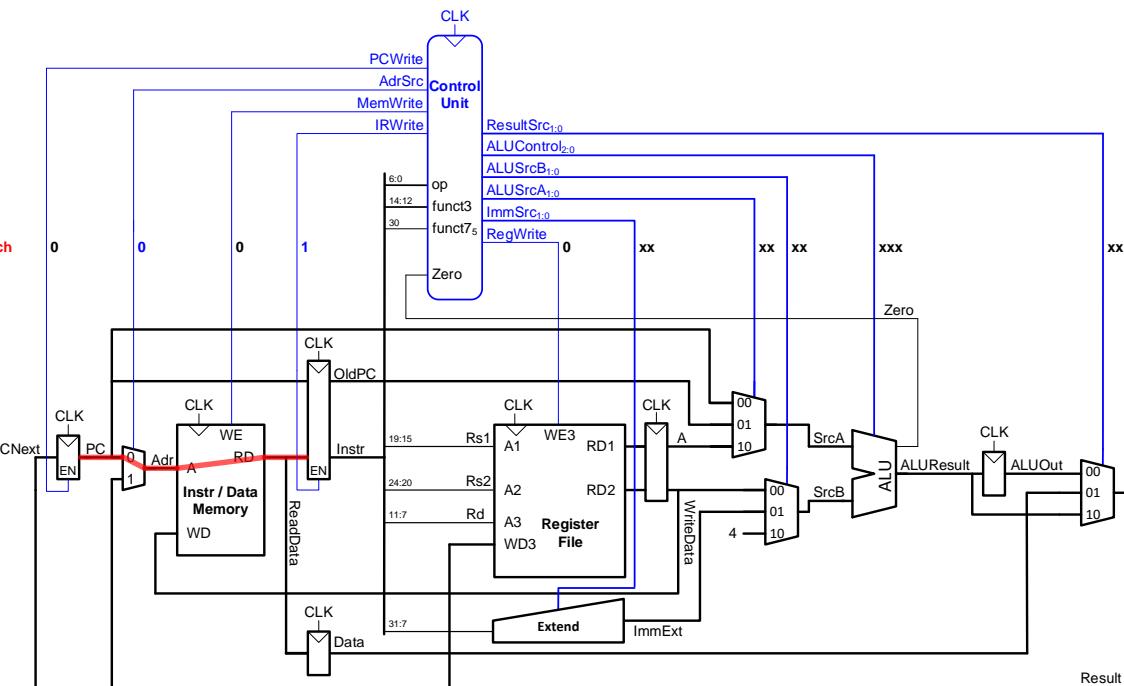
Main FSM: Write RF Datapath



Main FSM: Fetch Revisited

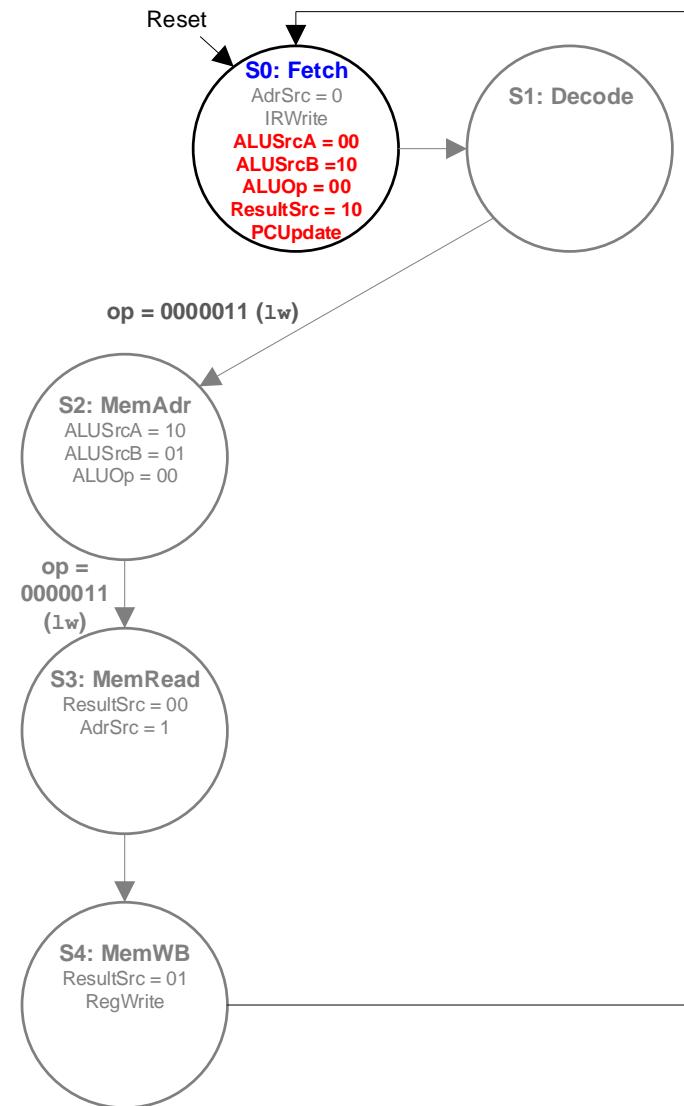


- ALU is not used
- We can use the ALU to do PC+4 to update PC to point to the next instruction.

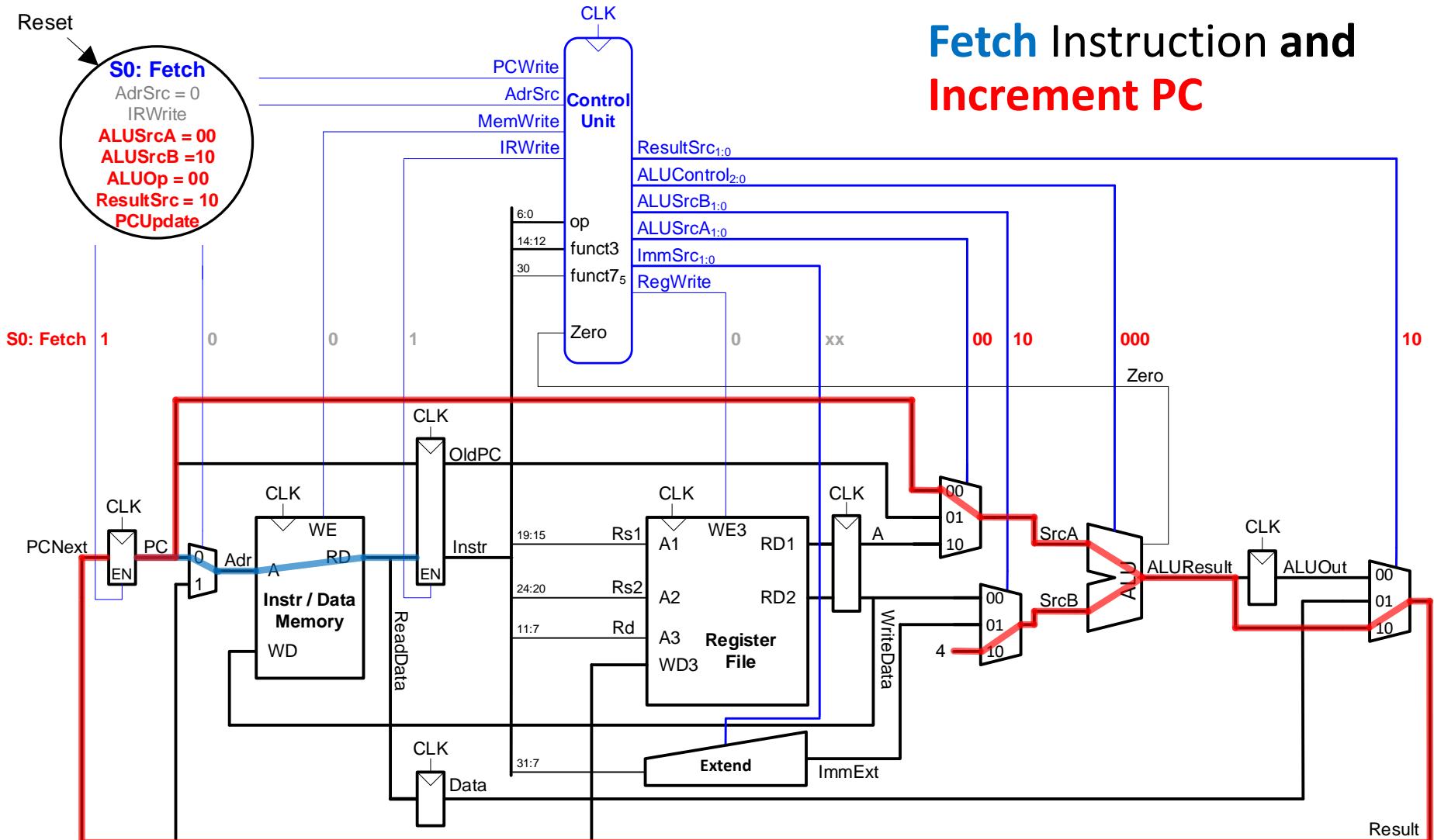


Main FSM: Fetch Revisited

Calculate **PC+4**
during Fetch stage
(ALU isn't being
used)



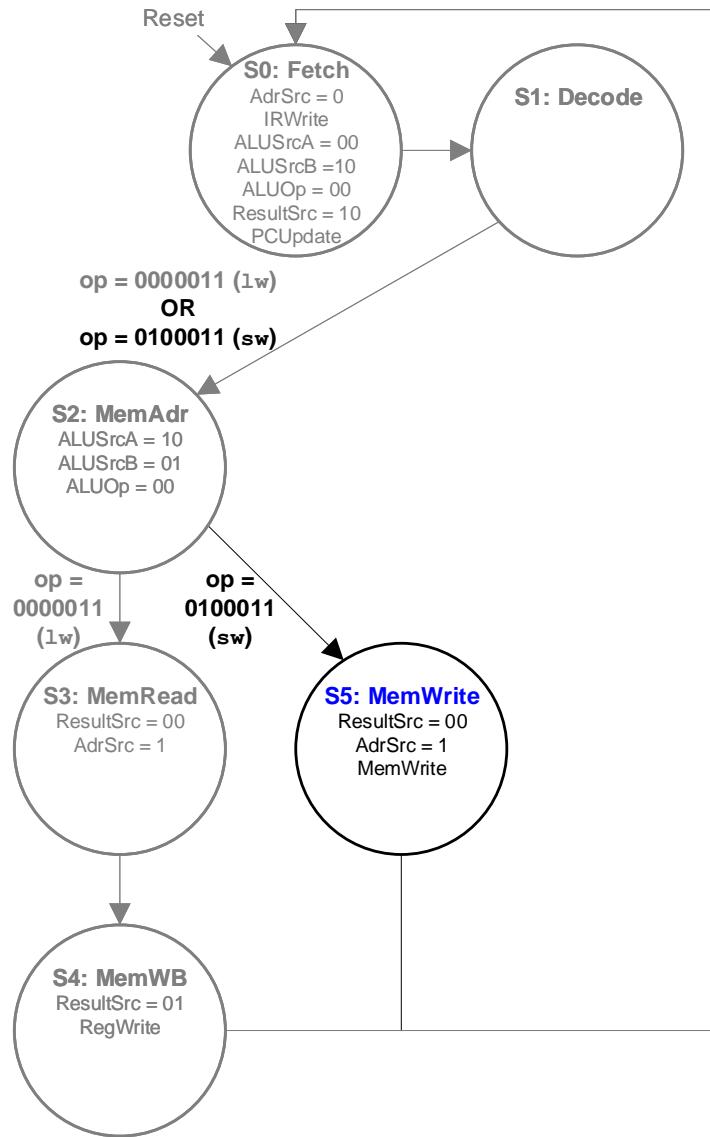
Main FSM: Fetch (PC+4) Datapath



Chapter 7: Microarchitecture

Multicycle Control: Other Instructions

Main FSM: SW



sw rs2, imm(rs1)

sw source_register, offset(base_register)

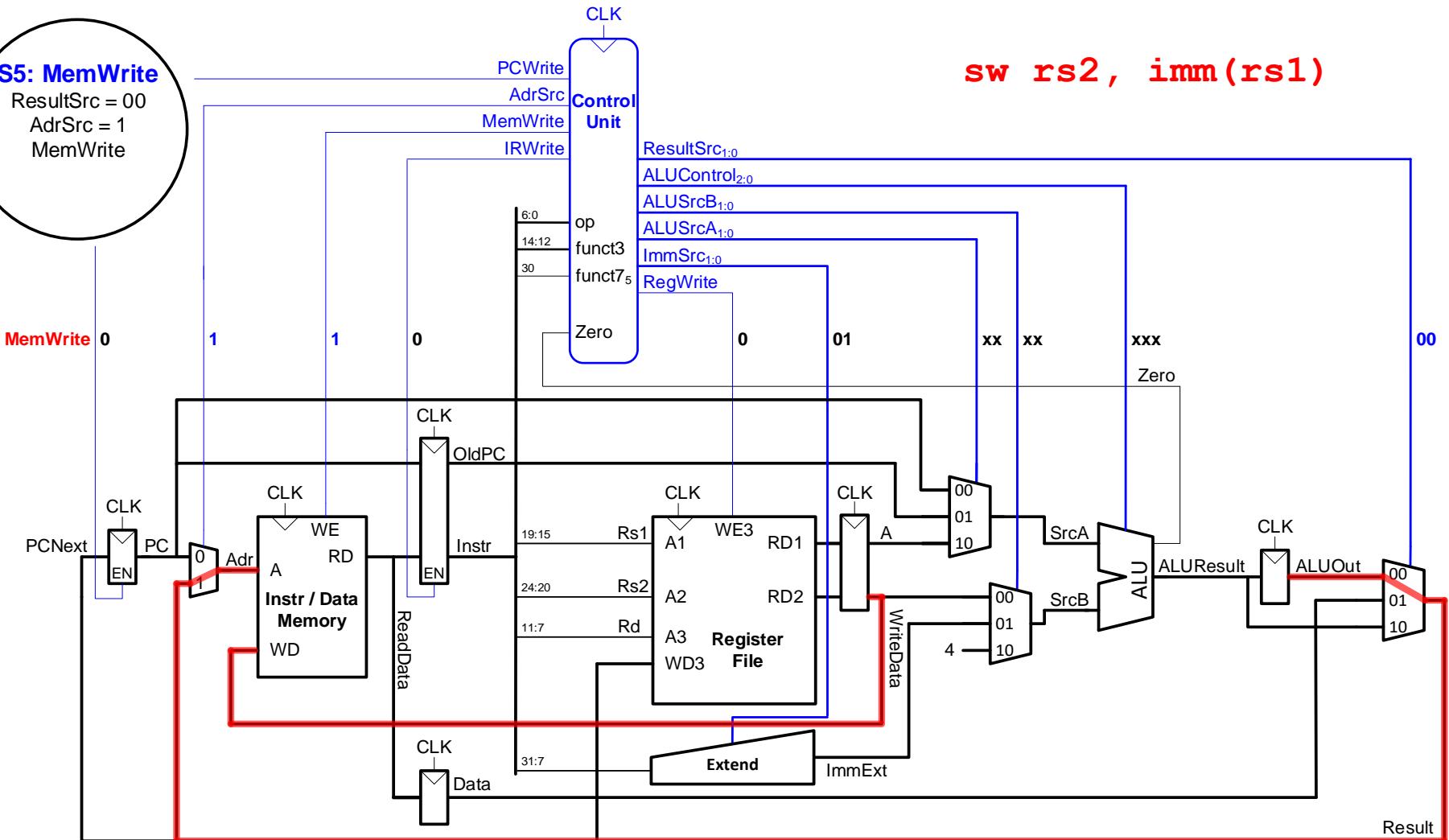
Example Program:

Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4(x9)	I	imm _{11:0} rs1 f3 rd	op 0000011 FFC4A303
0x1004	sw x6, 8(x9)	S	imm _{11:5} rs2 rs1 f3 imm _{4:0}	op 0100011 0064A423
0x1008	or x4, x5, x6	R	funct7 rs2 rs1 f3 rd	op 0110011 0062E233
0x100C	beq x4, x4, l7	B	imm _{12:5} rs2 rs1 f3 imm _{4:1,11}	op 1100011 FE420AE3

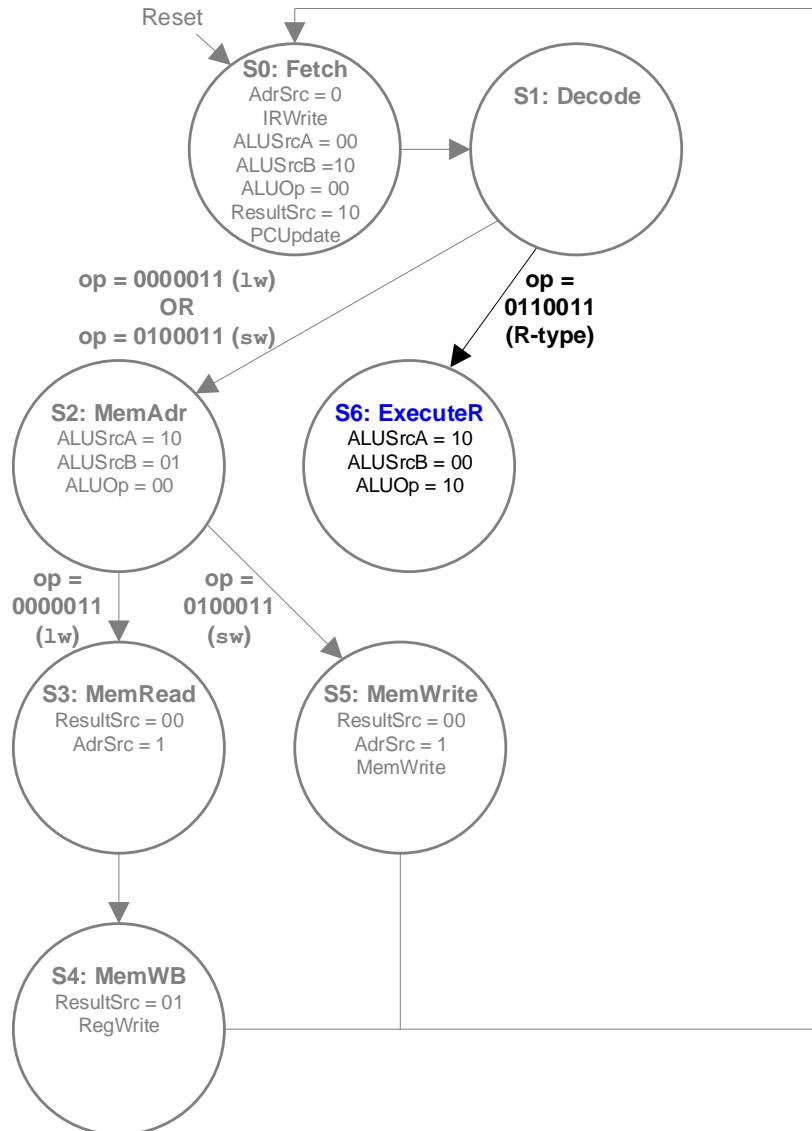
Main FSM: sw Datapath

S5: MemWrite
 ResultSrc = 00
 AdrSrc = 1
 MemWrite

sw rs2, imm(rs1)



Main FSM: R-Type Execute



Example Program:

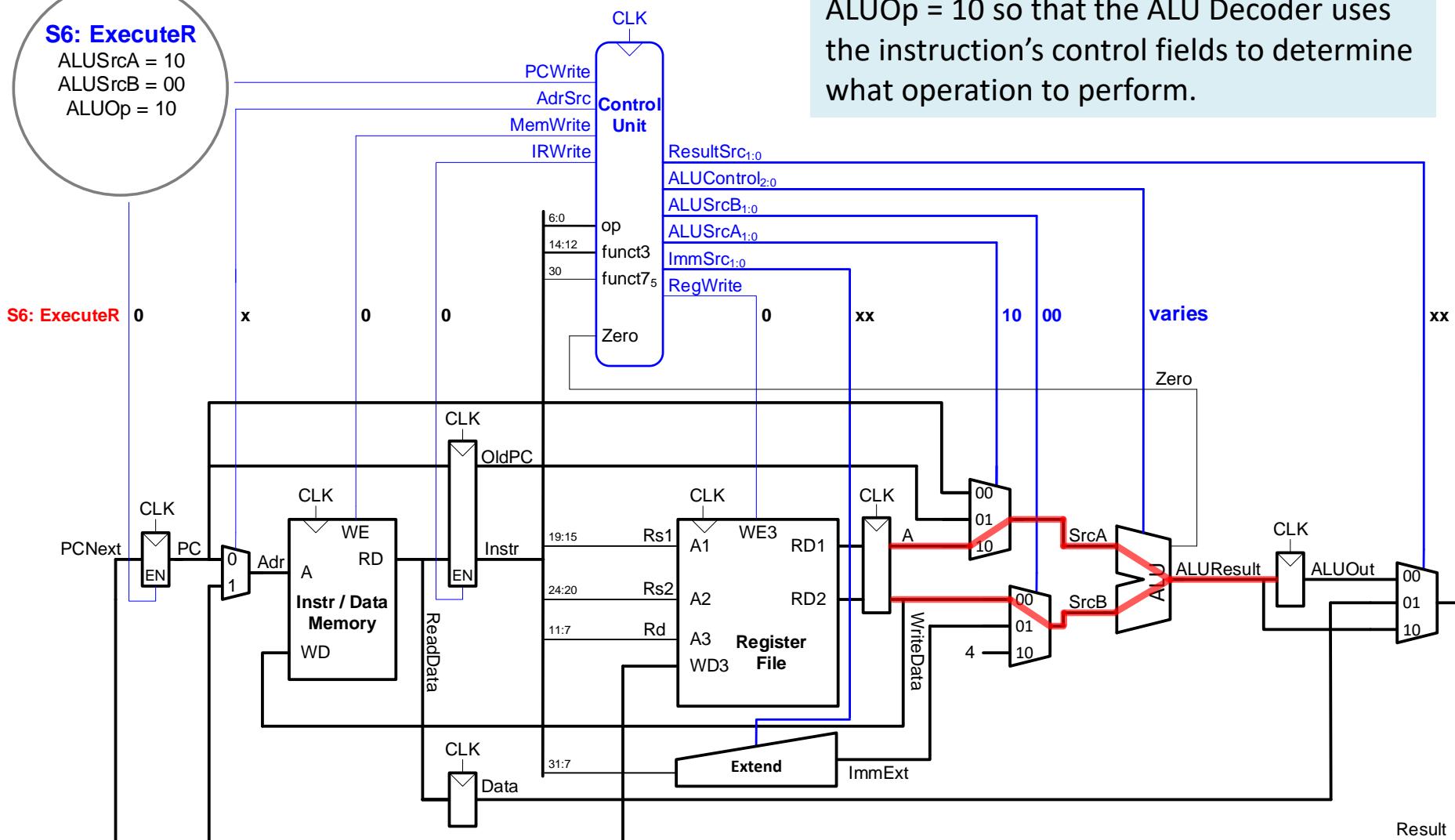
Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4(x9)	I	imm _{11:0} rs1 f3 rd op	0000011 FFC4A303
0x1004	sw x6, 8(x9)	S	imm _{11:5} rs2 rs1 f3 imm _{4:0} op	0100011 0064A423
0x1008	or x4, x5, x6	R	funct7 rs2 rs1 f3 rd op	0110011 0062E233
0x100C	beq x4, x4, l7	B	imm _{11:10} rs2 rs1 f3 imm _{4:1,11} op	1100011 FE420AE3

Main FSM: R-Type Execute Datapath

S6: ExecuteR

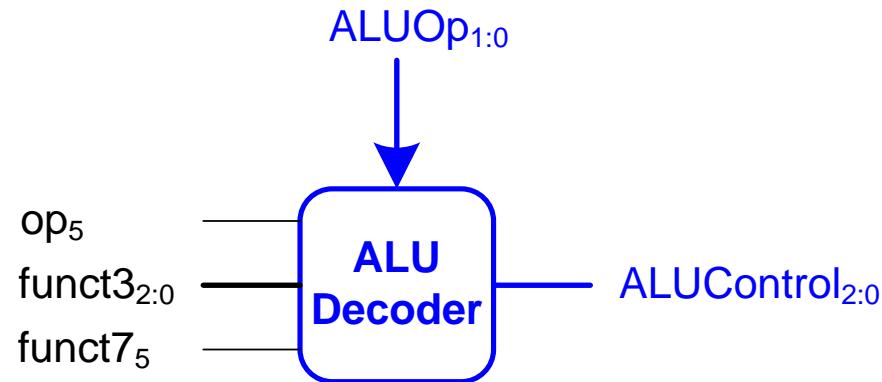
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10

ALUOp = 10 so that the ALU Decoder uses the instruction's control fields to determine what operation to perform.

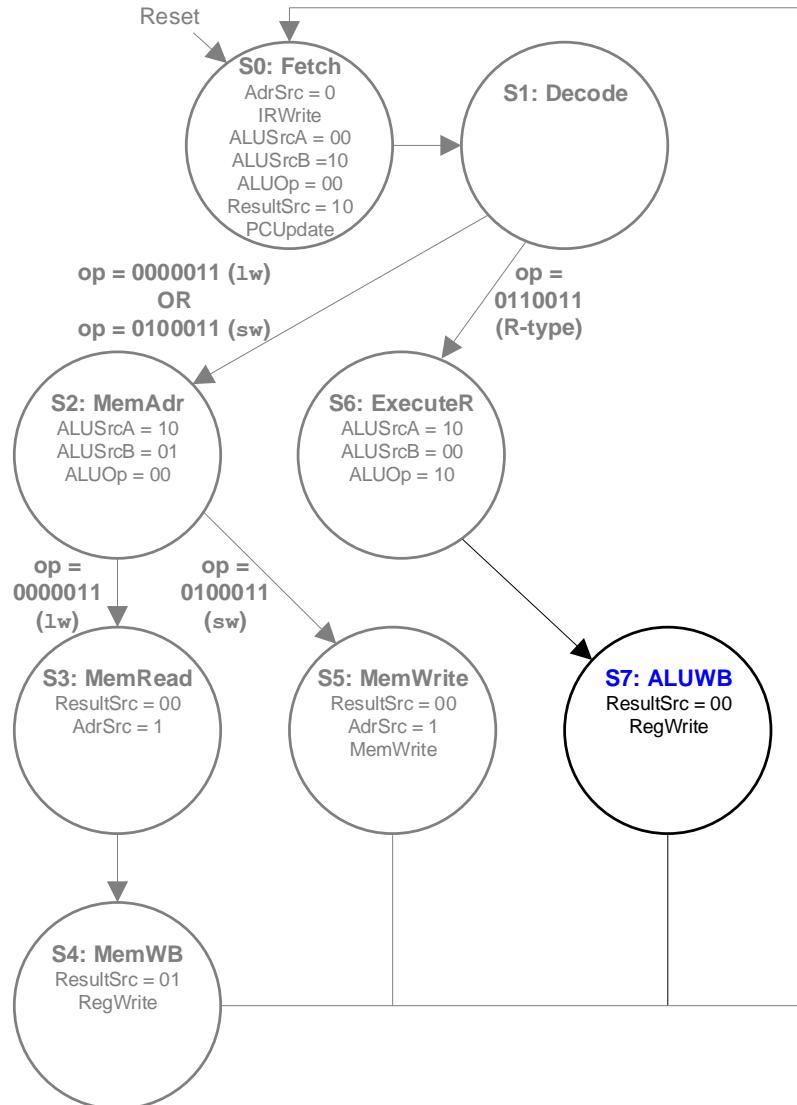


Single-Cycle Control: ALU Decoder

<i>ALUOp</i>	<i>funct3</i>	<i>op₅</i> , <i>funct7₅</i>	<i>Instruction</i>	<i>ALUControl_{2:0}</i>
00	x	x	lw, sw	000 (add)
01	x	x	beq	001 (subtract)
10	000	00, 01, 10	add	000 (add)
	000	11	sub	001 (subtract)
	010	x	slt	101 (set less than)
	110	x	or	011 (or)
	111	x	and	010 (and)



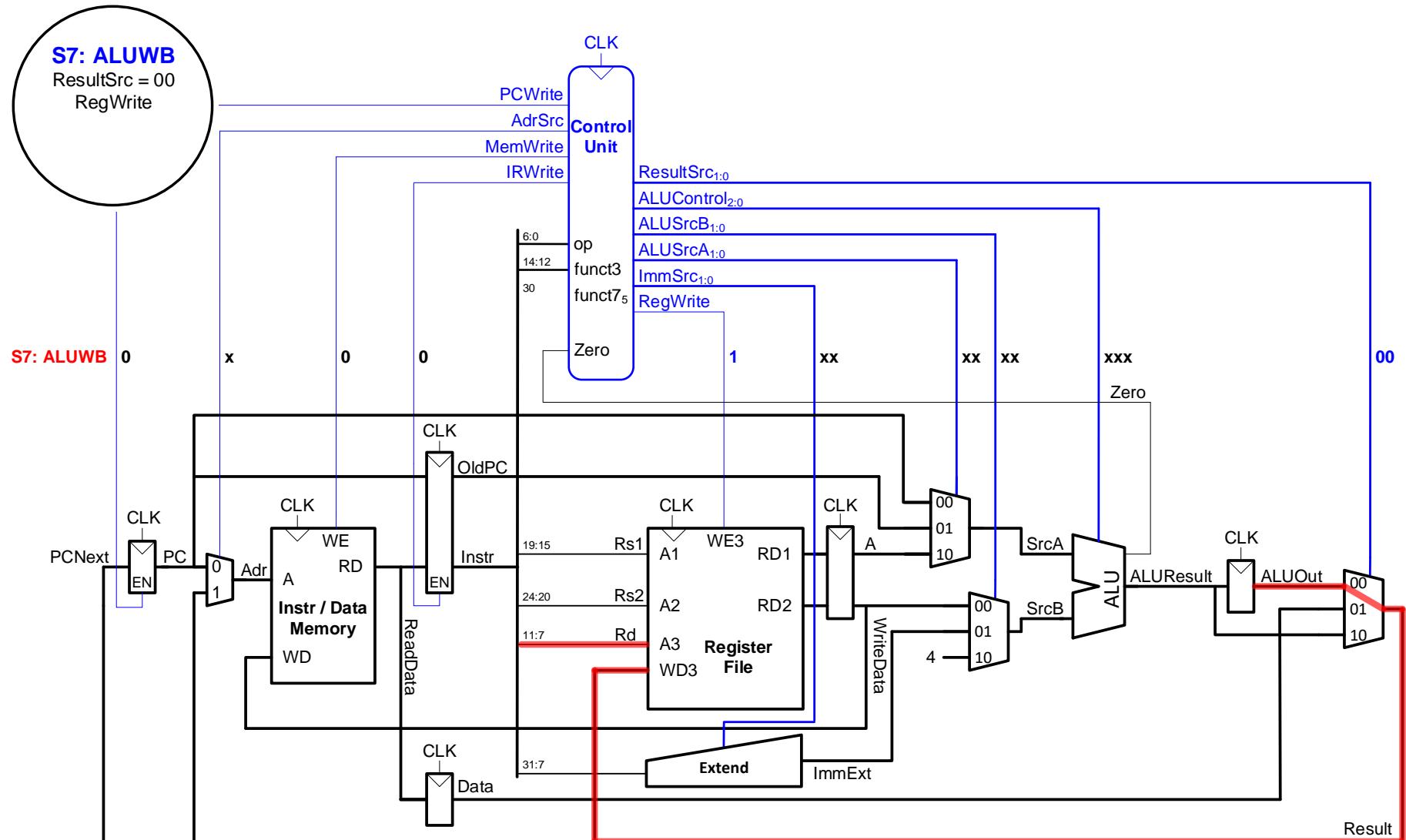
Main FSM: R-Type ALU Write Back



Example Program:

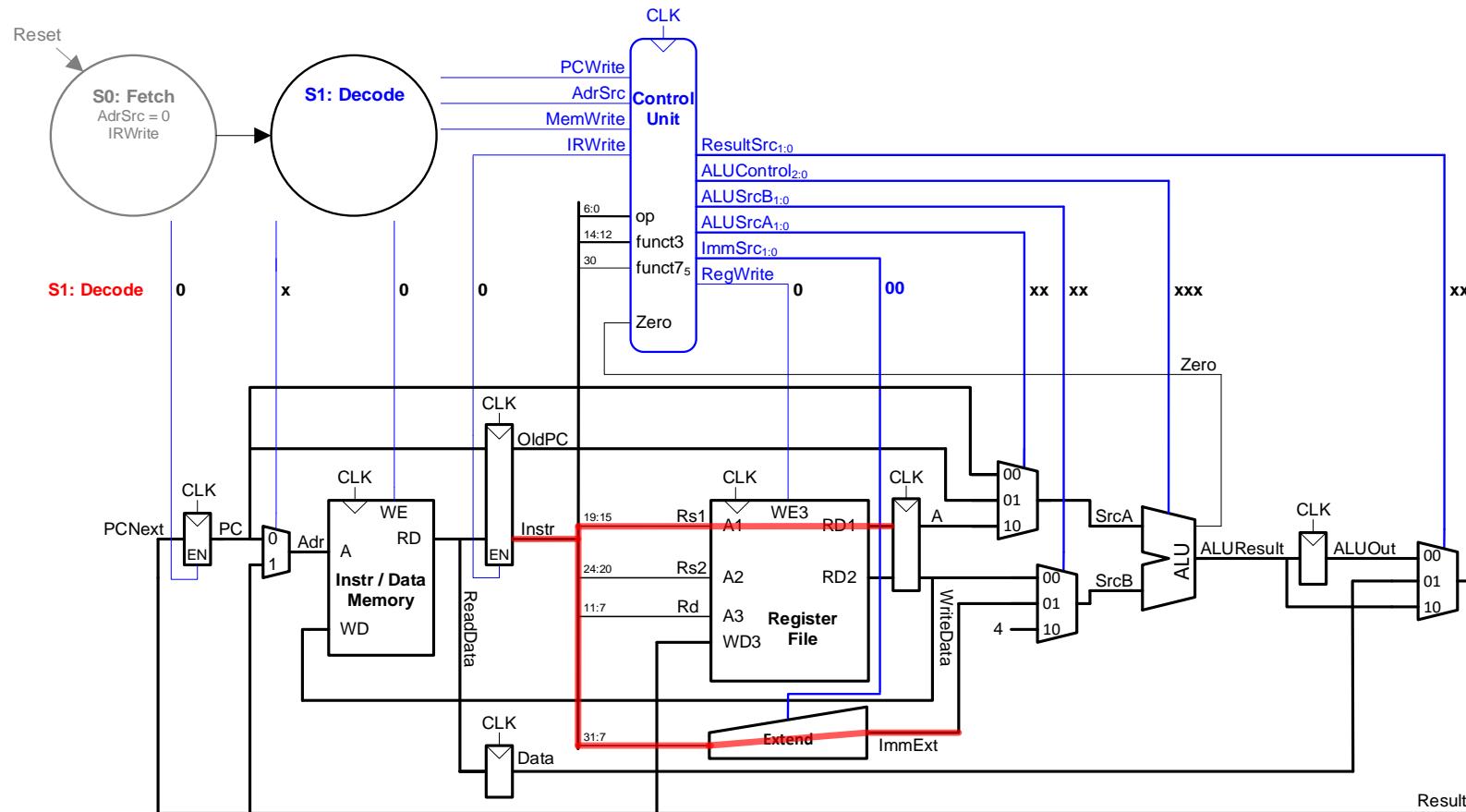
Address	Instruction	Type	Fields	Machine Language
0x1000	l7: lw x6, -4(x9)	I	imm _{11:0} rs1 f3 rd op	0000011 FFC4A303
0x1004	sw x6, 8(x9)	S	imm _{11:5} rs2 rs1 f3 imm _{4:0} op	0100011 0064A423
0x1008	or x4, x5, x6	R	funct7 rs2 rs1 f3 rd op	0110011 0062E233
0x100C	beq x4, x4, l7	B	imm _{12:5} rs2 rs1 f3 imm _{4:1,11} op	1100011 FE420AE3

Main FSM: R-Type ALU Write Back

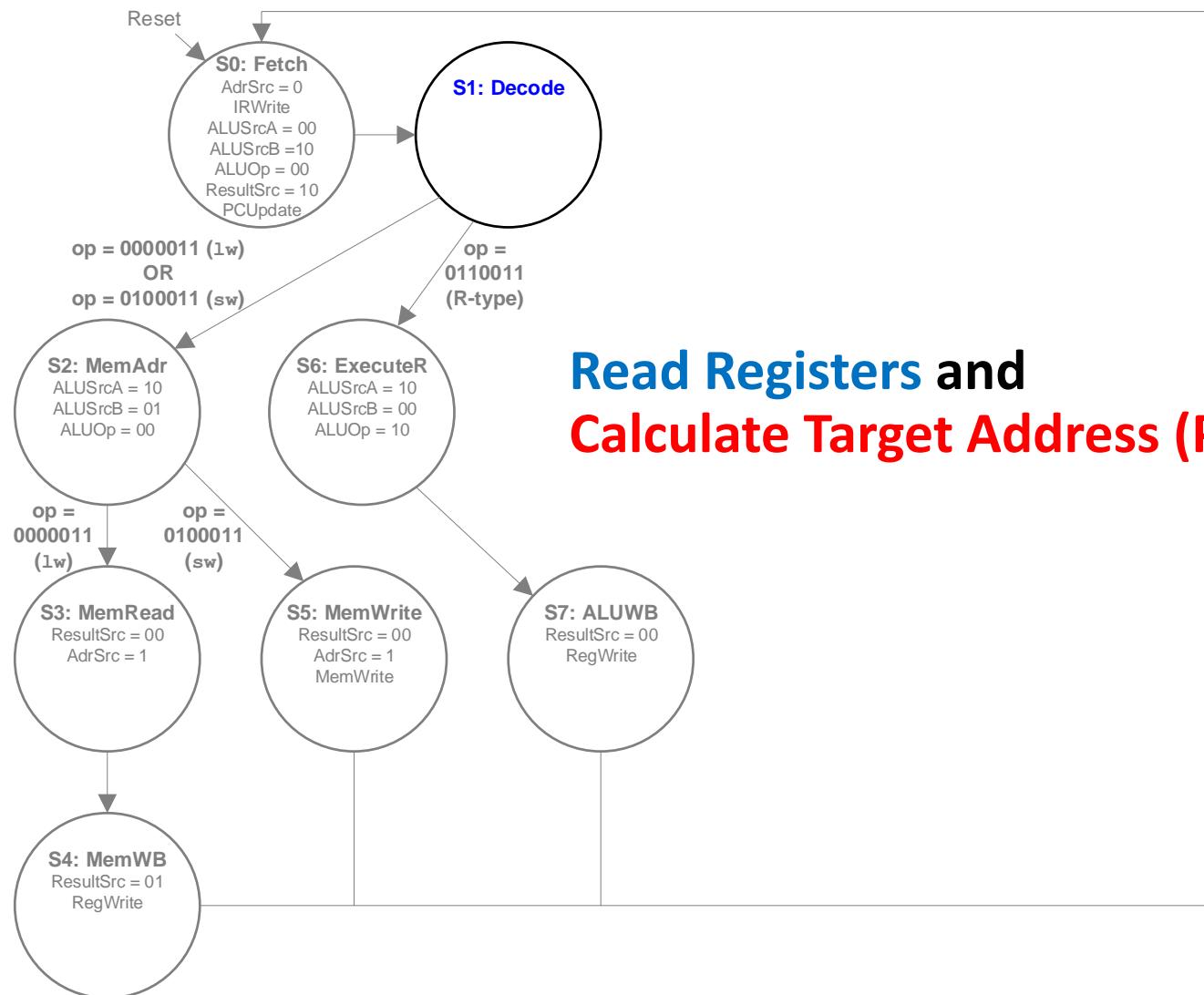


Main FSM: beq

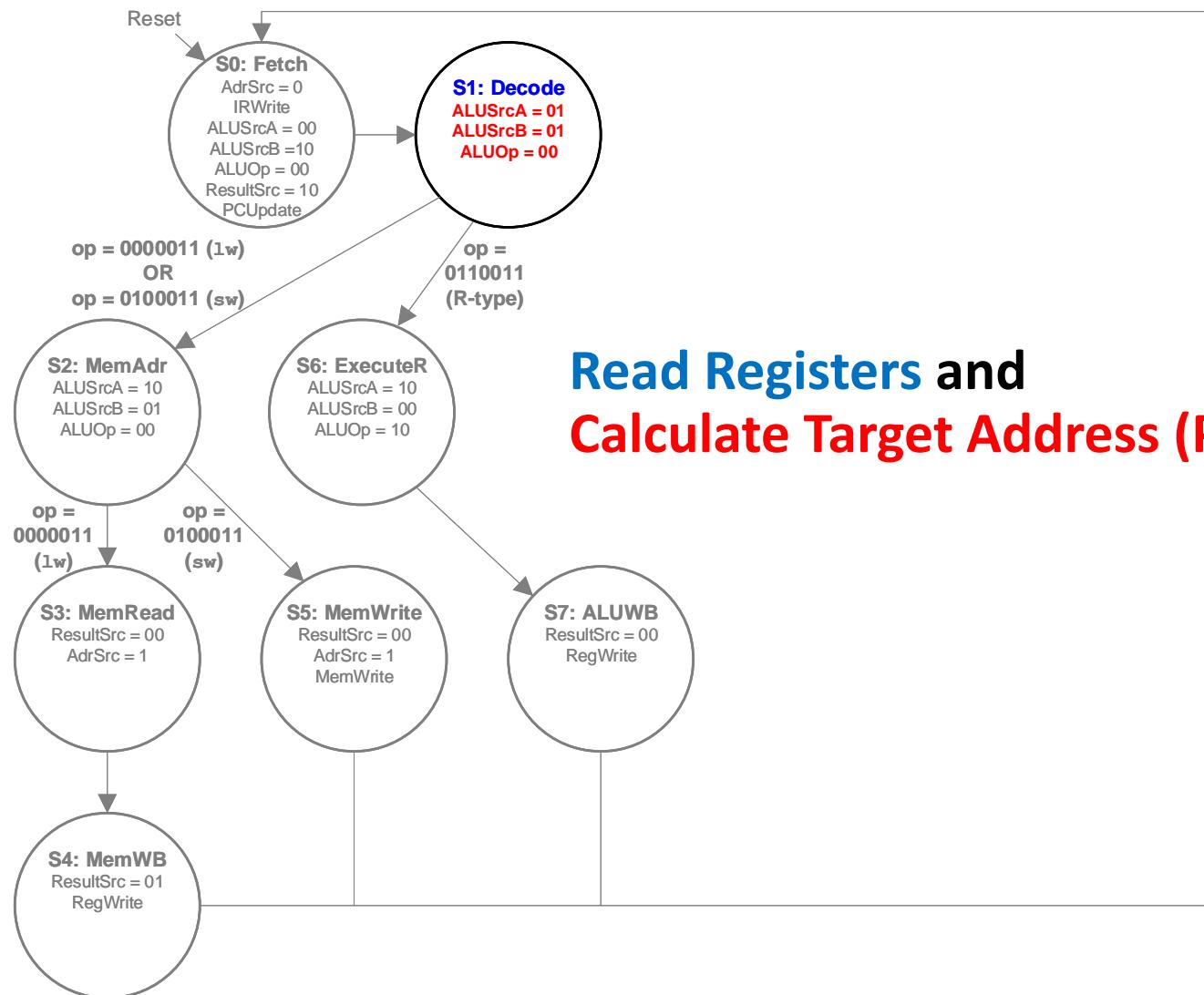
- Need to calculate:
 - Branch Target Address
 - **rs1 - rs2** (to see if equal)
- ALU isn't being used in Decode stage
 - Use it to calculate Target Address (PC + imm)



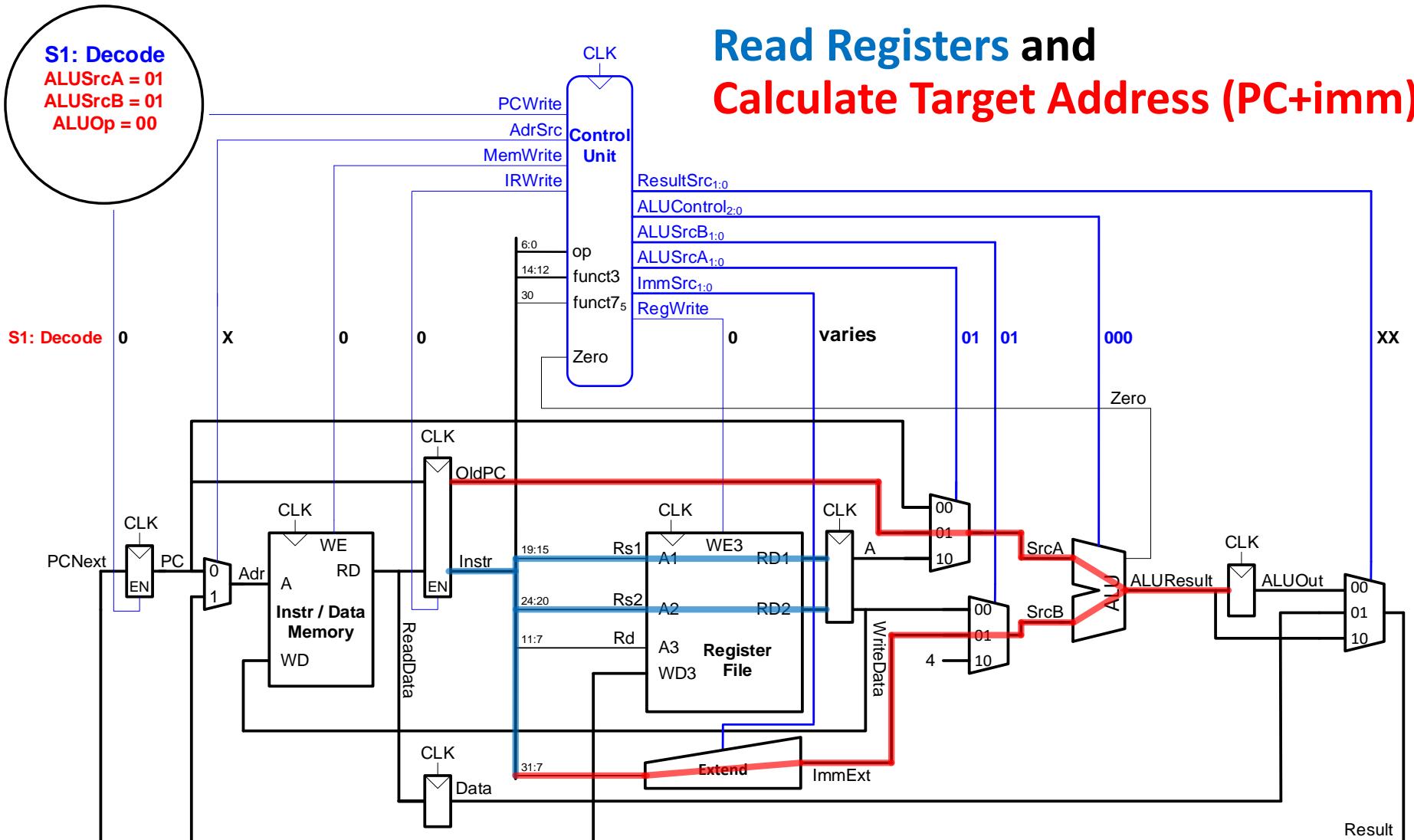
Main FSM: Decode Revisited



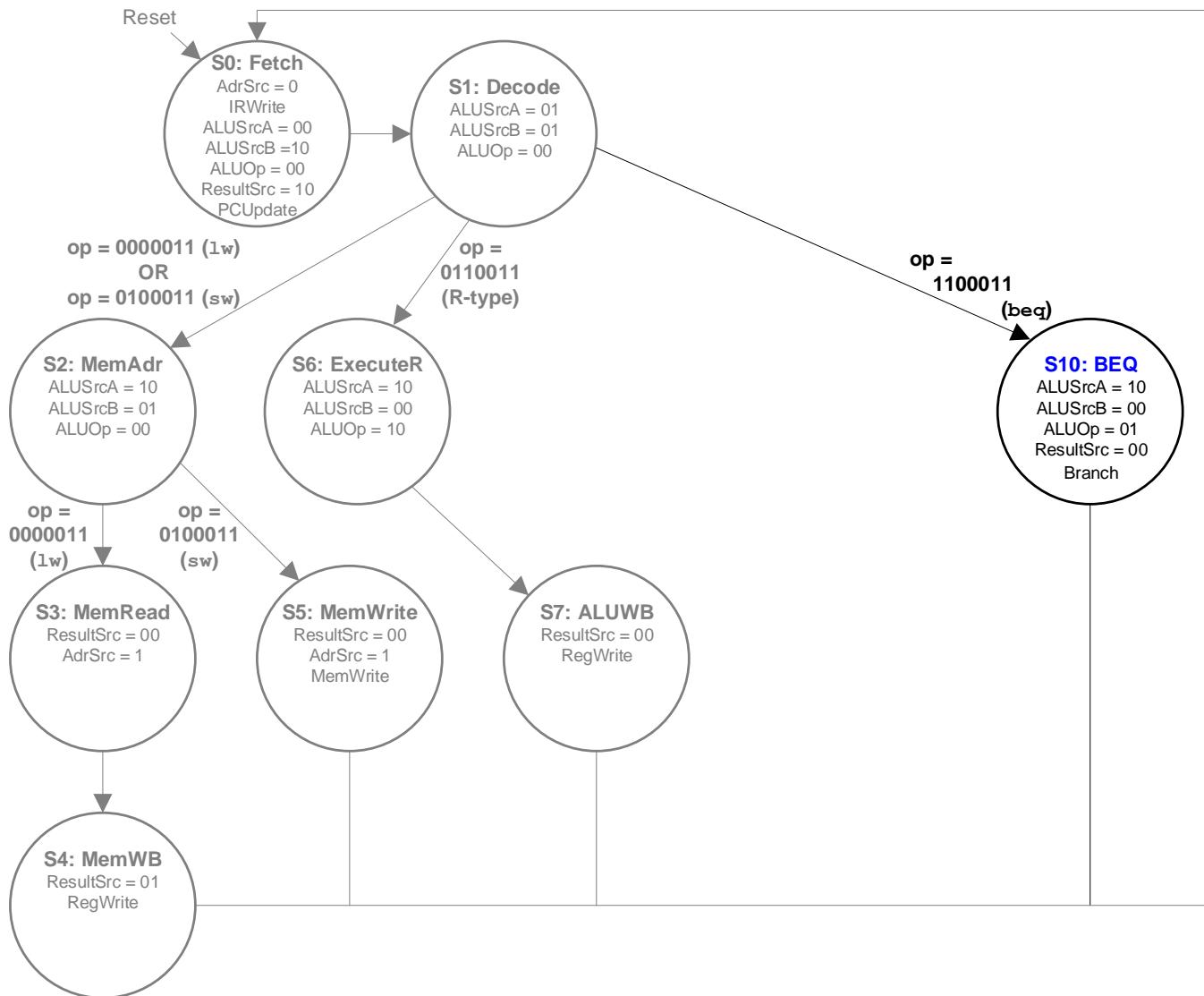
Main FSM: Decode Revisited



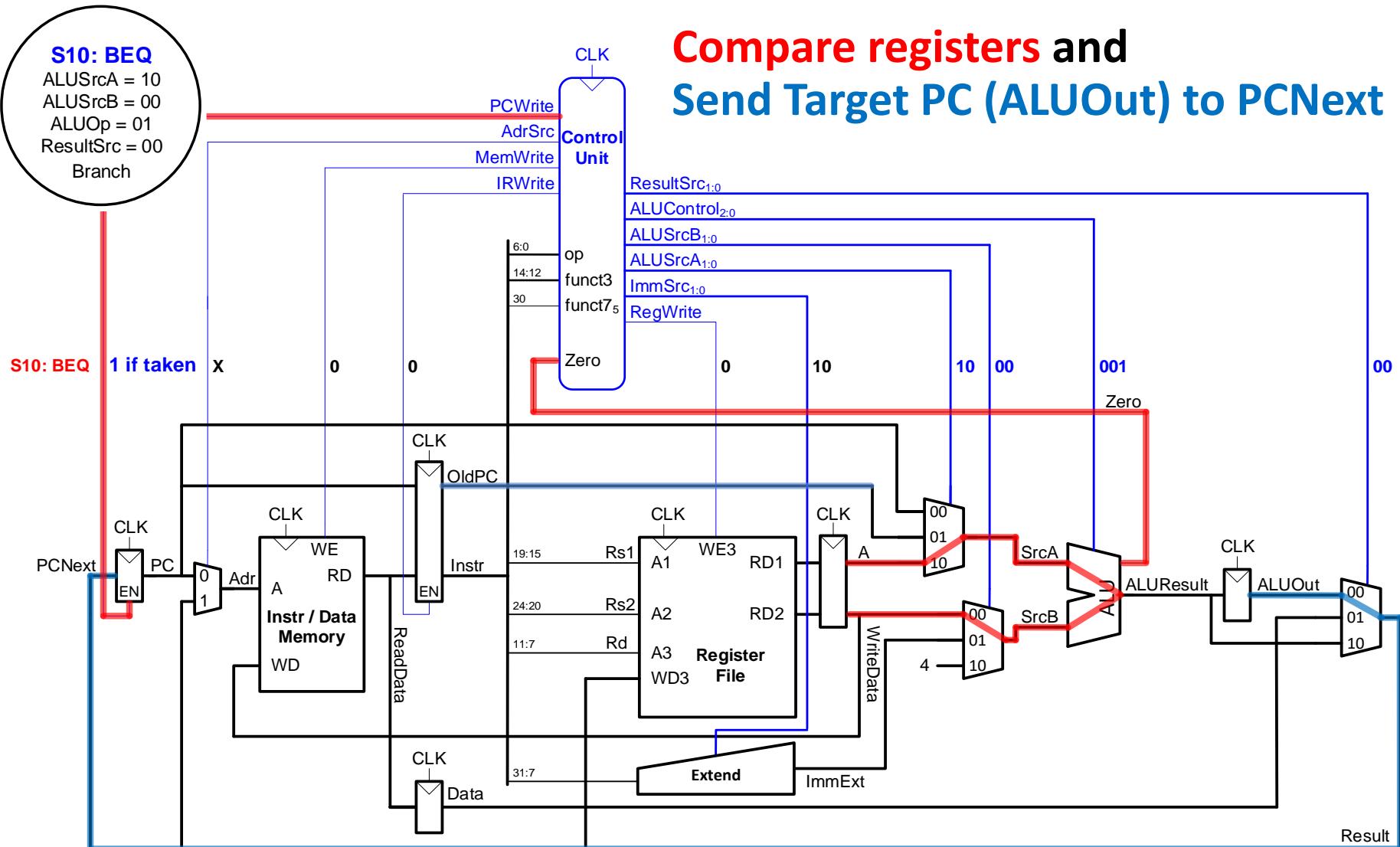
Main FSM: Decode (Target Address)



Main FSM: beq

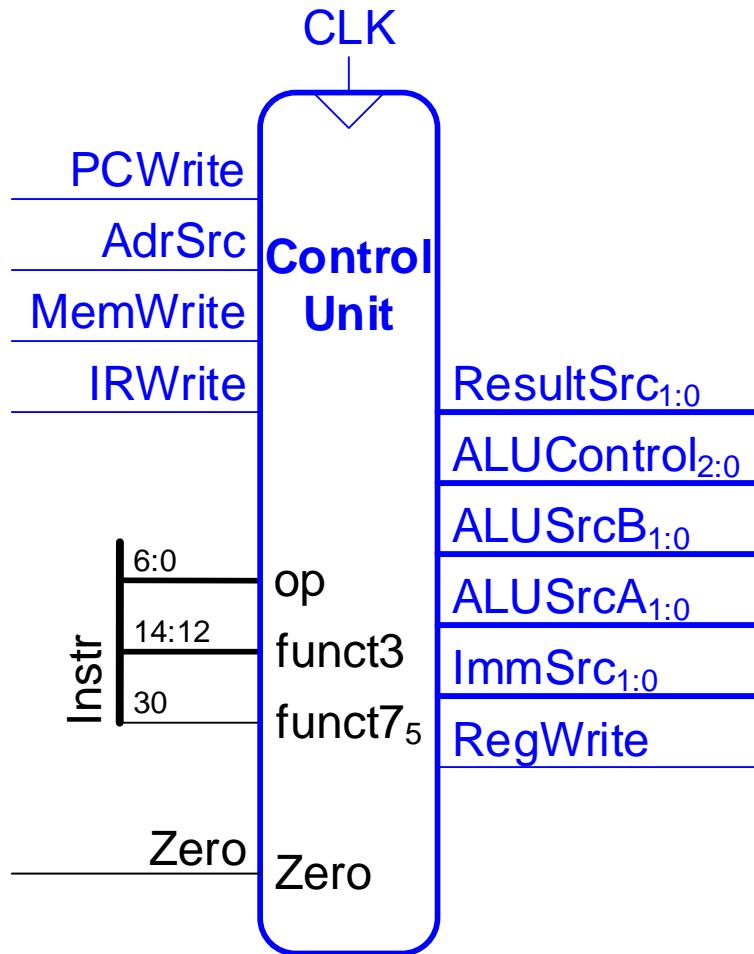


Main FSM: beq Datapath

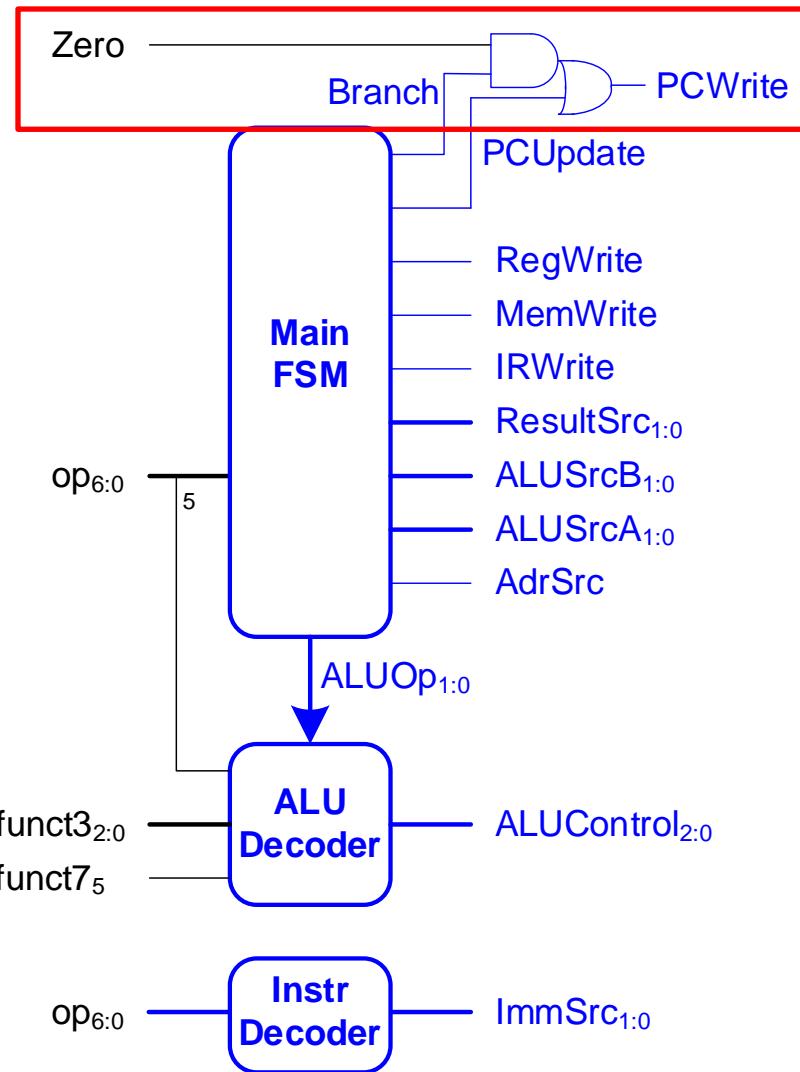


Multicycle Control

High-Level View



Low-Level View

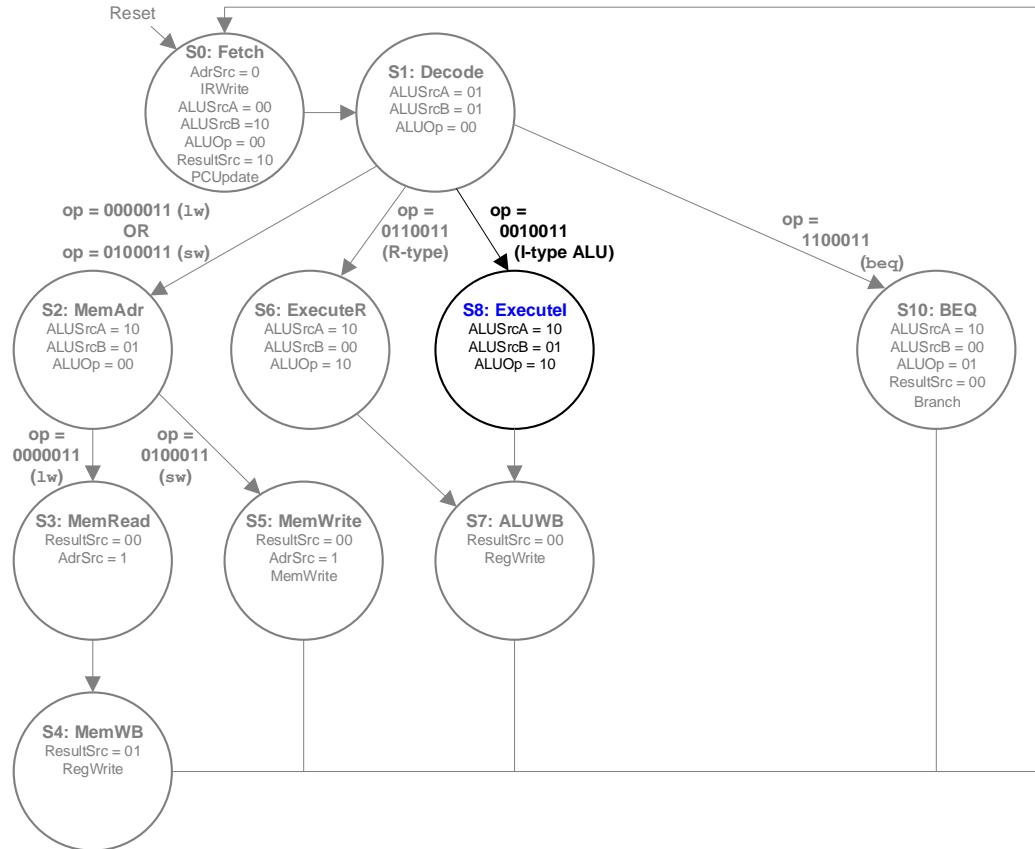


Chapter 7: Microarchitecture

Extending the RISC-V Multicycle Processor

Main FSM: I-Type ALU Execute

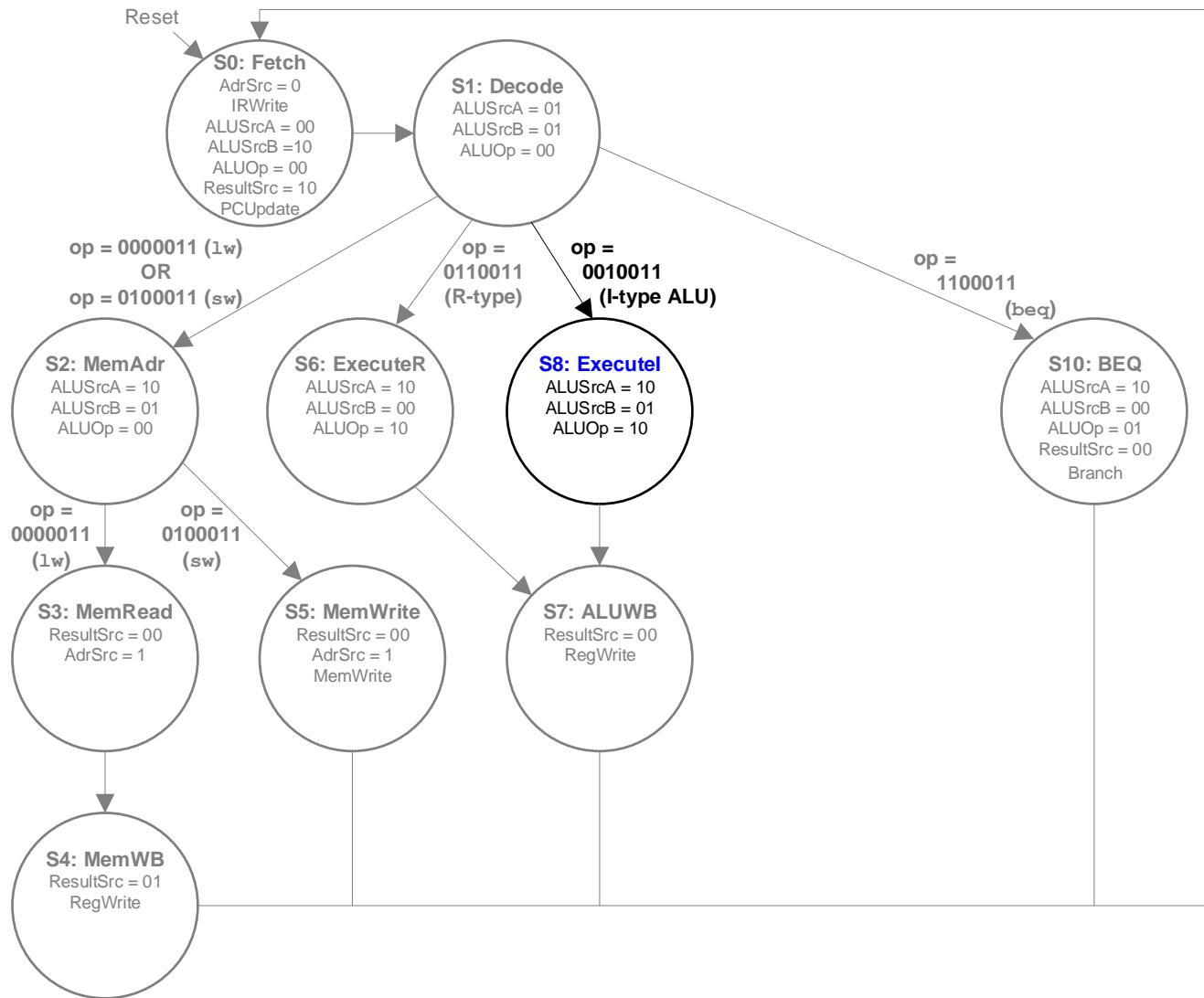
Modify the multicycle controller to handle the I-type ALU instructions (addi, andi, ori, slti).



- These I-type ALU instructions are nearly the same as their R-type equivalents (add, and or, and slt) except the second source comes from ImmExt rather than the register file.
- We introduce the Executel state to perform the desired computation for all I-type ALU instructions.
- After the Executel state, I-type ALU instructions proceed to the ALU writeback (ALUWB) state to write the result to the register file.

addi rd, rsl, imm	add immediate	rd = rsl + SignExt(imm)
add rd, rsl, rs2	add	rd = rsl + rs2

Main FSM: I-Type ALU Execute



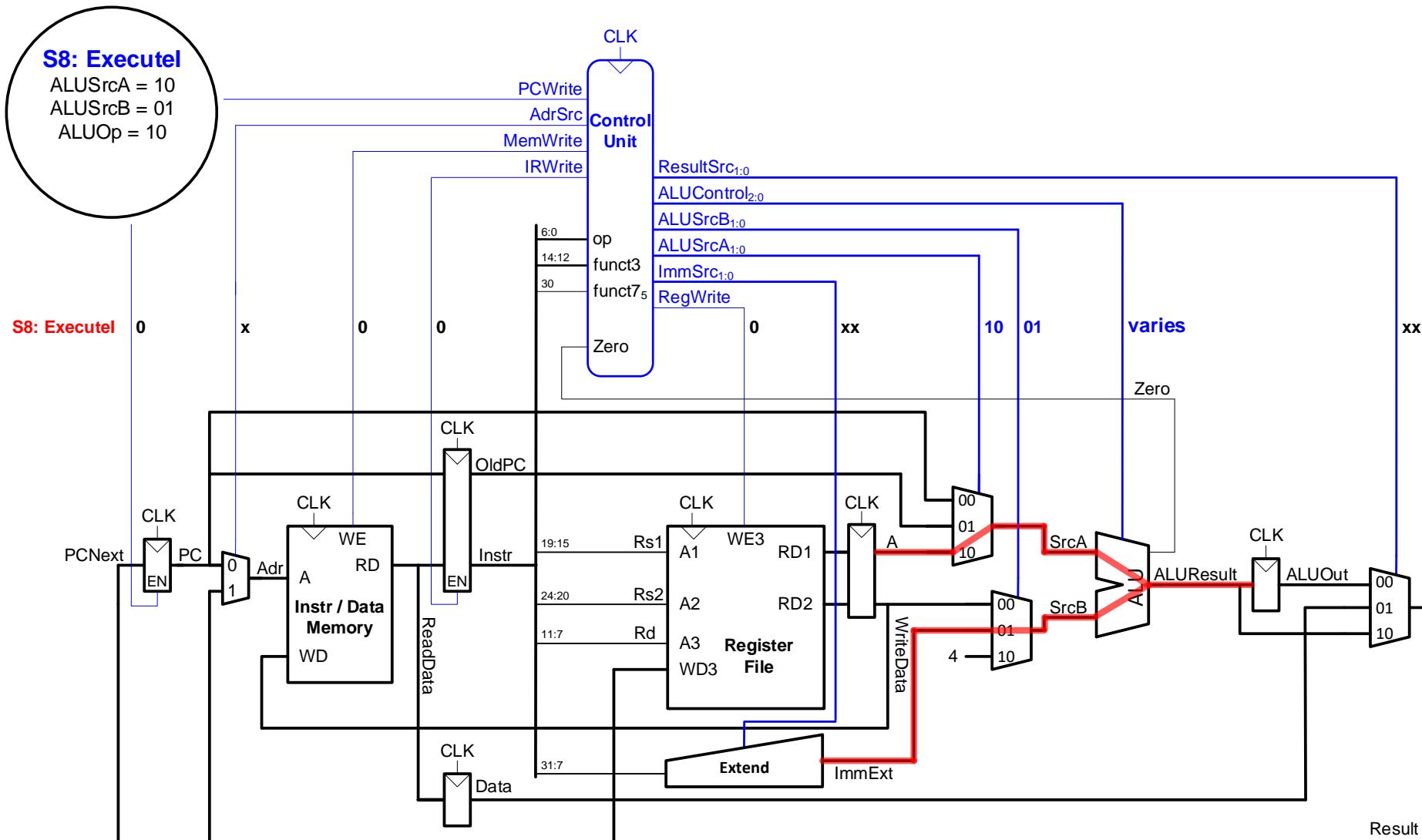
Modify the multicycle controller to handle the I-type ALU instructions (addi, andi, ori, slti).

Main FSM: I-Type ALU Execute

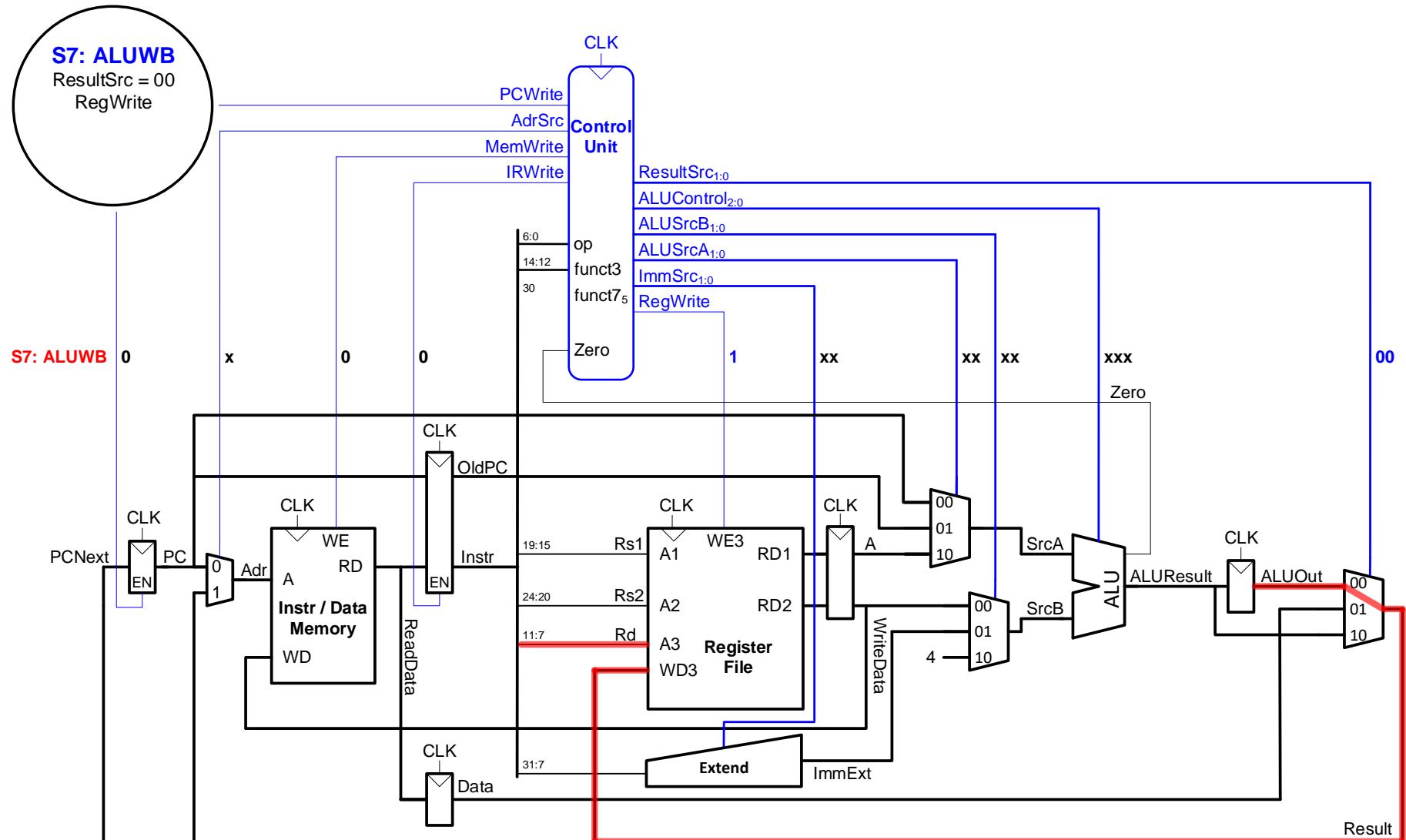
- These I-type ALU instructions are nearly the same as their R-type equivalents (add, and or, and slt) except the second source comes from ImmExt rather than the register file.

addi rd, rs1, imm	add immediate	$rd = rs1 + \text{SignExt}(imm)$
slli rd, rs1, uimm	shift left logical immediate	$rd = rs1 \ll uimm$
slti rd, rs1, imm	set less than immediate	$rd = (rs1 < \text{SignExt}(imm))$
sltiu rd, rs1, imm	set less than imm. unsigned	$rd = (rs1 < \text{SignExt}(imm))$
xori rd, rs1, imm	xor immediate	$rd = rs1 ^ \text{SignExt}(imm)$
srlti rd, rs1, uimm	shift right logical immediate	$rd = rs1 \gg uimm$
srai rd, rs1, uimm	shift right arithmetic imm.	$rd = rs1 \ggg uimm$
ori rd, rs1, imm	or immediate	$rd = rs1 \text{SignExt}(imm)$
andi rd, rs1, imm	and immediate	$rd = rs1 \& \text{SignExt}(imm)$
add rd, rs1, rs2	add	$rd = rs1 + rs2$
sub rd, rs1, rs2	sub	$rd = rs1 - rs2$
sll rd, rs1, rs2	shift left logical	$rd = rs1 \ll rs2_{4:0}$
slt rd, rs1, rs2	set less than	$rd = (rs1 < rs2)$
sltu rd, rs1, rs2	set less than unsigned	$rd = (rs1 < rs2)$
xor rd, rs1, rs2	xor	$rd = rs1 ^ rs2$
srl rd, rs1, rs2	shift right logical	$rd = rs1 \gg rs2_{4:0}$
sra rd, rs1, rs2	shift right arithmetic	$rd = rs1 \ggg rs2_{4:0}$
or rd, rs1, rs2	or	$rd = rs1 rs2$
and rd, rs1, rs2	and	$rd = rs1 \& rs2$

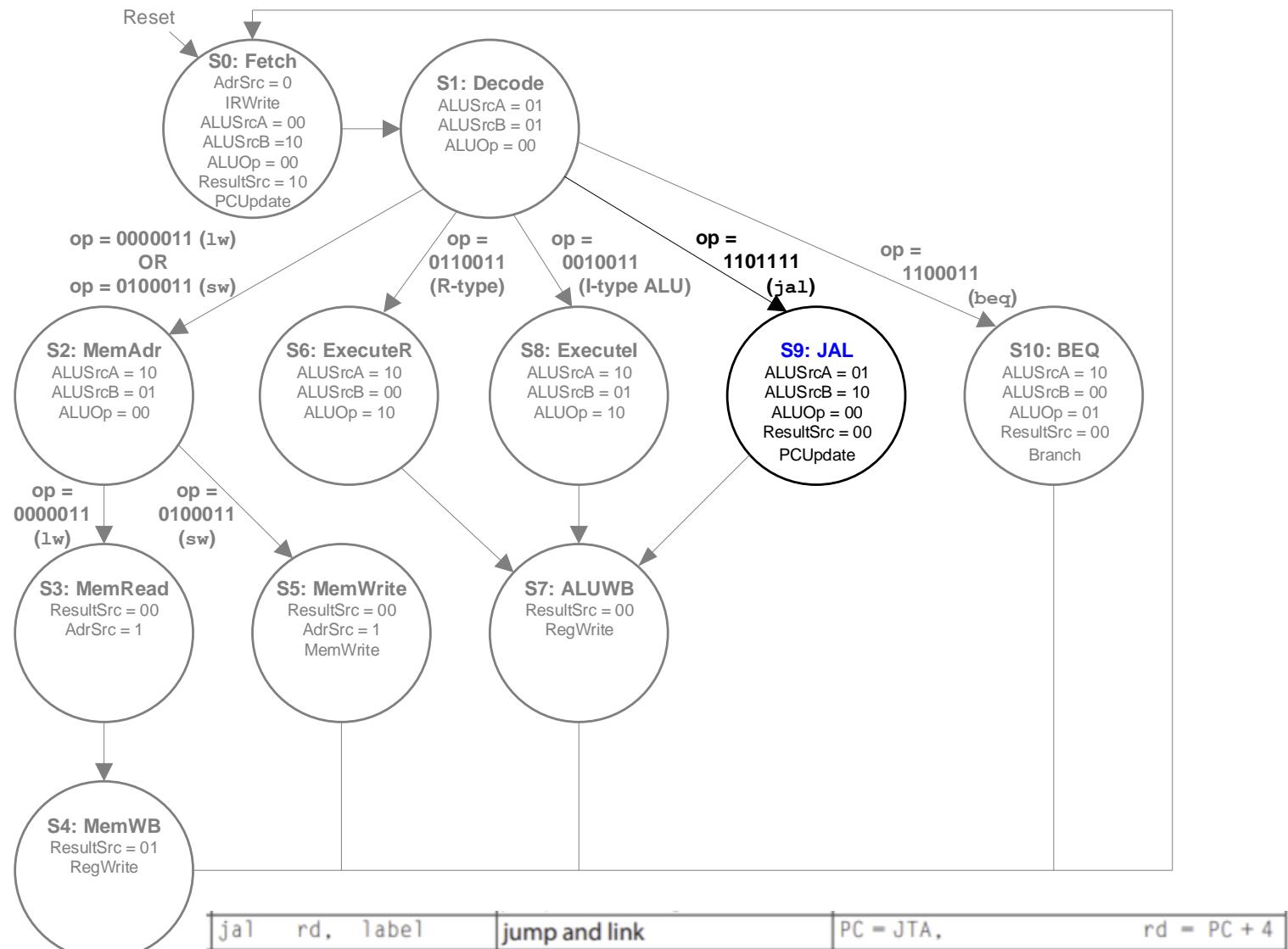
Main FSM: I-Type ALU Exec. Datapath



Main FSM: R-Type ALU Write Back



Main FSM: jal

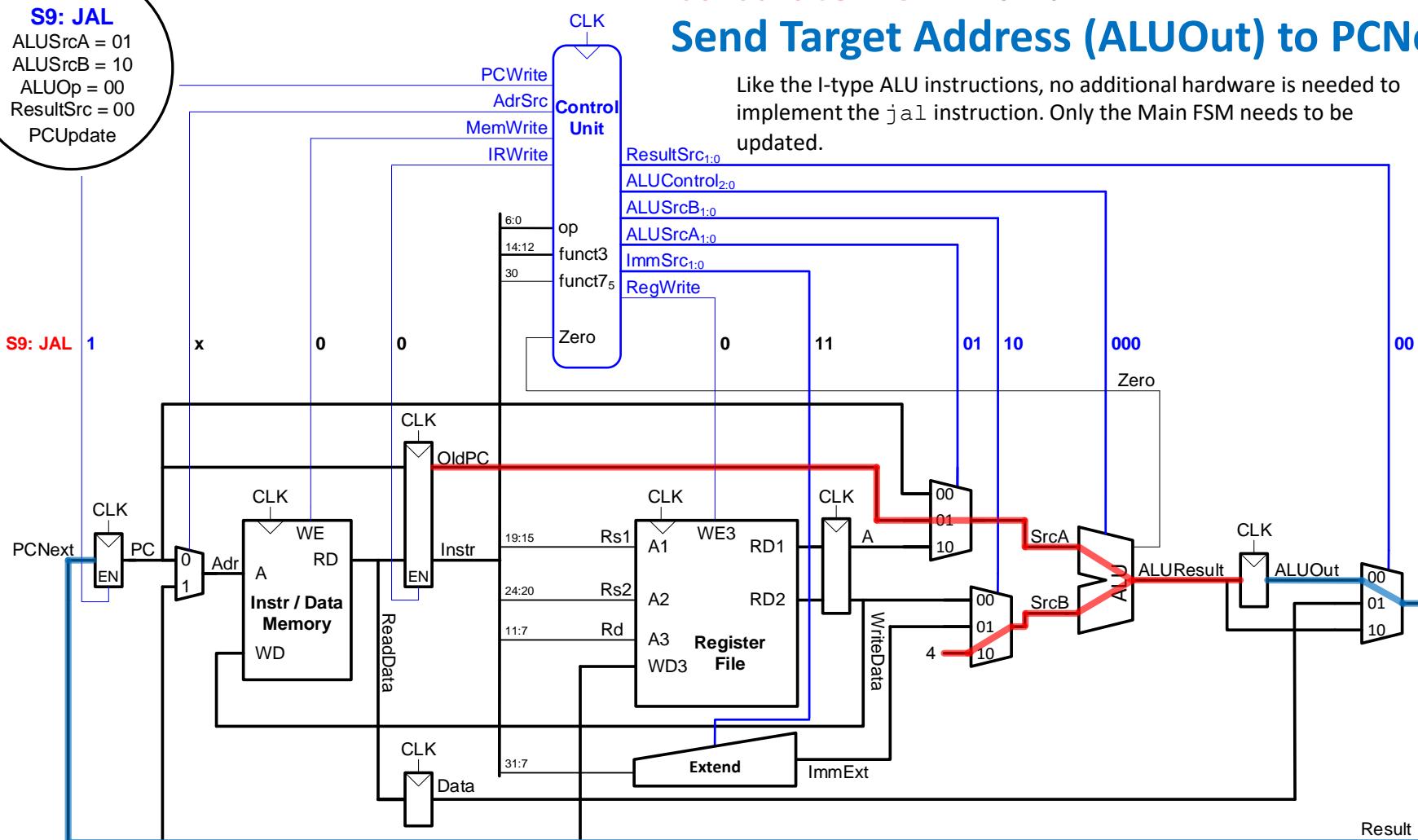


Main FSM: jal Datapath

S9: JAL
 ALUSrcA = 01
 ALUSrcB = 10
 ALUOp = 00
 ResultSrc = 00
 PCUpdate

Calculate PC + 4 and Send Target Address (ALUOut) to PCNext

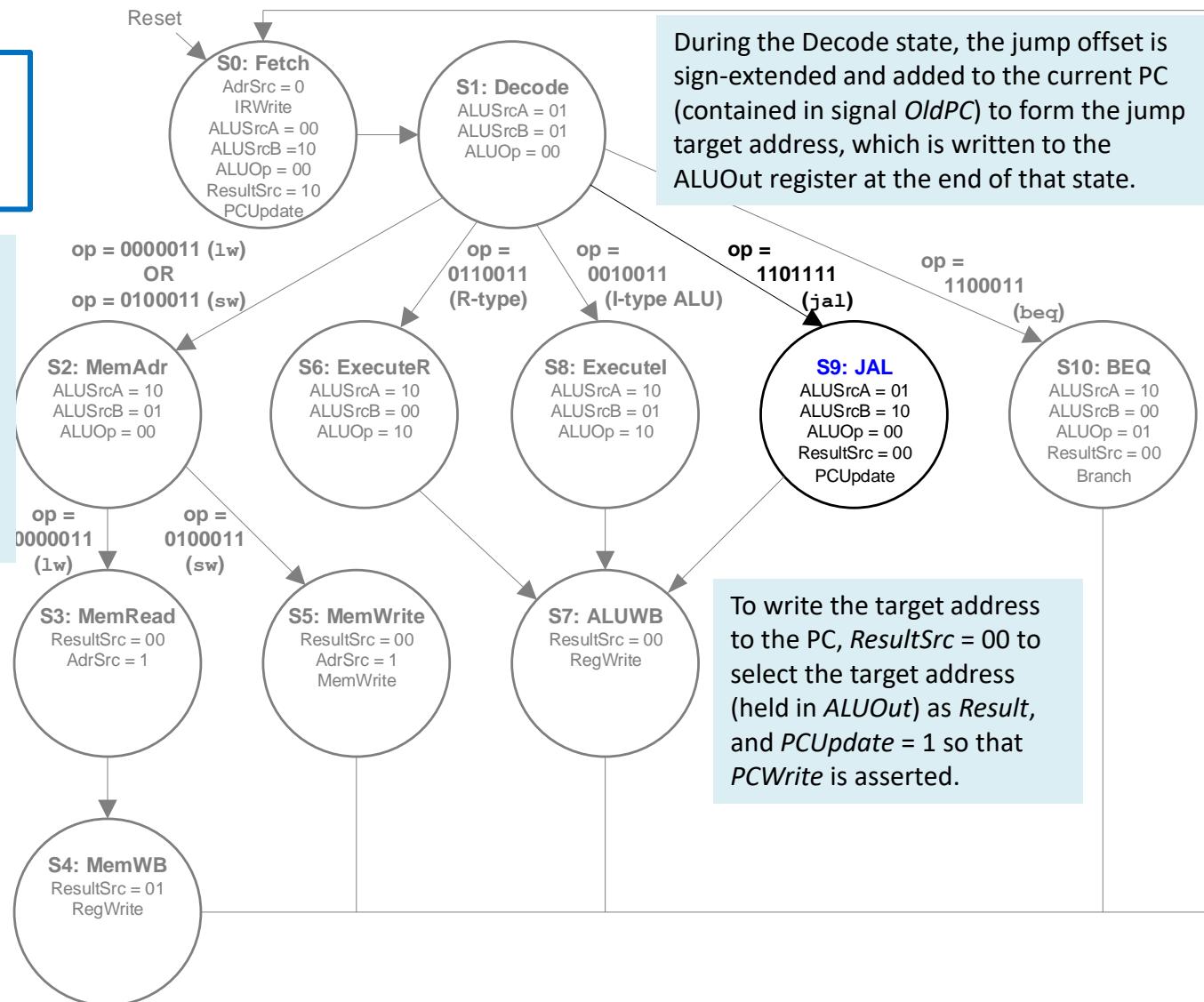
Like the I-type ALU instructions, no additional hardware is needed to implement the `jal` instruction. Only the Main FSM needs to be updated.



Main FSM: jal

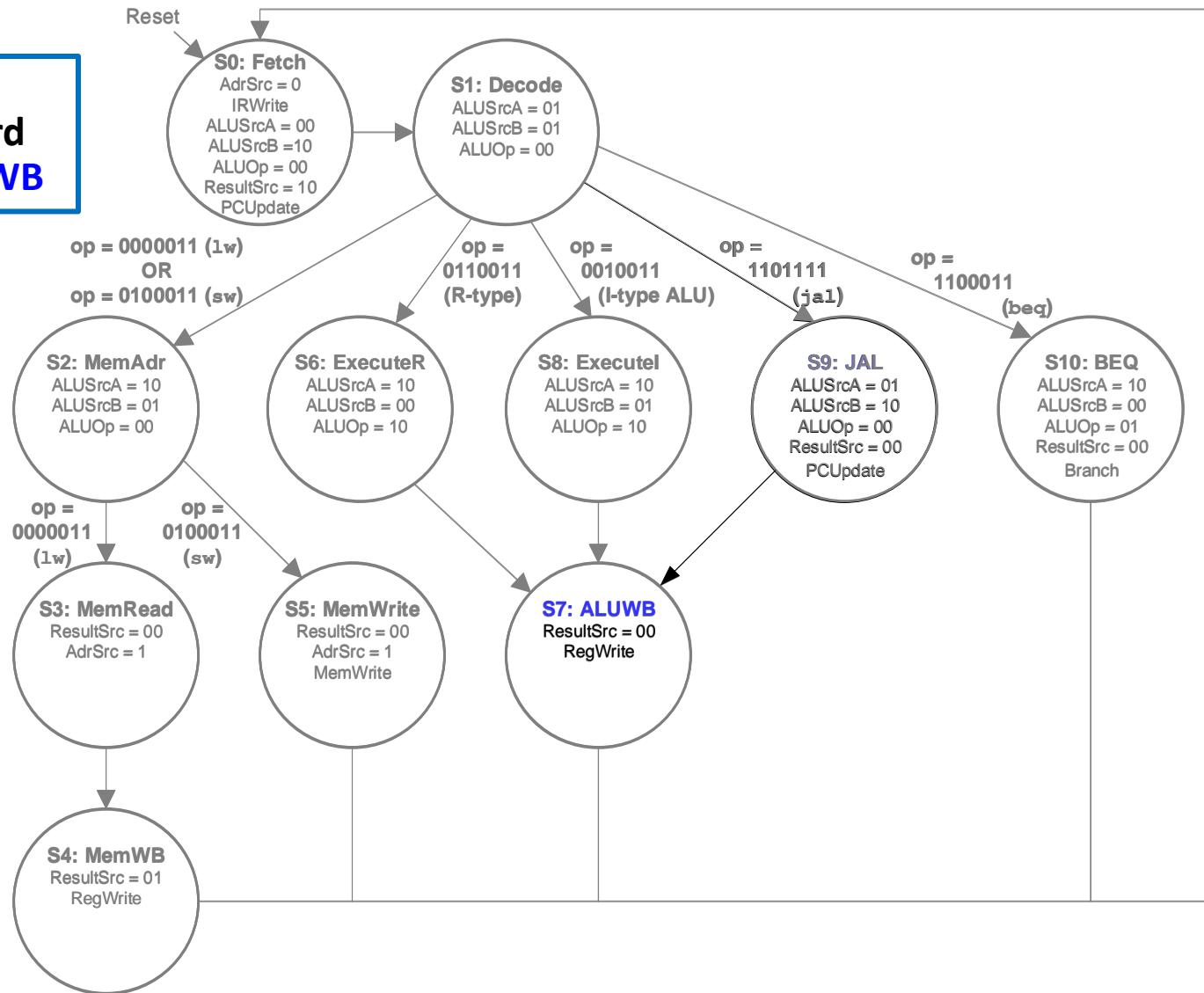
PC + 4 is
written to rd
in **S7: ALUWB**

- The ALU calculates the return address $PC+4$ (i.e., $OldPC+4$) using
 - $ALUSrcA = 01$ ($SrcA = OldPC$),
 - $ALUSrcB = 10$ ($SrcB = 4$), and
 - $ALUOp = 00$ for addition.

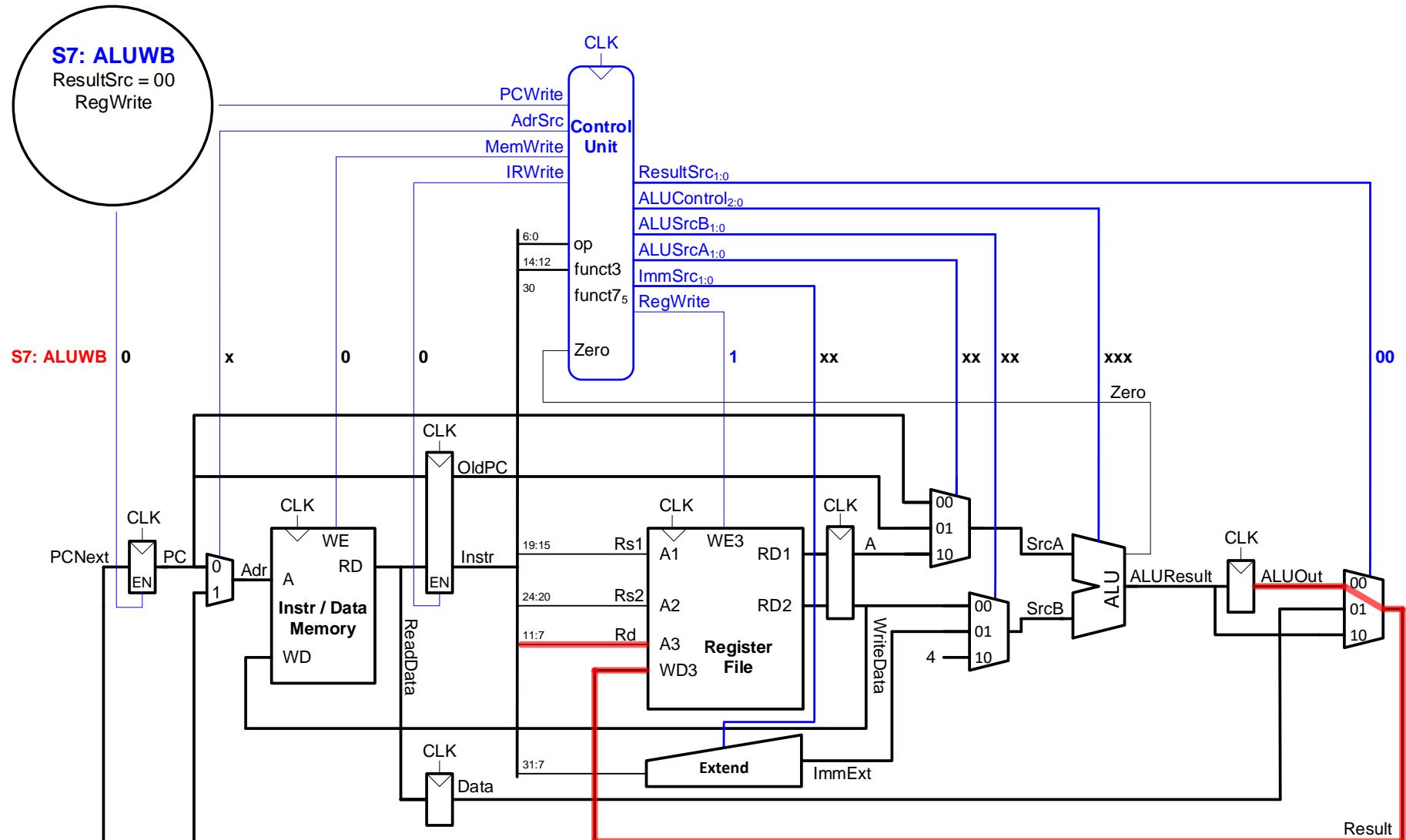


Main FSM: jal

PC + 4 is
written to rd
in **S7: ALUWB**



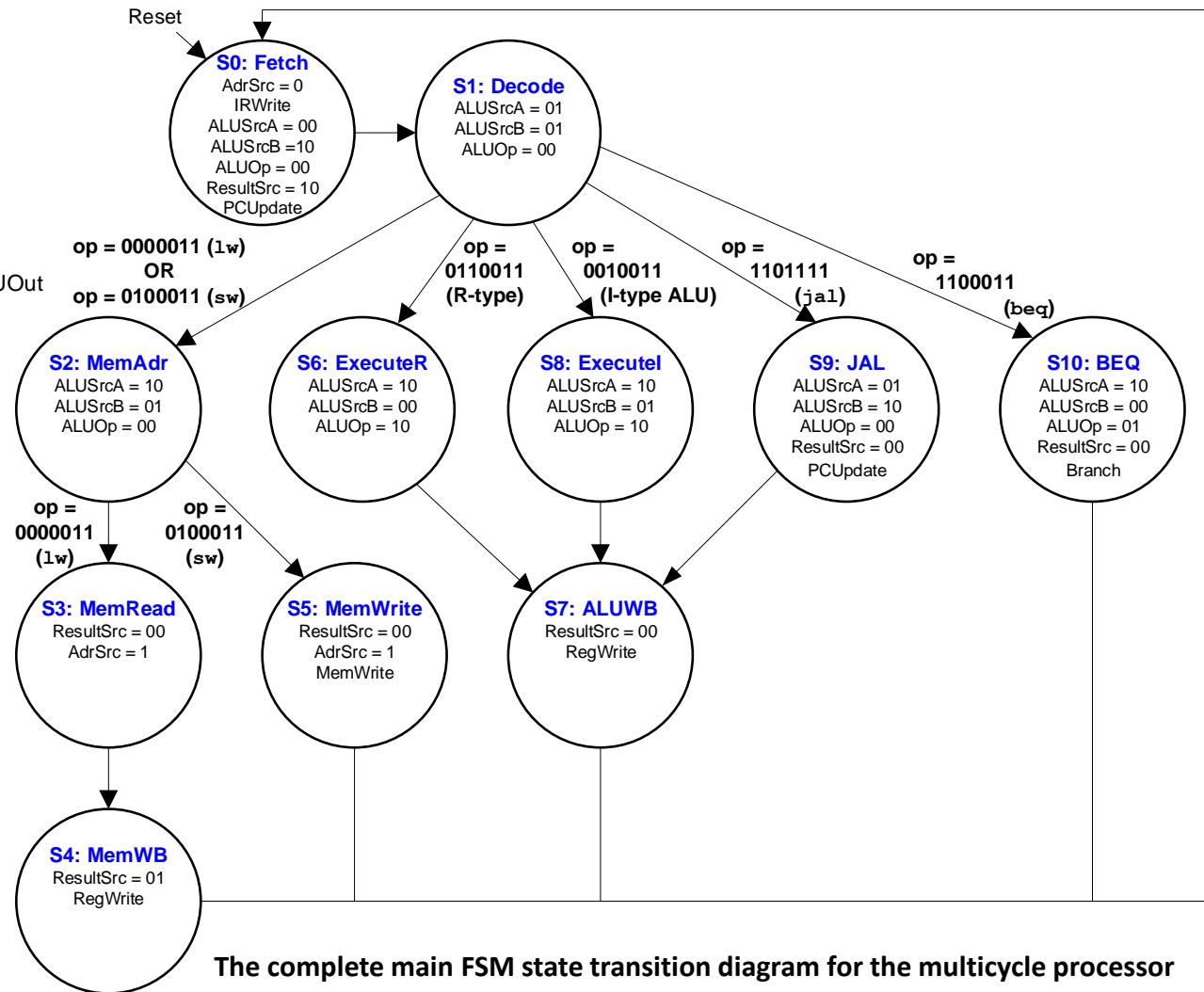
Main FSM: R-Type ALU Write Back



Multicycle Processor Main FSM

State	Datapath μOp
Fetch	Instr \leftarrow Mem[PC]; PC \leftarrow PC+4
Decode	ALUOut \leftarrow PCTarget
MemAdr	ALUOut \leftarrow rs1 + imm
MemRead	Data \leftarrow Mem[ALUOut]
MemWB	rd \leftarrow Data
MemWrite	Mem[ALUOut] \leftarrow rd
ExecuteR	ALUOut \leftarrow rs1 op rs2
Executel	ALUOut \leftarrow rs1 op imm
ALUWB	rd \leftarrow ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC \leftarrow ALUOut
JAL	PC \leftarrow ALUOut; ALUOut \leftarrow PC+4

The function of each state



Chapter 7: Microarchitecture

Multicycle Performance

Multicycle Processor Performance

- Instructions take different number of cycles:

Multicycle Processor Performance

- Instructions take different number of cycles:
 - 3 cycles: beq

Multicycle Processor Performance

- Instructions take different number of cycles:
 - 3 cycles: beq
 - 4 cycles: R-type, addi, sw , jal

Multicycle Processor Performance

- Instructions take different number of cycles:
 - 3 cycles: beq
 - 4 cycles: R-type, addi, sw , jal
 - 5 cycles: lw

Multicycle Processor Performance

- Instructions take different number of cycles:
 - 3 cycles: beq
 - 4 cycles: R-type, addi, sw , jal
 - 5 cycles: lw
- CPI is weighted average

Multicycle Processor Performance

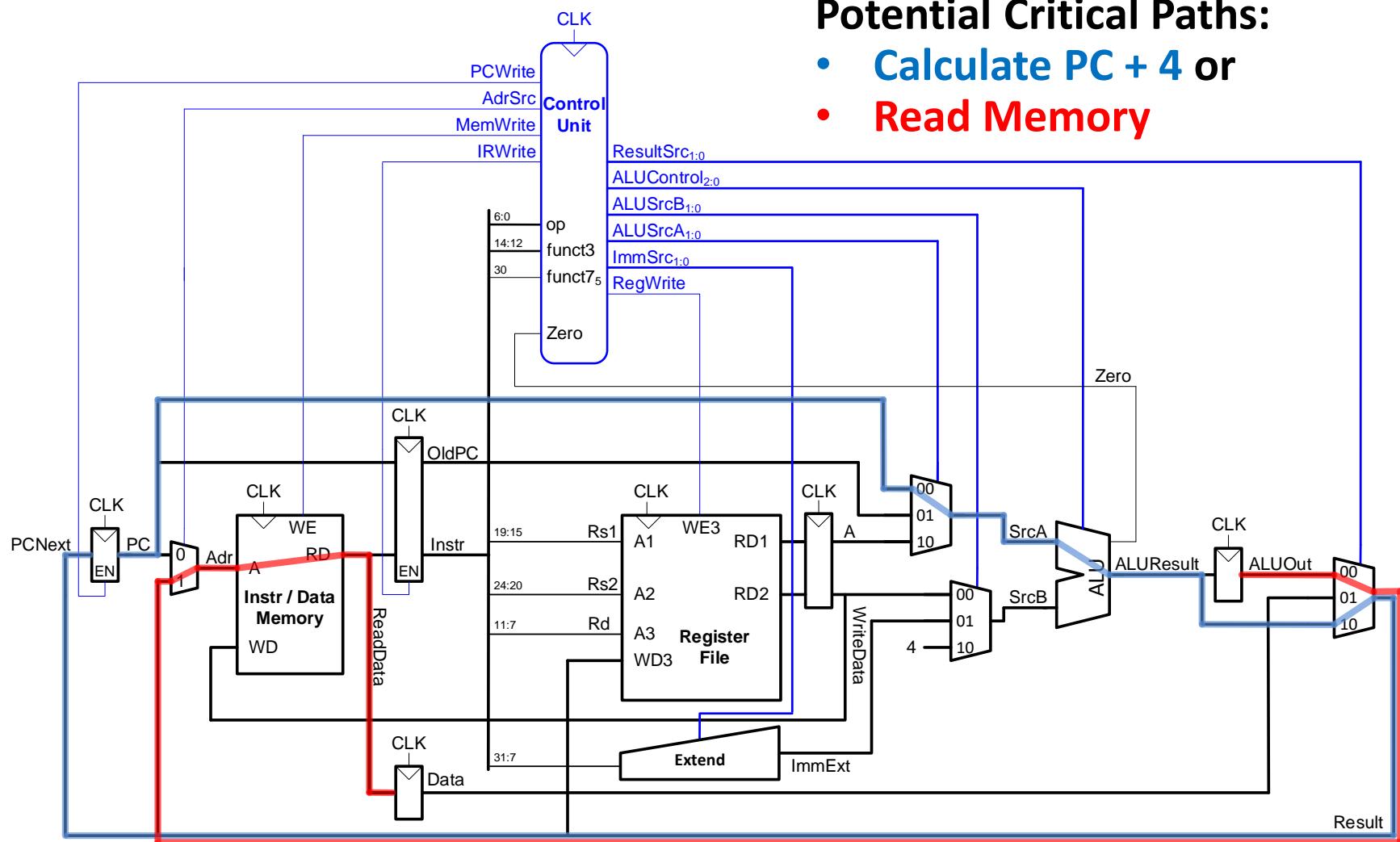
- Instructions take different number of cycles:
 - 3 cycles: beq
 - 4 cycles: R-type, addi, sw , jal
 - 5 cycles: lw
- CPI is weighted average
- SPECINT2000 benchmark:
 - **25%** loads
 - **10%** stores
 - **13%** branches
 - **52%** R-type

Multicycle Processor Performance

- Instructions take different number of cycles:
 - 3 cycles: beq
 - 4 cycles: R-type, addi, sw , jal
 - 5 cycles: lw
- CPI is weighted average
- SPECINT2000 benchmark:
 - 25% loads
 - 10% stores
 - 13% branches
 - 52% R-type

$$\text{Average CPI} = (0.13)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12$$

Multicycle Critical Path



Multicycle Processor Performance

Multicycle critical path:

- **Assumptions:**
 - RF is faster than memory
 - Writing memory is faster than reading memory

$$T_{c_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max(t_{ALU}, t_{mem}) + t_{setup}$$

Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$T_{c_multi} = t_{pcq} + t_{\text{dec}} + 2t_{\text{mux}} + \max(t_{\text{ALU}}, t_{\text{mem}}) + t_{\text{setup}}$$

=

Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$\begin{aligned}T_{c_multi} &= t_{pcq} + t_{\text{dec}} + 2t_{\text{mux}} + \max(t_{\text{ALU}}, t_{\text{mem}}) + t_{\text{setup}} \\&= (40 + 25 + 2*30 + 200 + 50) \text{ ps} = \mathbf{375 \text{ ps}}\end{aligned}$$

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c_multi} = 375 ps

Execution Time = (# instructions) \times CPI $\times T_c$

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c_multi} = 375 ps

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(4.12)(375 \times 10^{-12}) \\ &= \mathbf{155 \text{ seconds}}\end{aligned}$$

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- CPI = 4.12 cycles/instruction
- Clock cycle time: $T_{c_multi} = 375 \text{ ps}$

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(4.12)(375 \times 10^{-12}) \\ &= \mathbf{155 \text{ seconds}}\end{aligned}$$

This is **slower** than the single-cycle processor (75 sec.)

Chapter 7: Microarchitecture

Pipelined RISC-V Processor

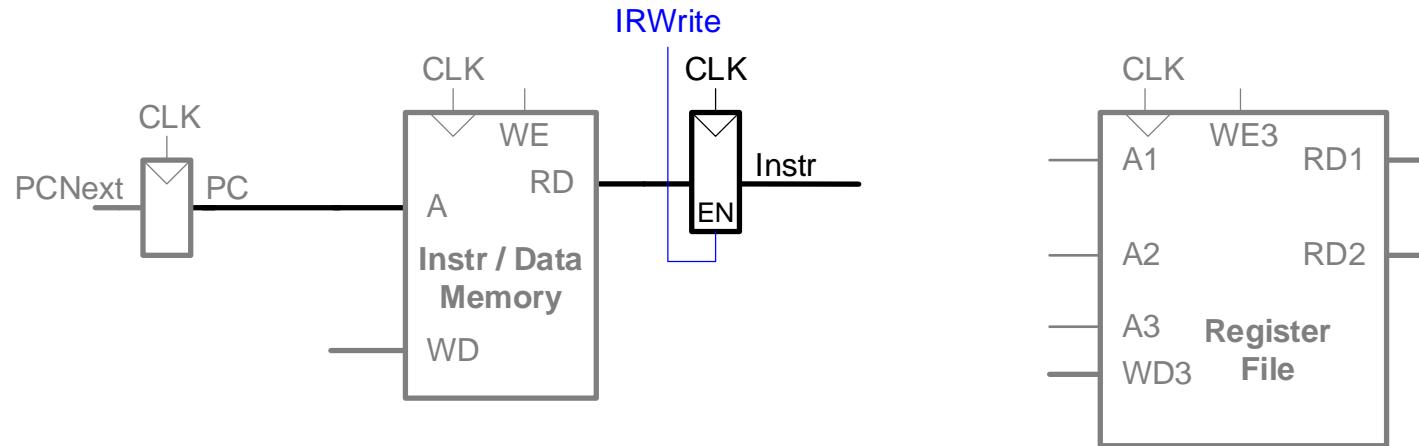
Pipelined RISC-V Processor

- **Temporal parallelism**
- Divide single-cycle processor into **5 stages**:
 - Fetch
 - Decode
 - Execute
 - Memory
 - Writeback
- Add **pipeline registers** between stages
 - Thus, five instructions can execute simultaneously, one in each state
 - Because each state has only 1/5 of the entire logic, the clock frequency can be approximately 5 times faster

Similar to the five steps that the multicycle processor used to execute the `lw` instruction

Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction

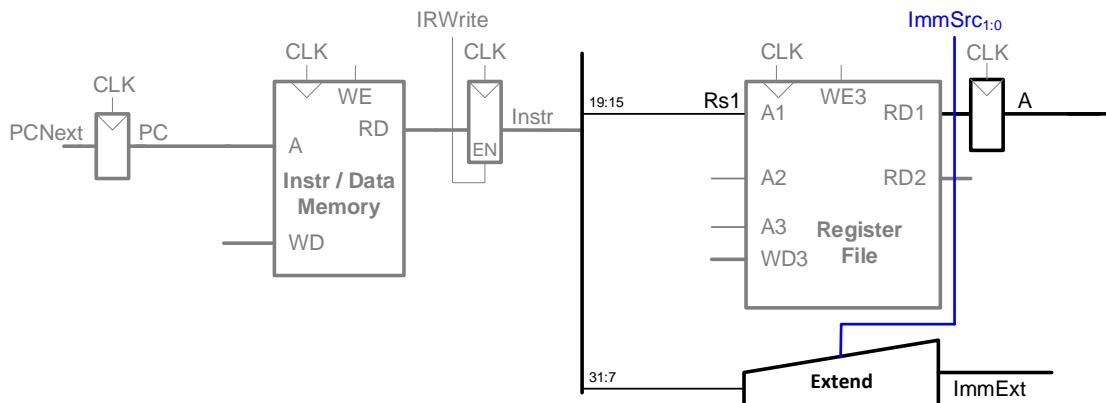


- As with the single-cycle processor, gradually build the data path by adding components to handle each instruction step.
- The PC contains the address of the instruction (l_w) to execute.
- The first step is to read this instruction from memory.
 - The PC is connected to the address input of the memory
 - The instruction is read and stored in a new non-architectural instruction register (IR), so it is available for future cycles.
 - The IR write-enable signal, *IRWrite*, is asserted when the IR should be loaded with a new instruction.

Multicycle Datapath: lw Get Sources

STEP 2: (Decode and Read Reg)Read source operand from RF and extend immediate

First, build the datapath for the lw instruction.

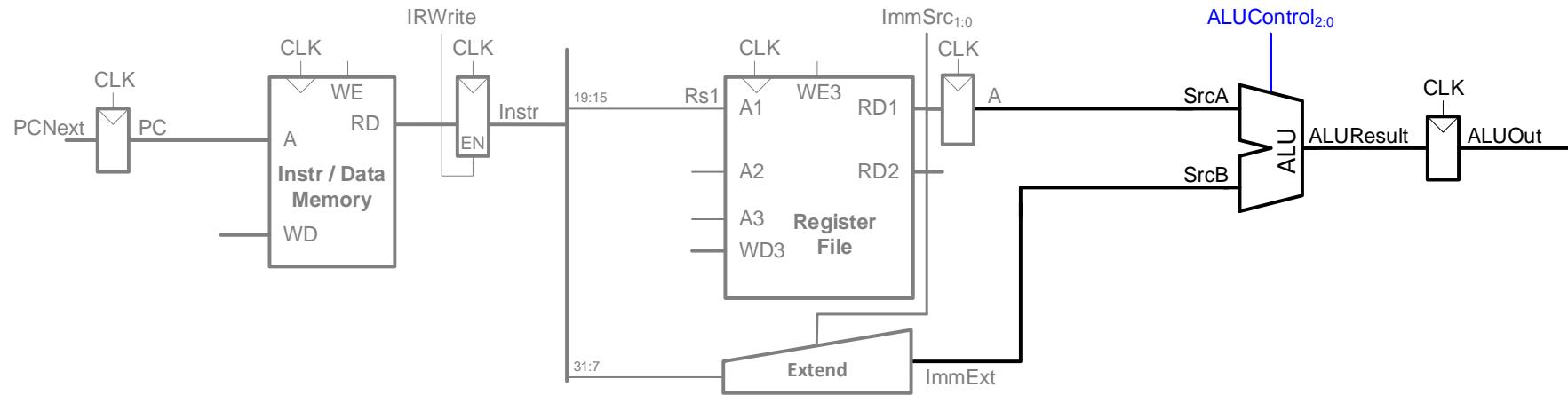


- After fetching lw , the 2nd step is to read the source register containing the base address.
- The source register is specified in the **rs1** field, $Instr_{19:15}$.
- The $Instr_{19:15}$ bits are connected to the Register file input address A1.
- The Register File reads the register onto RD1.
- The value from RD1 is stored in another non-architectural register, A.
- The lw instruction also requires a 12-bit offset in the immediate field of the instruction, $Instr_{31:20}$, which must be sign-extended to 32 bits.
- The Extend unit takes a 2-bit **ImmSrc** control signal to specify a 12-, 13-, or 21-bit immediate to extend for various types of instructions.
- The **ImmExt** is a combinational function of **Instr** and will not change while the current instruction is being processed, so no need to have a non-architectural register to hold the constant value.

Address	Instruction	Type	Fields	Machine Language
0x1000	L7: lw x6, -4 (x9)	I	$imm_{11:0}$ rs1 f3 rd op	1111111111100 01001 010 00110 0000011 FFC4A303

Multicycle Datapath: lw Address

STEP 3: (Execute ALU) Compute the memory address

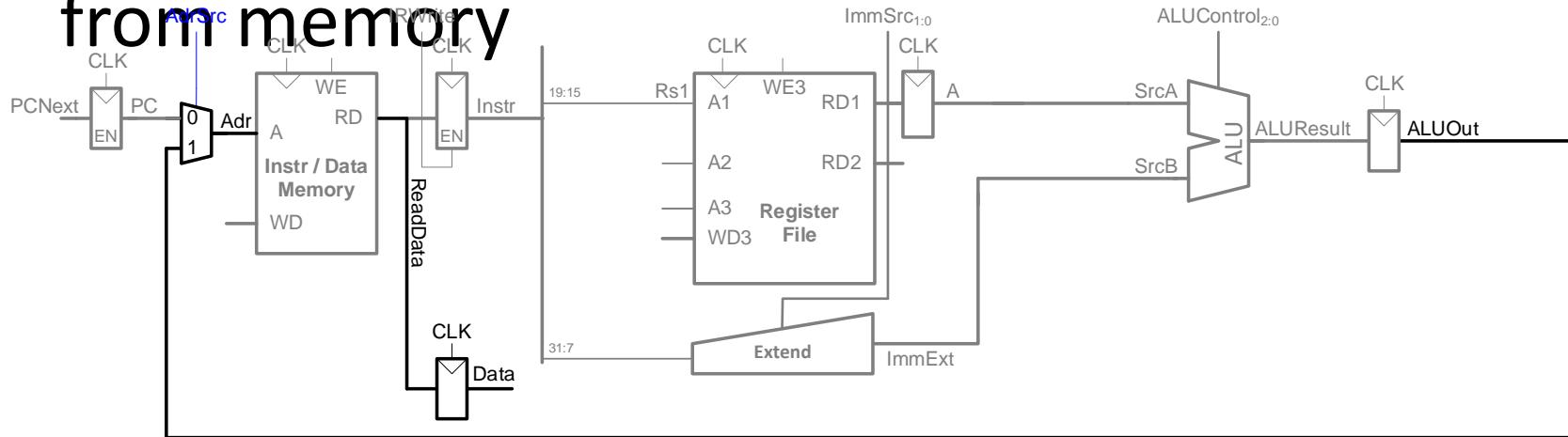


- In the third step, the ALU computes the memory address for the load instruction by adding the base address Rs_1 and the offset.
- The $ALUControl_{2:0}$ should be set to 000 to perform the addition.
- $ALUResult$ is stored in a non-architectural register, $ALUout$.

Address	Instruction	Type	Fields	Machine Language
0x1000	L7: lw x6, -4(x9)	I	imm _{11:0} rs1 f3 rd op	111111111110 01001 010 00110 0000011 FFC4A303

Multicycle Datapath: lw Memory Read

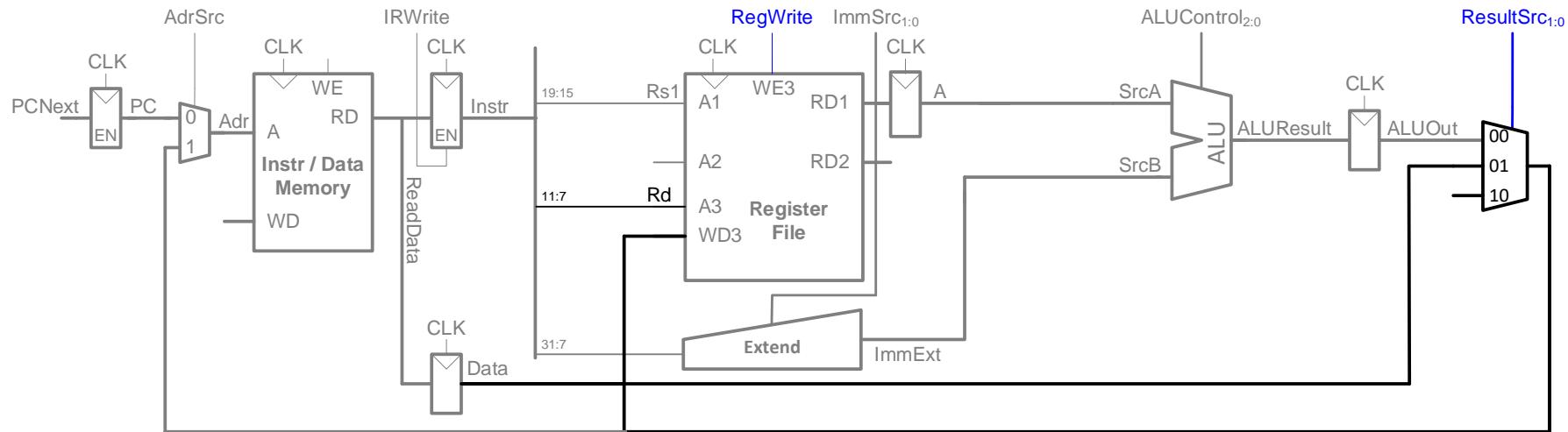
STEP 4: (Memory Read/Write) Read data from memory



- The fourth step is to load the data from the calculated address in the memory.
- We add a mux in front of the memory to choose the memory address, Adr , from either the PC or $ALUOut$ based on the $AdrSrc$ select signal.
- The data read from the memory is stored in another non-architectural register, $Data$.
- The address(Adr) multiplexer permits us to reuse the memory unit during the lw instruction.
 - In the first step, the address is taken from the PC to fetch the instruction.
 - In the fourth step, the address is taken from $ALUOut$ to load the data.
 - Hence, $AdrSrc$ must have different values during different steps of a single instruction.
 - We will develop a Finite-State-Machine (FSM) controller to generate these sequences of control signals.

Multicycle Datapath: 1w Write Register

STEP 5: (Write Reg) Write data back to register file



- Finally, the data is written back to the register file.
- The destination register is specified by the **rd** field of the instruction, $Instr_{11:7}$.
- The result comes from the non-architectural Data register.
 - Instead of connecting the Data register directly to the register file's **WD3** write port, we add a multiplexer on the Result bus to choose either **ALUOut** or **Data** before feeding Result back to the register file's write-data port (**WD3**).
 - This will be helpful because other instructions will need to write a result from the ALU to the register file.
 - The **RegWrite** signal is set to 1 to indicate that the register file should be updated.

Throughput vs Lanency

- **Throughput** is the number of instructions completed per second.
- **Latency** is the time it takes for a given instruction to complete, from start to finish.
- Ideally, the latency of each instruction is unchanged, but the throughput is five times faster
- Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency.
- Pipelining introduces some overhead, so the throughput will not be as high as the ideal desire
- Modern high-performance microprocessors are pipelined.

Pipelined RISC-V Processor

- **Temporal parallelism**

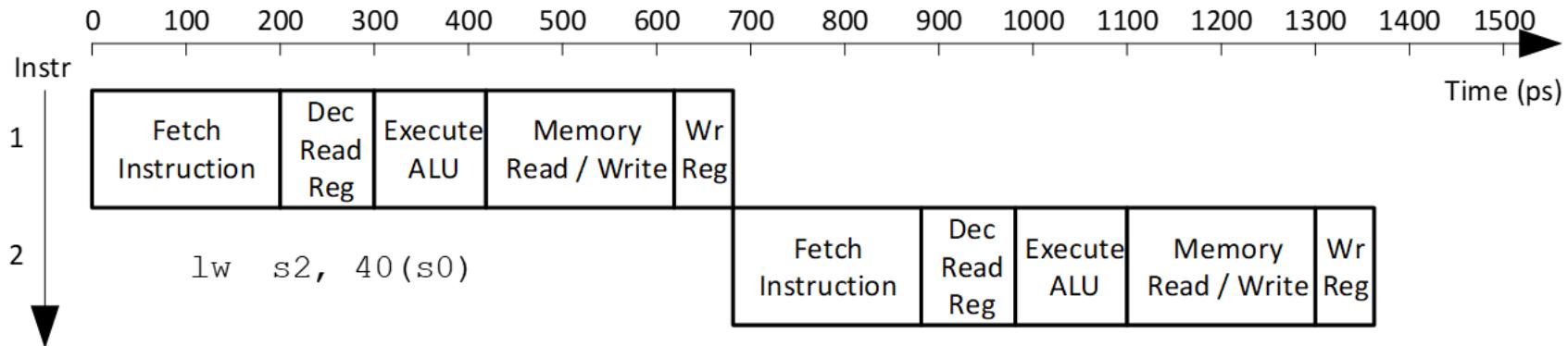
- Executing multiple instructions at different stages of a pipeline simultaneously.
- Allows a processor to initiate a new instruction at each clock cycle, with each instruction moving through various pipeline stages independently.
- Increases throughput, as multiple instructions are "in flight" within the pipeline at any given moment, improving the processor's overall efficiency and speed.

- **Add pipeline registers between stages**

- To hold intermediate results, thus multiple instructions can execute simultaneously, one in each state.
- Because each state has shorter combinational logic, the clock frequency can be faster than the single-cycle processor.

Single-Cycle vs. Pipelined Processor

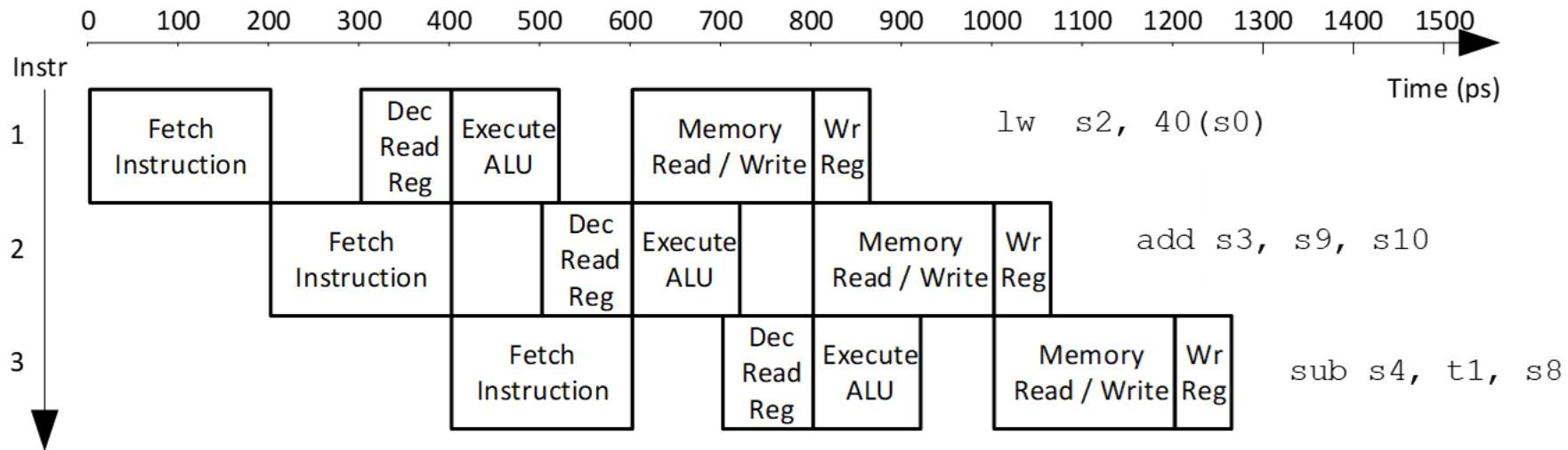
Single-Cycle



1. The first instruction is read from memory at time 0. (200 ps)
2. Next, the operands are read from the register file. (100 ps)
3. Then, the ALU executes the necessary computation. (120 ps)
4. Finally, the data memory may be accessed. (200 ps)
5. The result is written back to the register file at 680 ps. (60 ps)
 - The second instruction begins when the first completes.
 - Hence, in this diagram, the single-cycle processor has an instruction **latency** of $200 + 100 + 120 + 200 + 60 = 680 \text{ ps}$ (see [Table 7.7](#) on page 415), and
 - a **throughput** of **1 instruction per 680 ps** (1.47 billion instructions per second).

Single-Cycle vs. Pipelined Processor

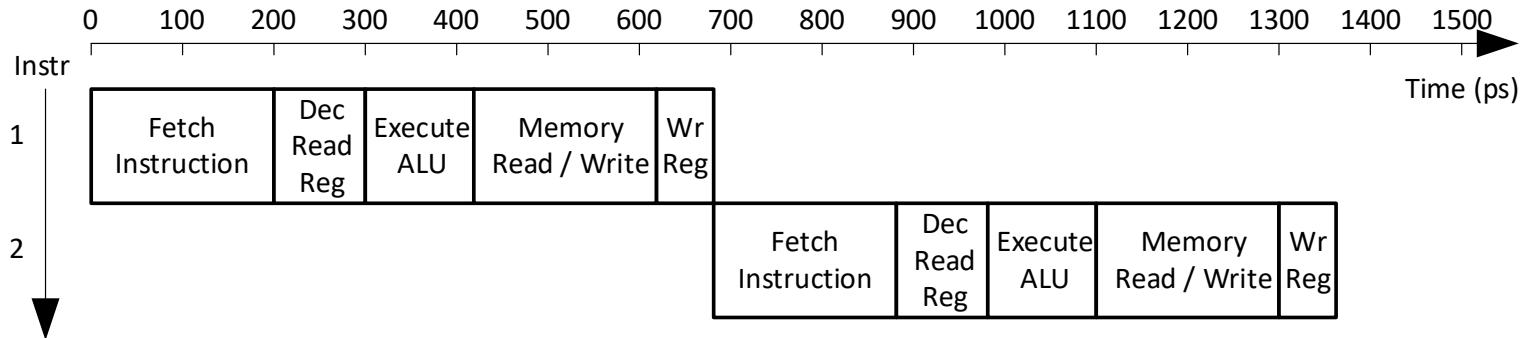
Pipelined



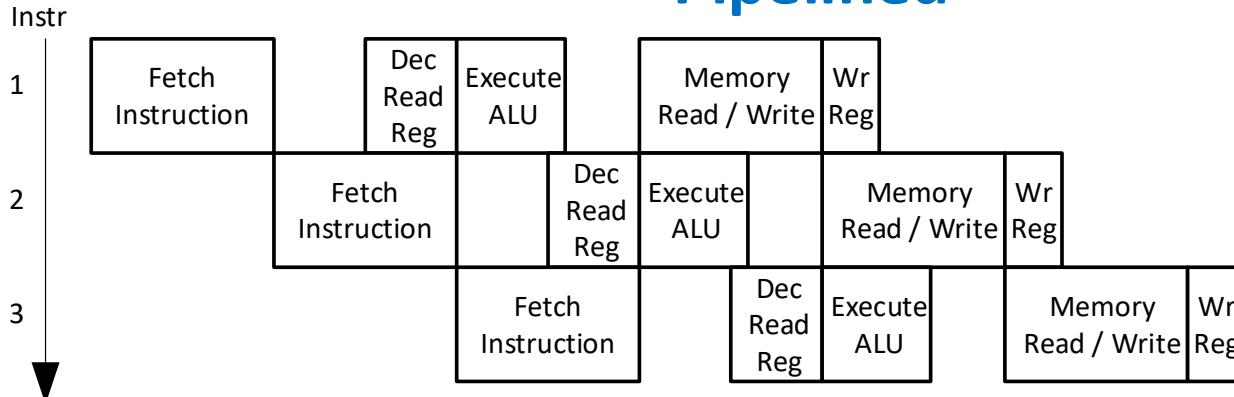
- The length of a pipeline stage is set at 200 ps by the slowest stage, the fetch instruction from memory or Memory Read/Write.
- At time 0, the 1st instruction is **fetched** from memory.
- At 200 ps, the 1st instruction enters the **Decode** stage, and a 2nd instruction is **fetched**.
- At 400 ps, the 1st instruction **executes**, the 2nd instruction enters the **Decode** state, and a 3rd instruction is **fetched**.
- And so forth, until all the instructions are complete.
- The **Latency** is $5 \times 200 \text{ ps} = 1000 \text{ ps}$, longer than the single-cycle processor.
- The **throughput** is **1 instruction per 200 ps** (5 billion instructions per second), i.e. one instruction completes every clock cycle.
- The throughput is 3.4 ($680 \text{ ps} / 200 \text{ ps}$) times as much as the single-cycle processor (not quite 5 times, but still a substantial speedup).

Single-Cycle vs. Pipelined Processor

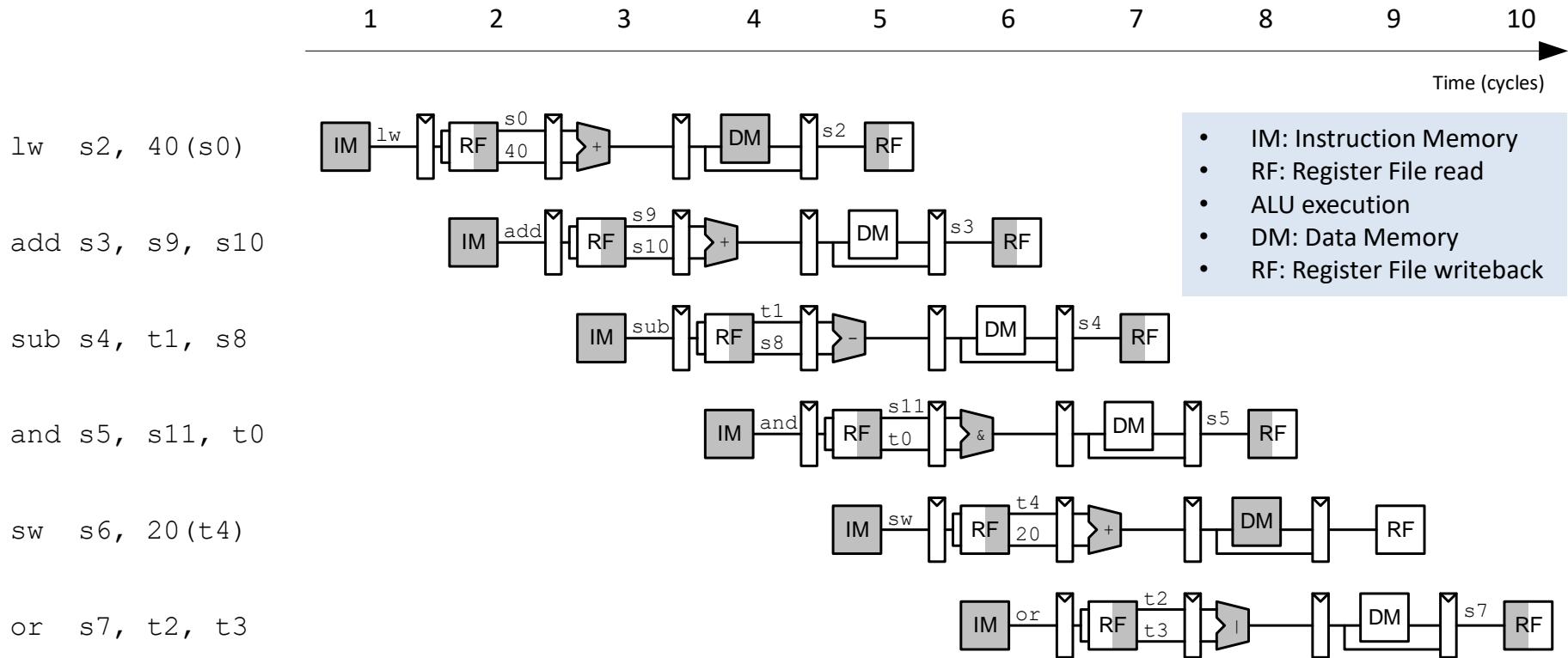
Single-Cycle



Pipelined

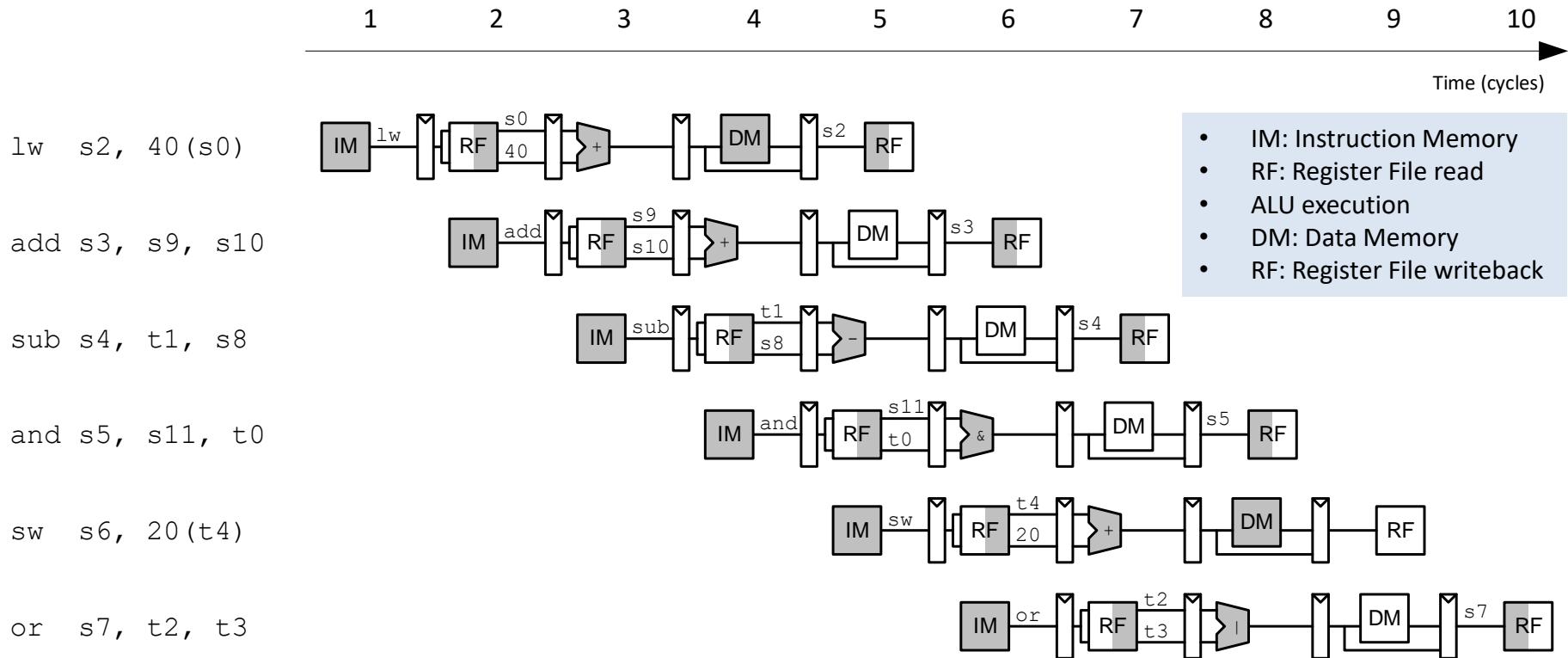


Pipelined Processor Abstraction



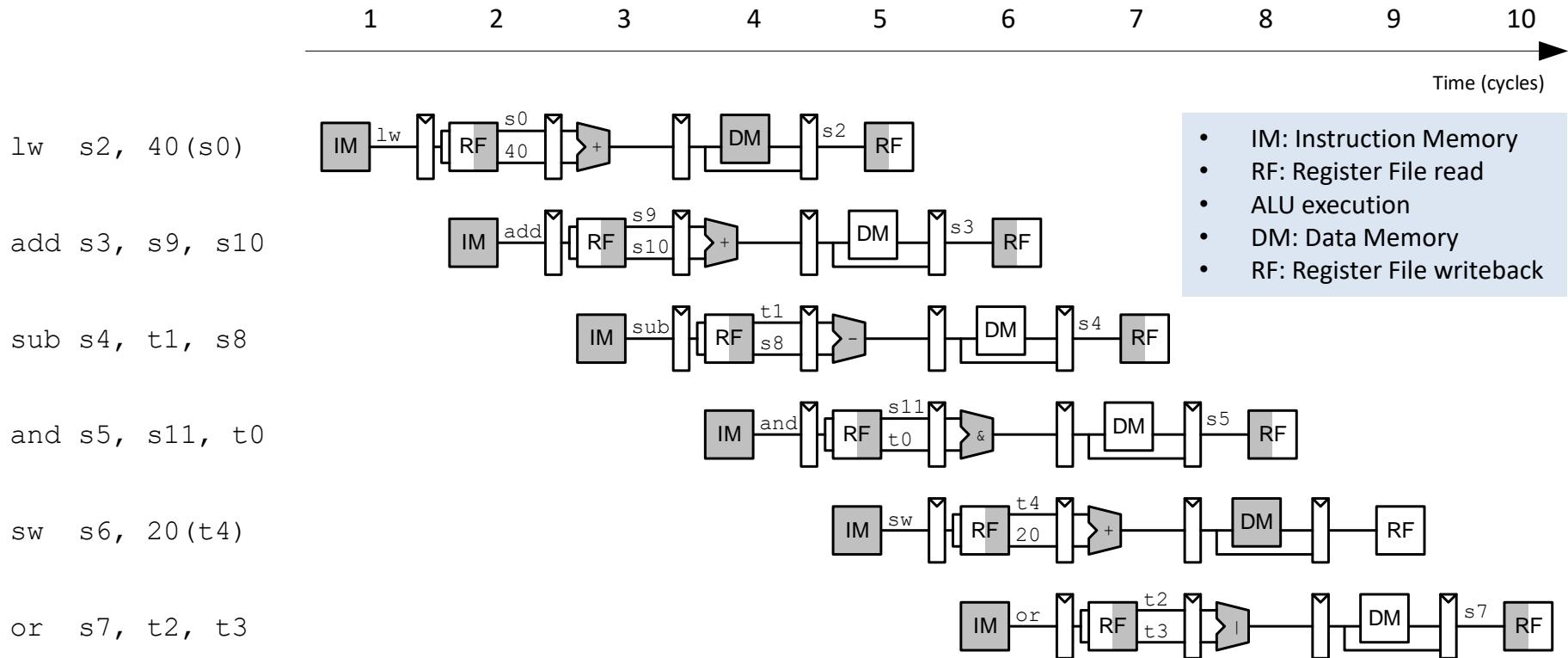
- Reading across a row shows the clock cycle in which a particular instruction is in each stage. For example,
- the sub instruction is fetched in cycle 3 and executed in cycle 5.

Pipelined Processor Abstraction



- Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example,
- in cycle 6, the register file is writing a sum to s3, the data memory is idle, the ALU is computing ($s_{11} \& t_0$), t_4 is being read from the register file, and the `or` instruction is being fetched from the instruction memory.

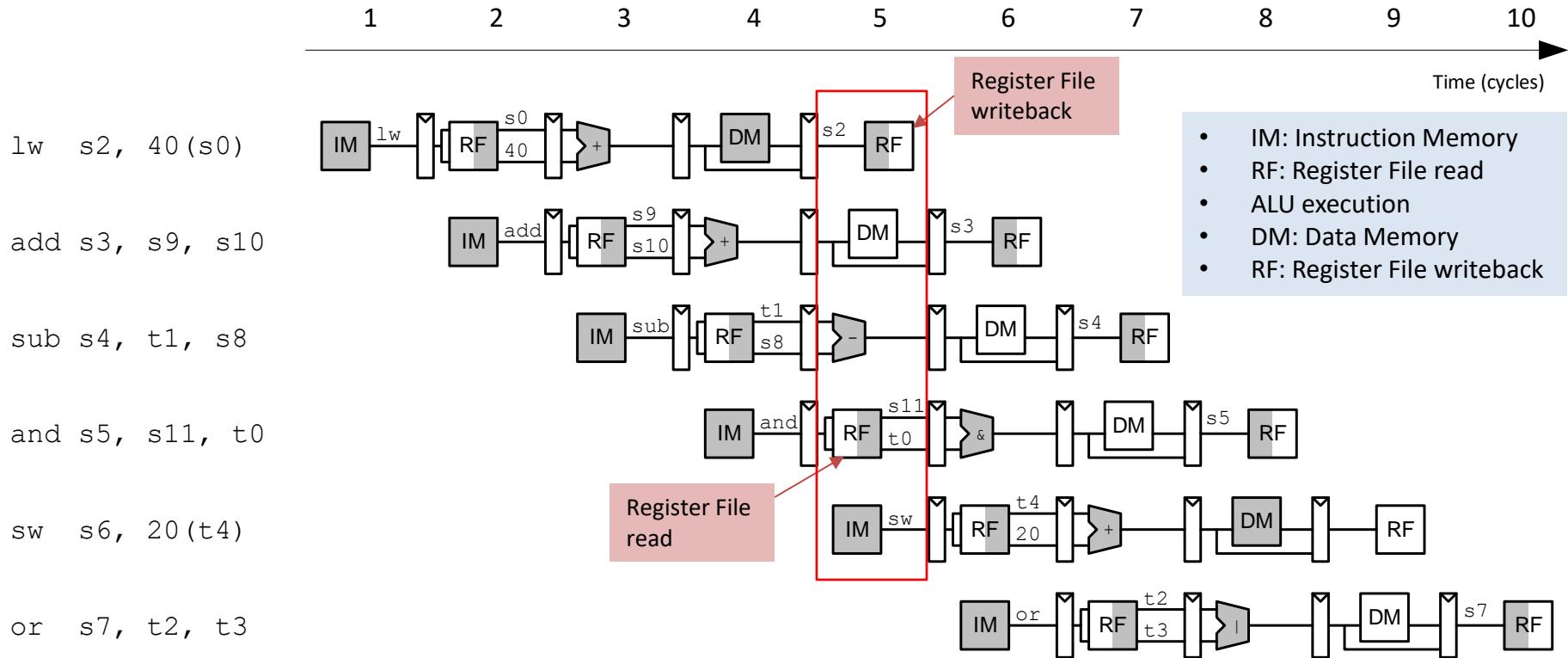
Pipelined Processor Abstraction



Stages are shaded to indicate when they are used. For example,

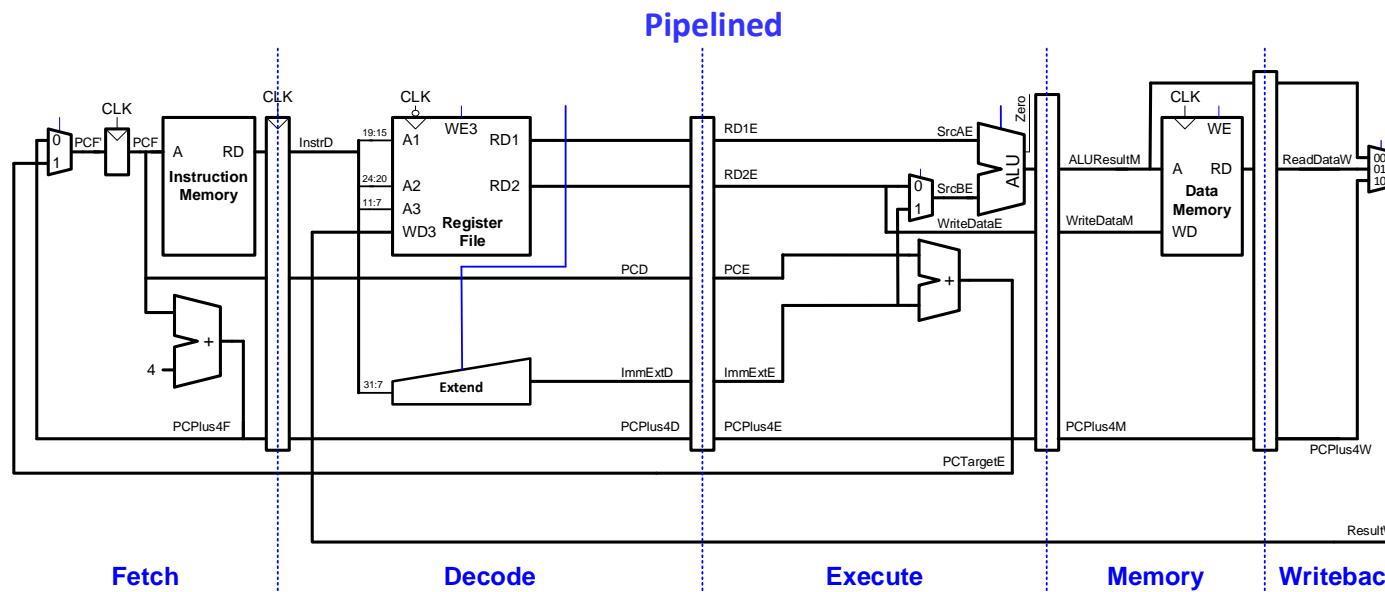
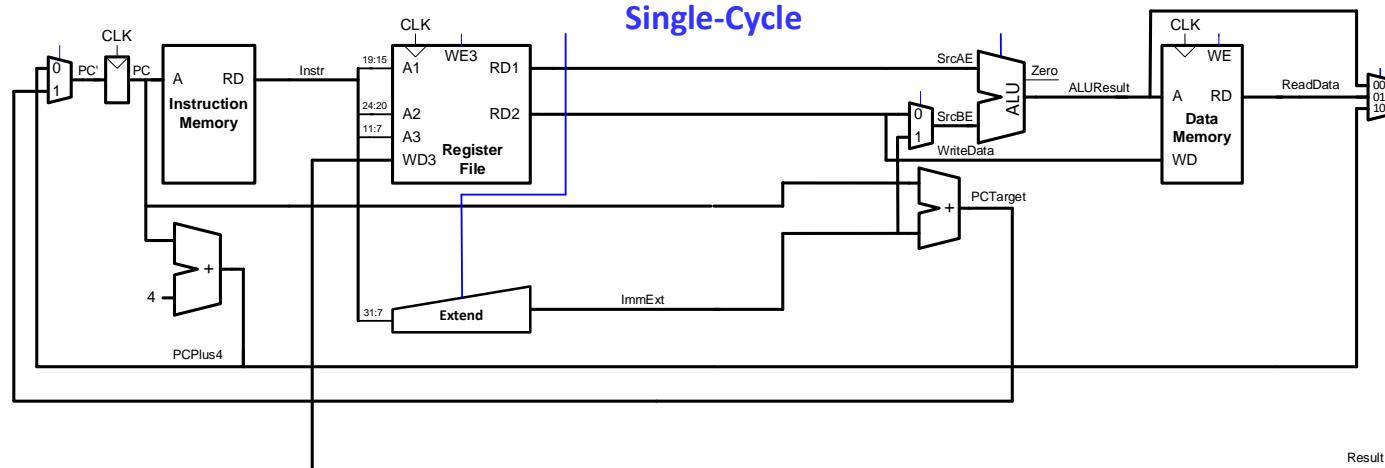
- the data memory is used by `lw` in cycle 4 and by `sw` in cycle 8.
- The instruction memory and ALU are used in every cycle.
- The register file is written by every instruction except `sw`.

Pipelined Processor Abstraction



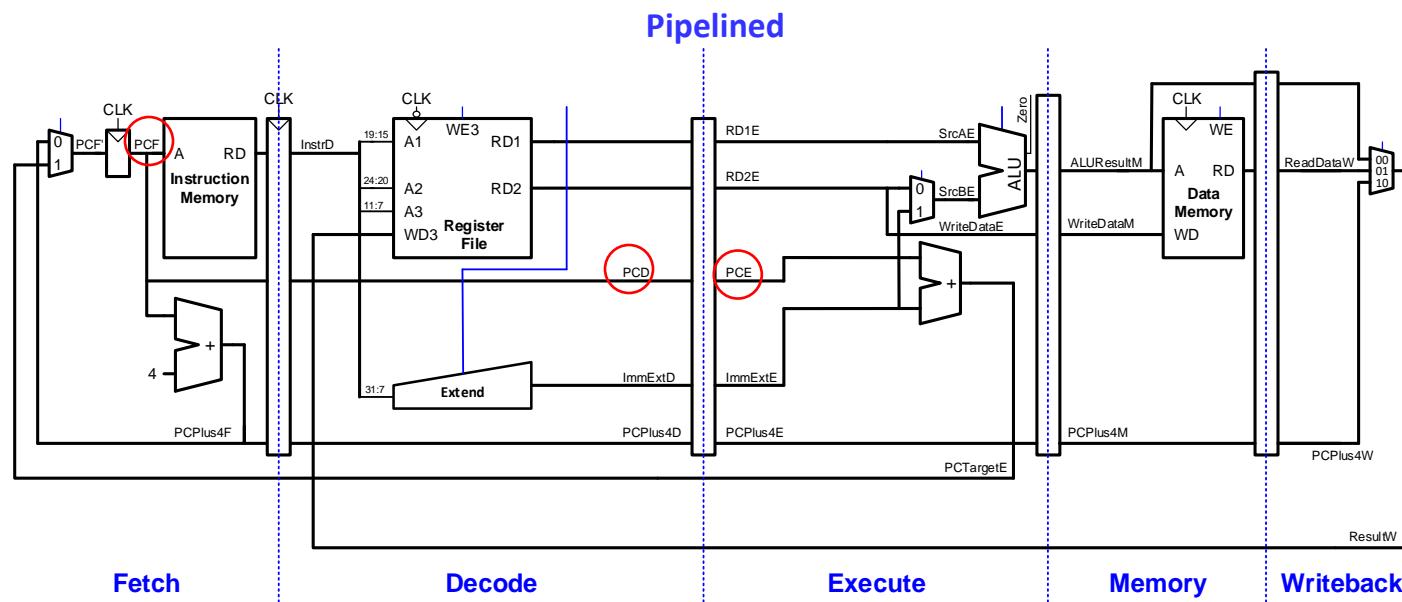
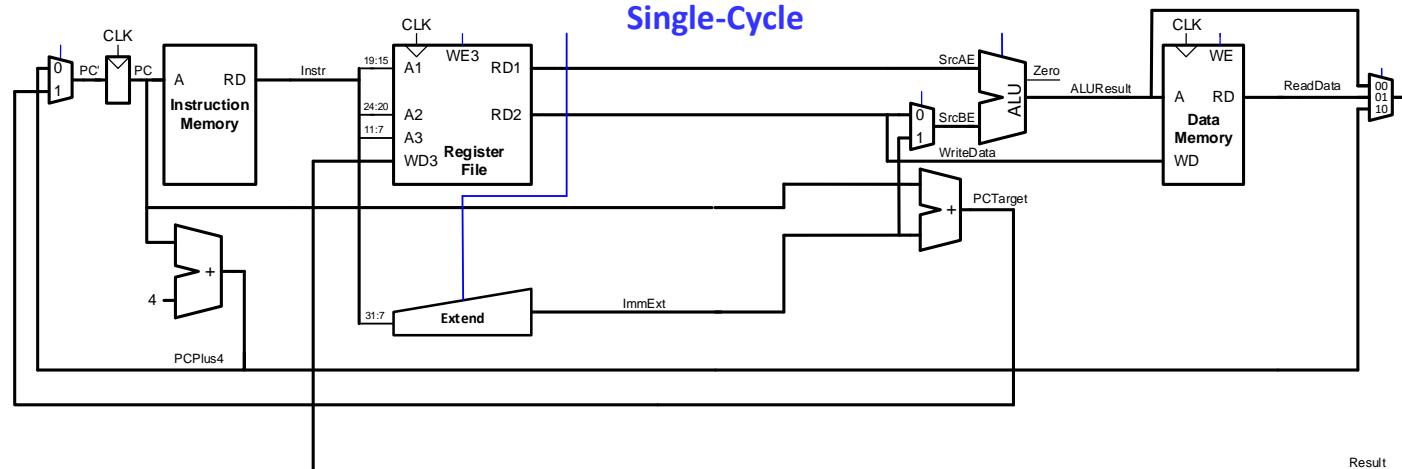
- In the pipelined processor, the register file (RF) is used twice in every cycle:
 - it is written in the first part of a cycle and
 - read in the second part, as suggested by the shading.
- So, data can be written by one instruction and read by another within a single cycle.

Single-Cycle & Pipelined Datapaths



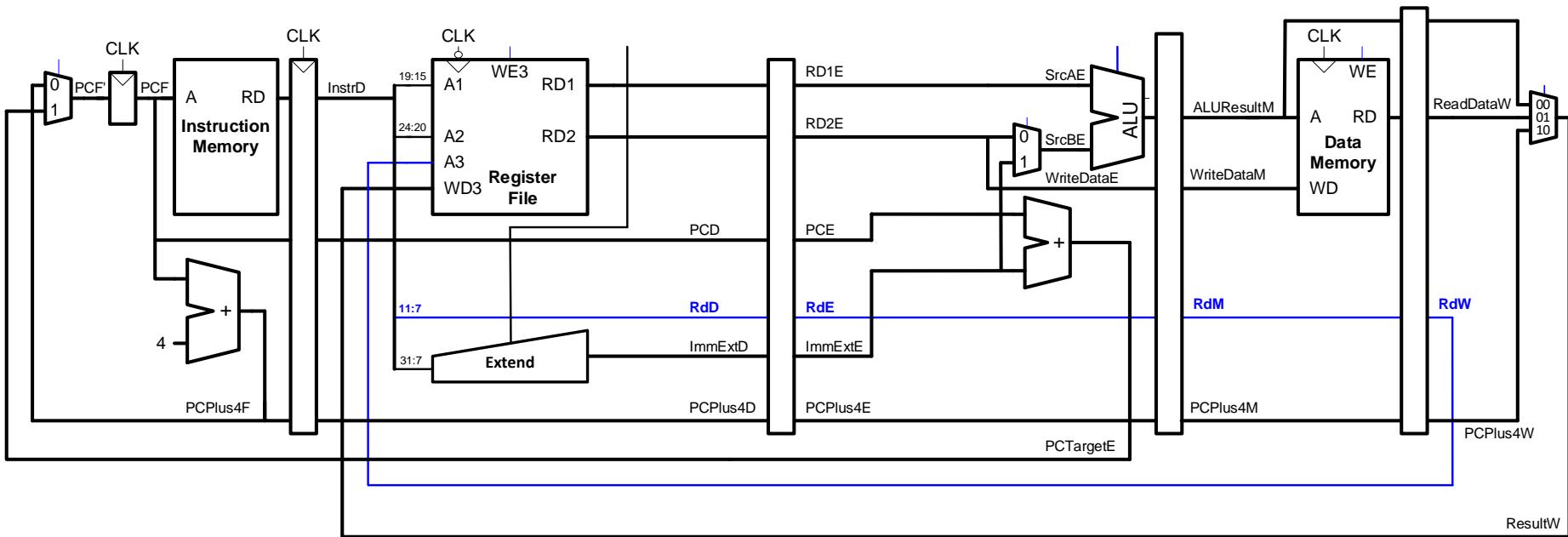
The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers.

Single-Cycle & Pipelined Datapaths



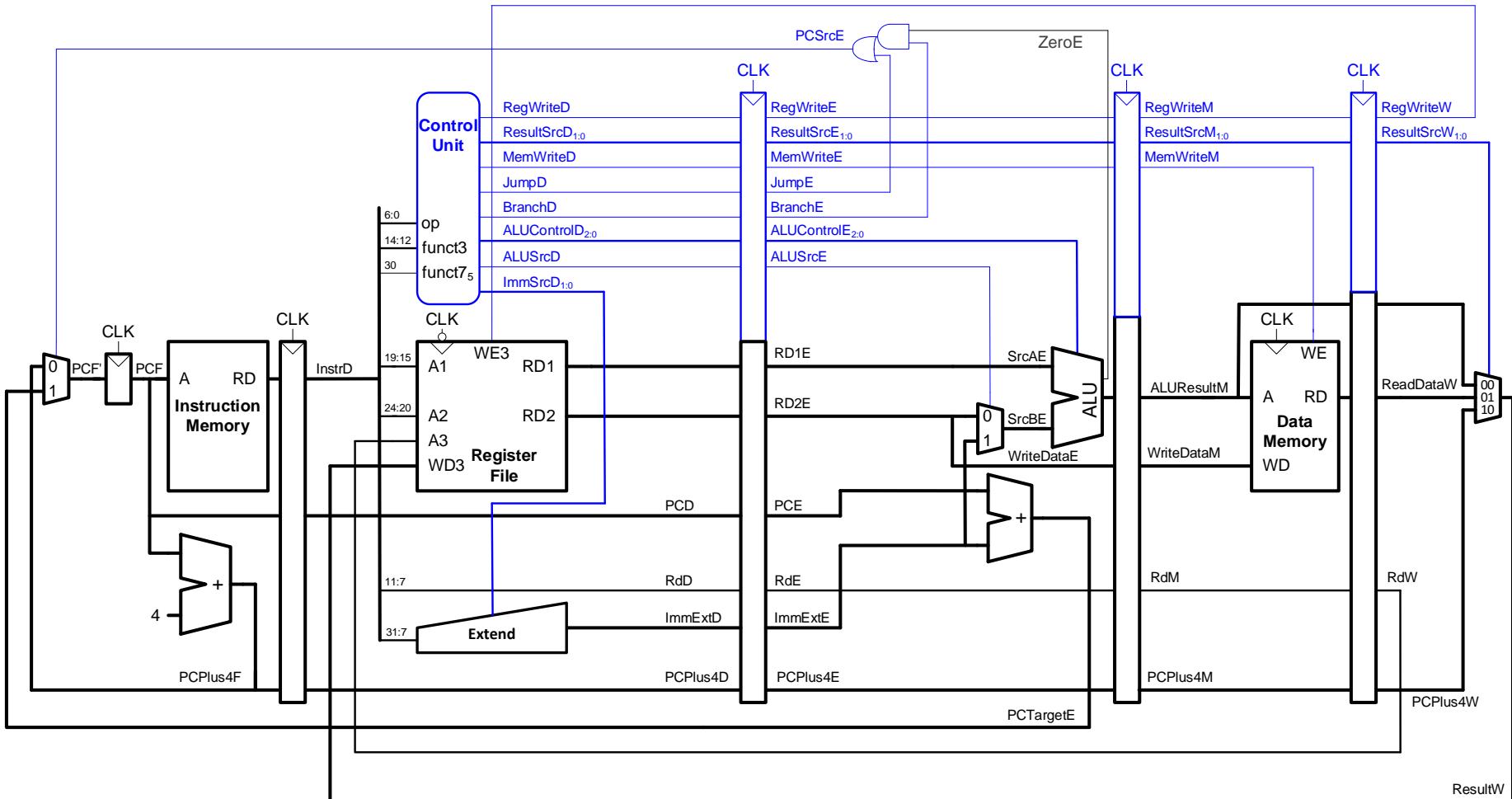
Signals in Pipelined Processor are appended with first letter of stage (i.e., PC_F, PC_D, PC_E).

Corrected Pipelined Datapath



- **Rd** must arrive at same time as **Result**
- Register file written on **falling edge** of **CLK**

Pipelined Processor with Control



- Same control unit as single-cycle processor
- Control signals travel with the instruction (drop off when used)

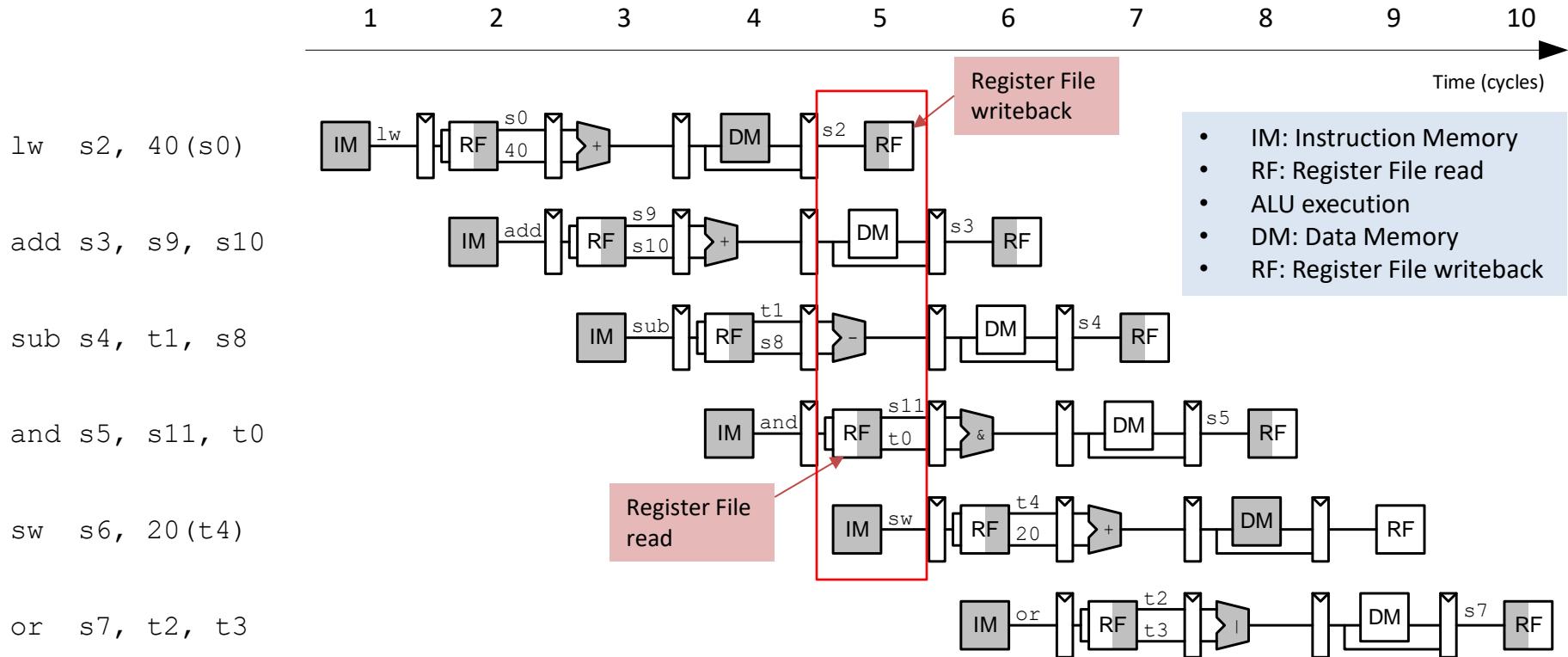
Chapter 7: Microarchitecture

Pipelined Processor Hazards

Pipelined Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
 - **Data hazard:** register value not yet written back to register file
 - **Control hazard:** next instruction not decided yet (caused by branch)

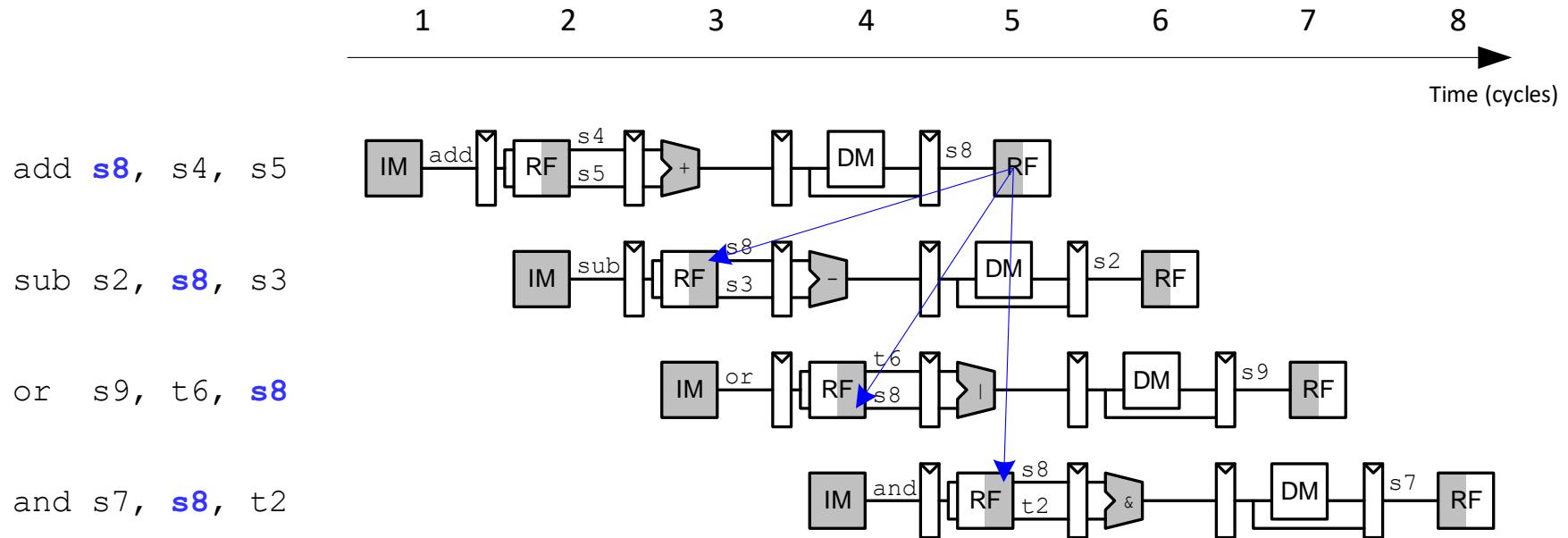
Pipelined Processor Data Hazards



A key challenge in pipelined systems is handling **hazards**, e.g. when a later instruction needs the result of an earlier one before it's complete.

- If the `add` used `s2` as a source instead of `s10`, a hazard would occur because the `s2` register has not yet been written by the `lw` instruction when it is read by `add` in cycle 3.

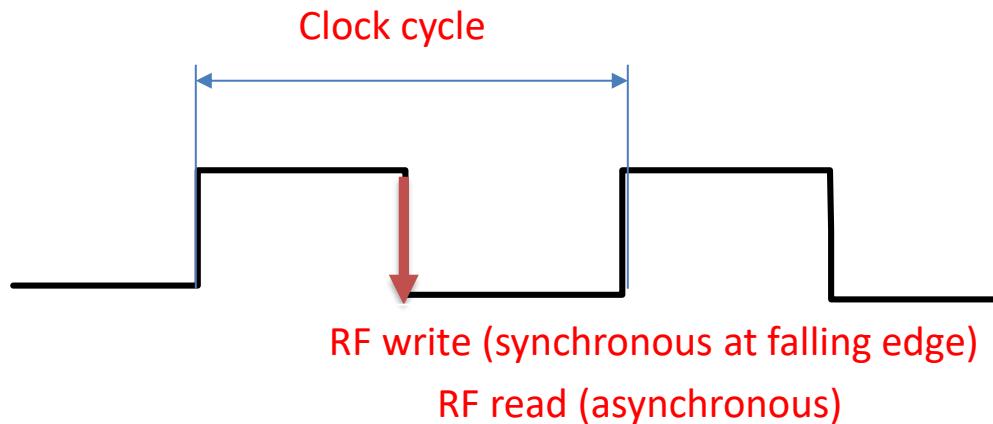
Data Hazard



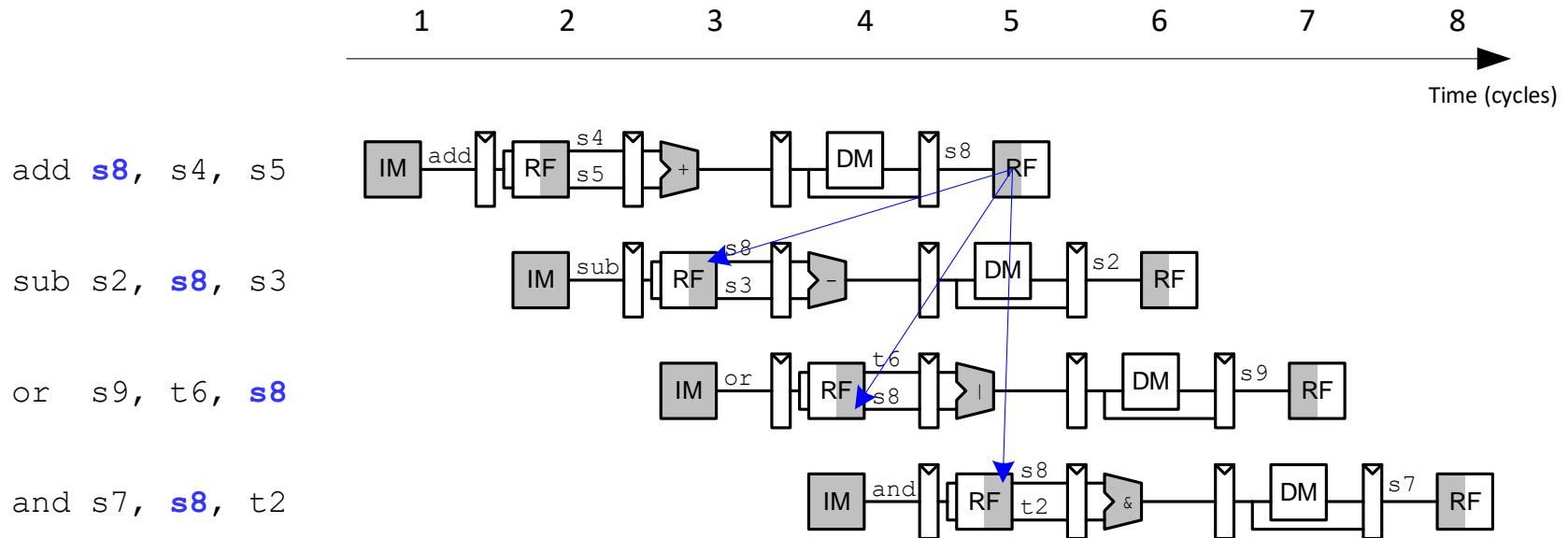
- **Read after write (RAW) hazard:** When one instruction writes a register before the write operation has completed, subsequent instructions read this register

Register File (RF) Write/Read

- RF is written during the first half of the clock cycle at the falling edge.
- RF is read asynchronously during the second half of the clock cycle.
- So a register can be written and read back in the same clock cycle without introducing a hazard.



Read after write (RAW) hazard



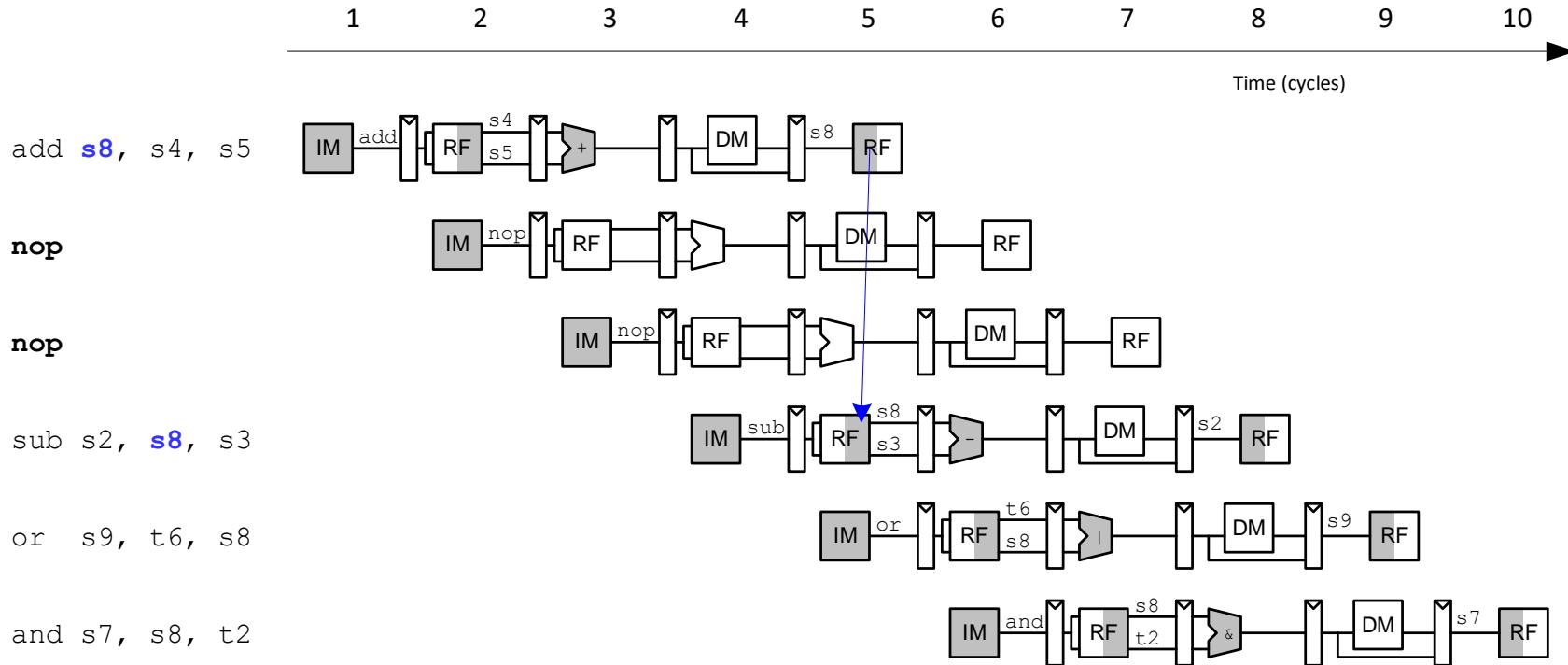
- The `add` instruction writes a result into `s8` in the first half of cycle 5.
- However, the `sub` instruction reads `s8` on cycle 3, obtaining the wrong value.
- The `or` instruction reads `s8` on cycle 4, again obtaining the wrong value.
- The `and` instruction reads `s8` in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5. Subsequent instructions also read the correct value of `s8`.
- The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions reads that register.
- Without special treatment, the pipeline will compute the wrong result.

Handling Data Hazards

- Insert nops in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Handling Data Hazards: Insert nops

- Insert enough **nops** for the result to be ready
- Or move independent useful instructions forward

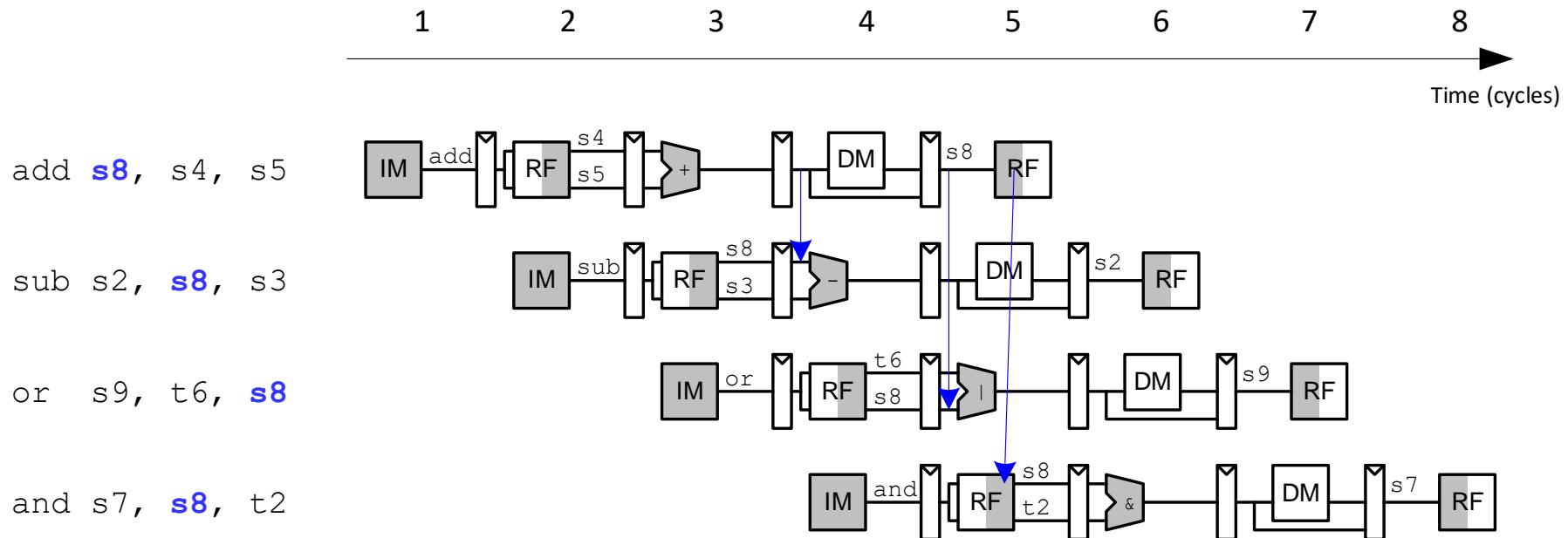


A software solution could insert `nop` instructions between the `add` and the `sub` instruction, ensuring the dependent instruction does not read the result (`s8`) until it is available in the register file.

Such a software interlock complicates programming and degrades performance, not ideal.

Data Forwarding

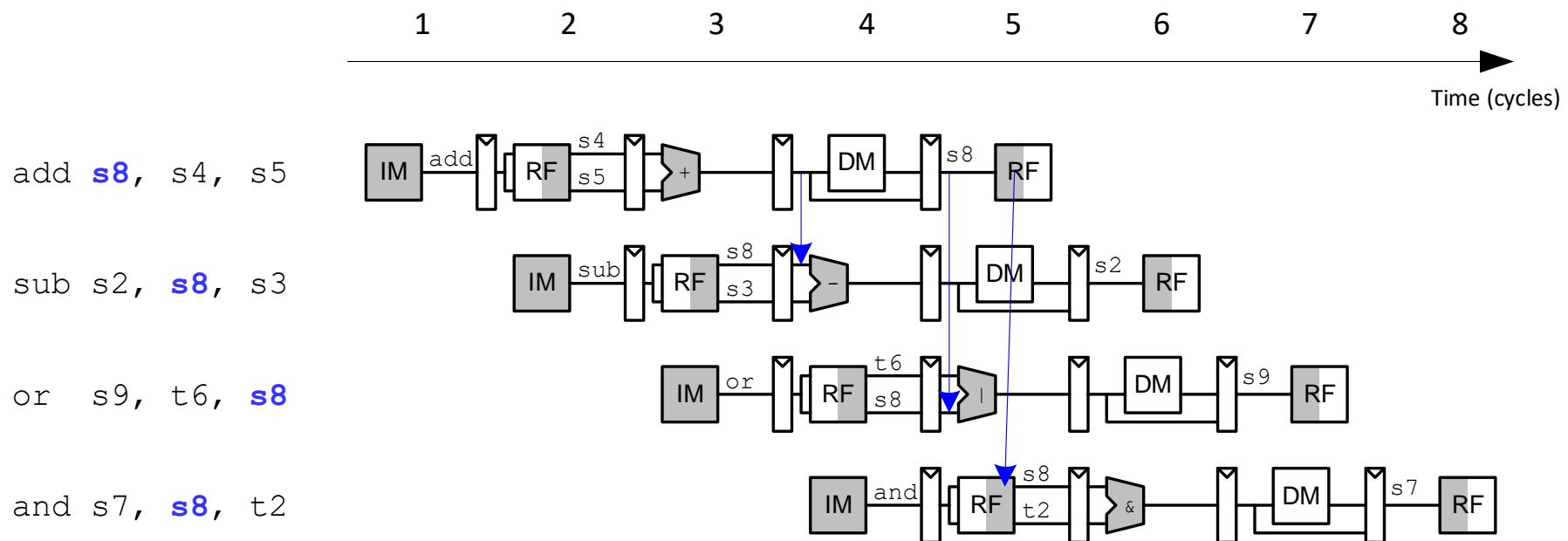
- Data is **available on internal busses** before it is written back to the register file (RF).
- **Forward data from internal busses to Execute stage.**



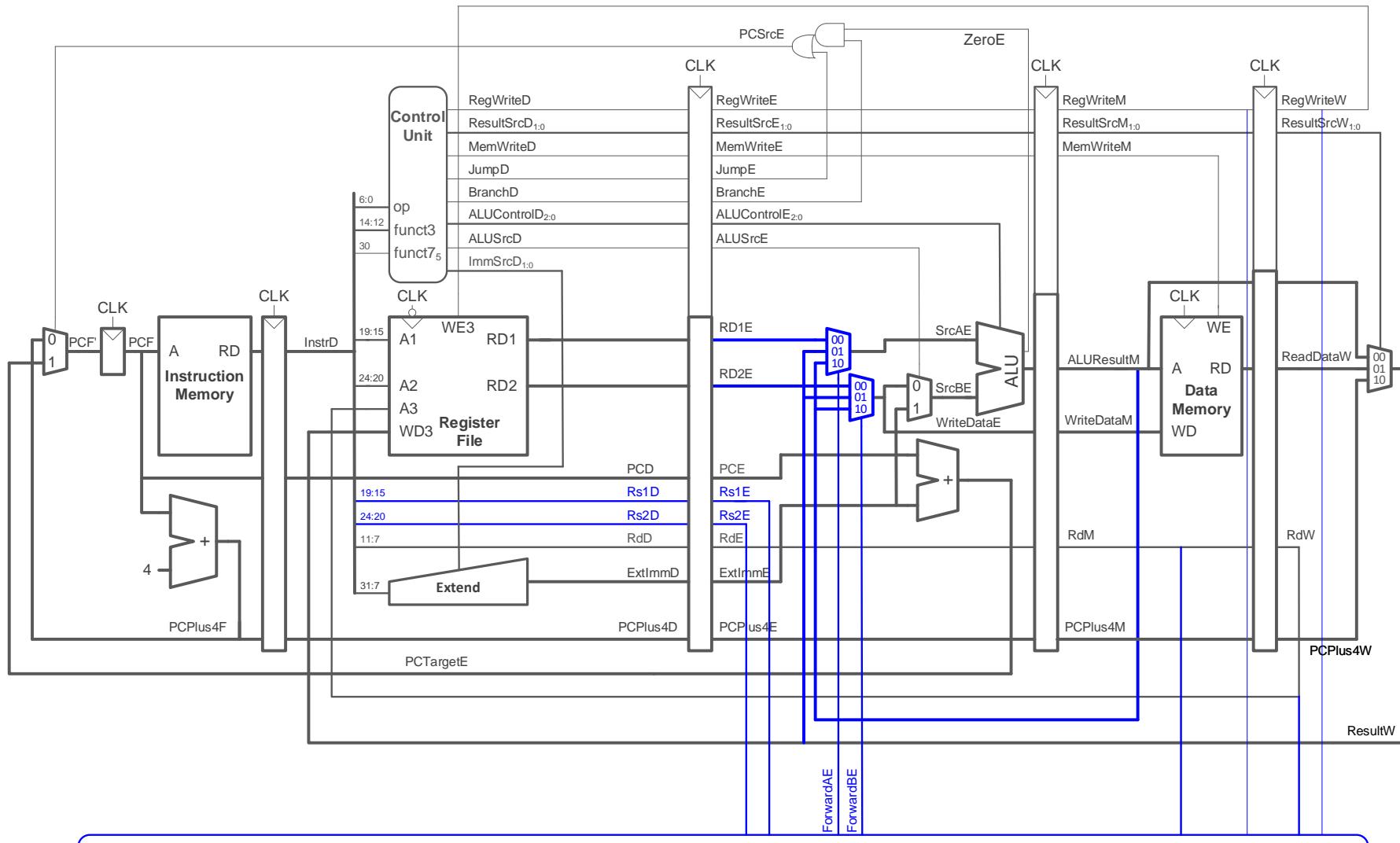
- The sum from the add instruction is computed by the ALU in cycle 3 and is not strictly needed by the and instruction until the ALU uses it in cycle 4.
- In principle, we should be able to forward the result from one instruction to the next to resolve the RAW hazard without waiting for the result to appear in the register file and without slowing down the pipeline.

Data Forwarding

- Check if source register **in Execute stage matches** destination register of instruction **in Memory or Writeback stage**.
- If so, forward result.



Data Forwarding: Hazard Unit



Data Forwarding

- **Case 1:** Execute stage $Rs1$ or $Rs2$ matches **Memory** stage Rd ?
Forward from Memory stage
- **Case 2:** Execute stage $Rs1$ or $Rs2$ matches **Writeback** stage Rd ?
Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for $Rs1$:

```
if      ((Rs1E == RdM) AND RegWriteM)          // Case 1
        ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW)          // Case 2
        ForwardAE = 01
else
        ForwardAE = 00                           // Case 3
```

ForwardBE equations are similar (replace $Rs1E$ with $Rs2E$)

Data Forwarding

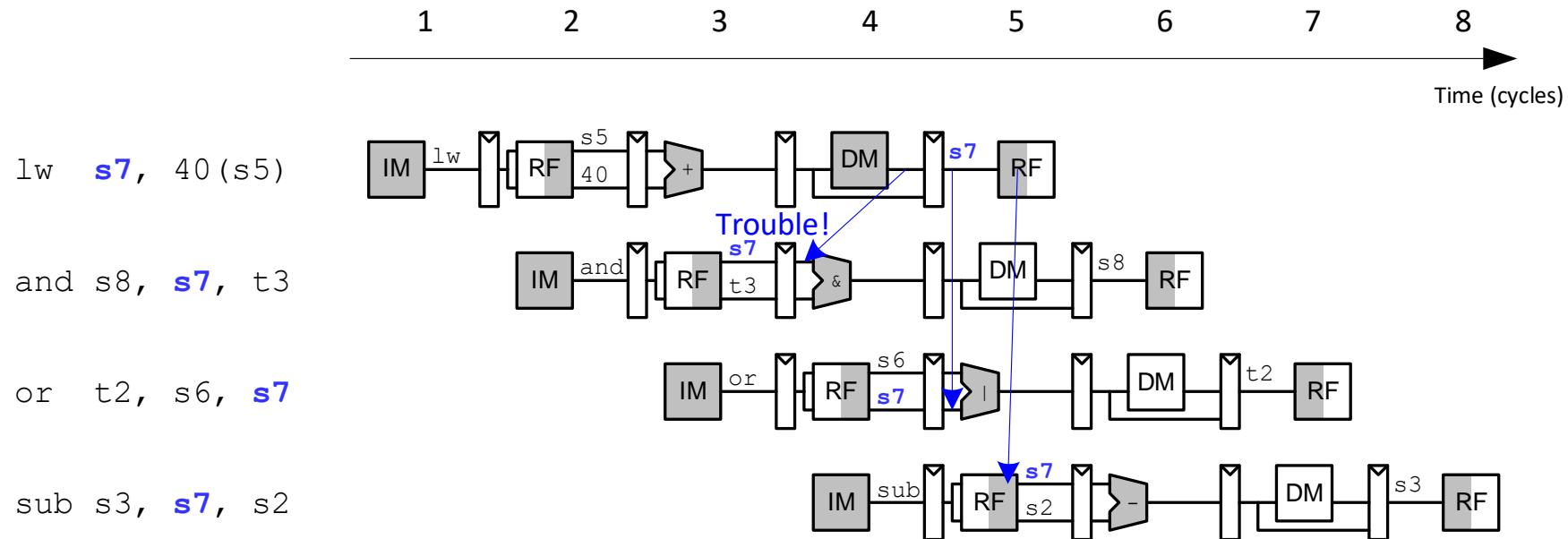
- **Case 1:** Execute stage $Rs1$ or $Rs2$ matches **Memory** stage Rd ?
Forward from Memory stage
- **Case 2:** Execute stage $Rs1$ or $Rs2$ matches **Writeback** stage Rd ?
Forward from Writeback stage
- **Case 3:** Otherwise use value read from register file (as usual)

Equations for $Rs1$:

```
if      ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0) // Case 1  
        ForwardAE = 10  
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0) // Case 2  
        ForwardAE = 01  
else          ForwardAE = 00                                // Case 3
```

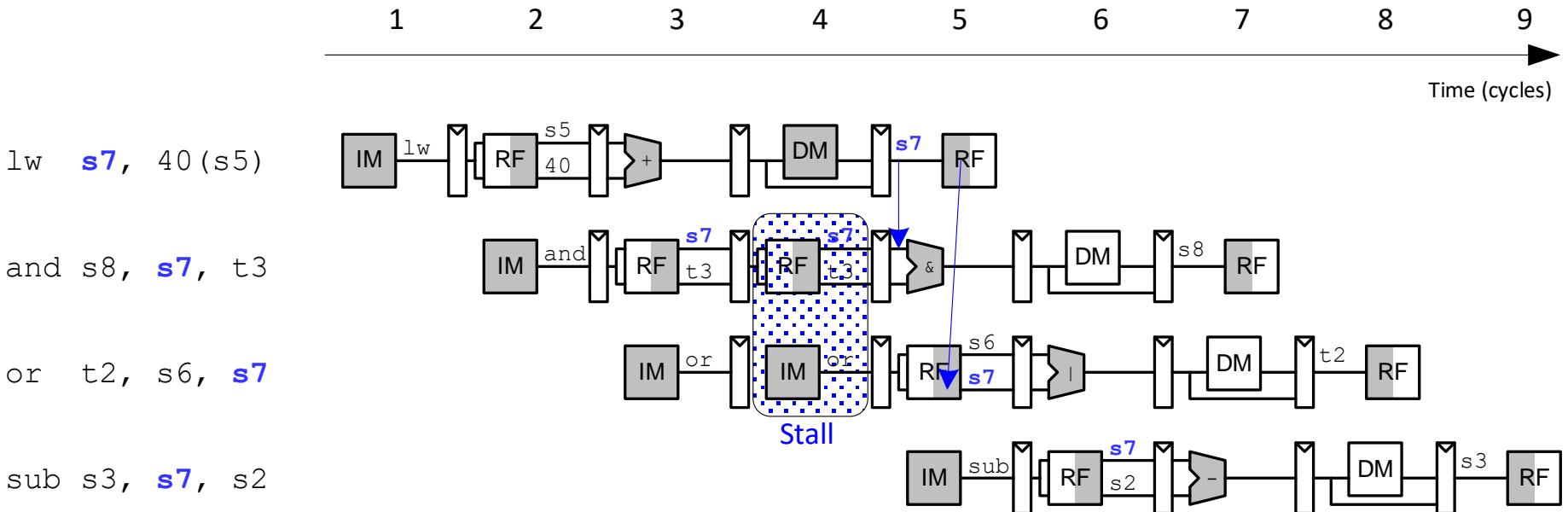
ForwardBE equations are similar (replace $Rs1E$ with $Rs2E$)

Data Hazard due to `lw` Dependency



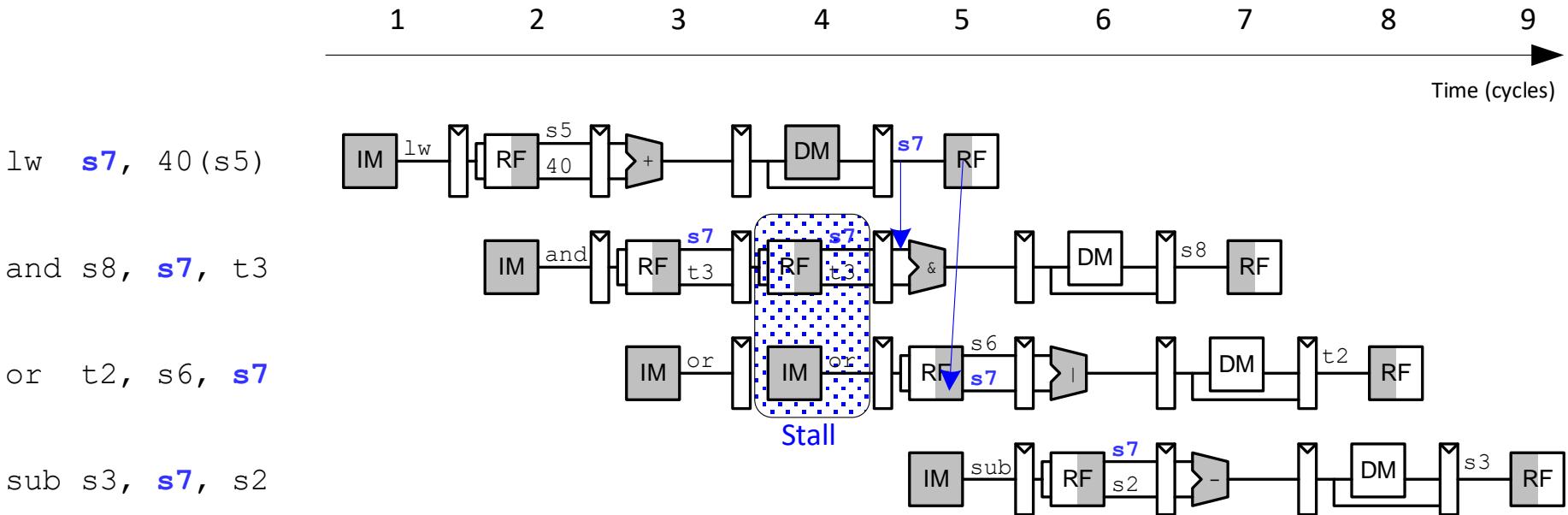
- Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of instruction because its result can then be forwarded to the Execute stage of the next instruction.
- Unfortunately, the `lw` instruction does not finish reading data from memory until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction.
- We say that the `lw` instruction has a ***two-cycle latency*** because a dependent instruction cannot use its result until two cycles later. The above figure shows this problem.
- The `lw` instruction receives data from memory at the end of cycle 4, but the `and` instruction needs that data (the value in `s7`) as a source operand at the beginning of cycle 4. **There is no way to solve this hazard with forwarding.**

Stalling to solve `lw` Data Dependency



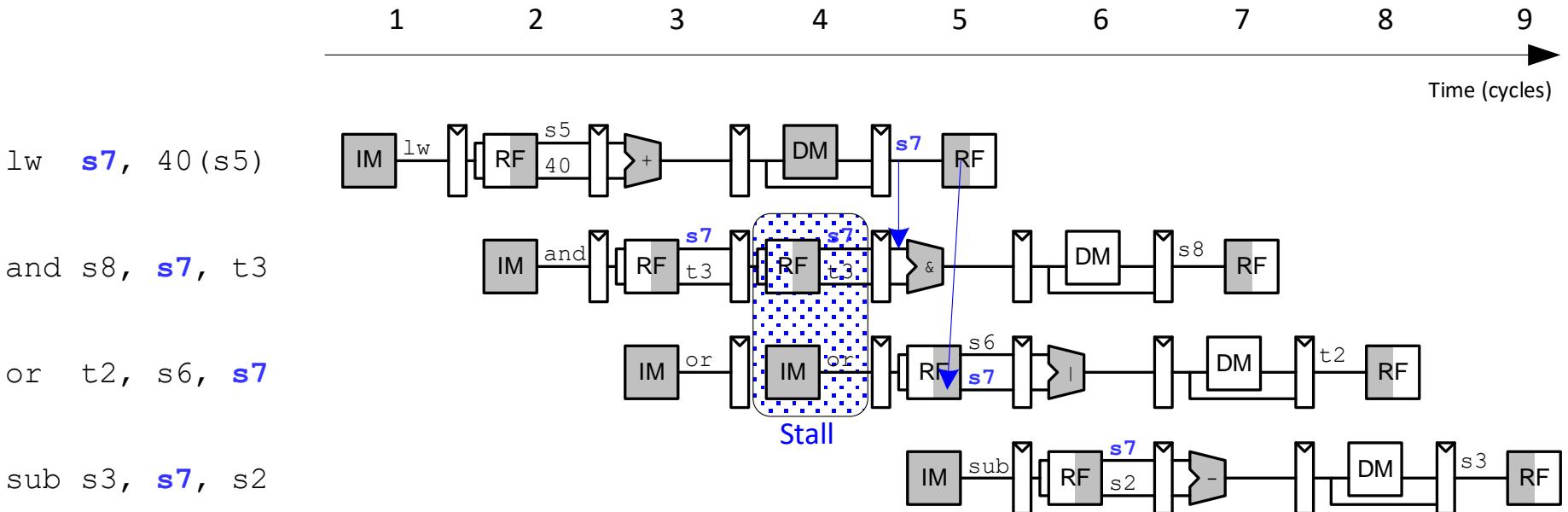
- A solution is to *stall* the pipeline, holding up the operation until the data is available.
- The above Figure shows stalling the dependent instruction (`and`) in the *Decode* stage; `and` enters the *Decode* stage in cycle 3 and stalls there through cycle 4.
- The subsequent instruction (`or`) must remain in the *Fetch* stage during both cycles as well because the *Decode* stage is full.
- In cycle 5, the result can be forwarded from the *Writeback* stage of `lw` to the *Execute* stage of `and`. Also, in cycle 5, source `s7` of the `or` instruction is read directly from the register file, with no need for forwarding.

Stalling to solve l_w Data Dependency



- Note that the Execute stage is unused in cycle 4. Likewise, Memory is unused in cycle 5 and Writeback is unused in cycle 6.
- This unused stage propagating through the pipeline is called a *bubble*, which behaves like a `nop` instruction.
- The bubble is introduced by zeroing out the Execute stage control signals during a Decode stage stall so that the *bubble* performs no action and changes no architectural state.

Stalling to solve lw Data Dependency



- In summary, stalling a stage is performed by disabling its pipeline register (i.e., ***the register to the left of a stage***) so that the stage's inputs do not change.
- When a stage is stalled, all previous stages must also be stalled so that no subsequent instructions are lost.
- The pipeline register directly after the stalled stage must be cleared (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary.

Stalling Logic

- Is either **source register in the Decode stage** the same as the **destination register in the Execute stage**?
AND
- Is the instruction in the **Execute stage a *lw*?**

$IwStall = ((Rs1D == RdE) \text{ OR } (Rs2D == RdE)) \text{ AND } ResultSrcE_0$

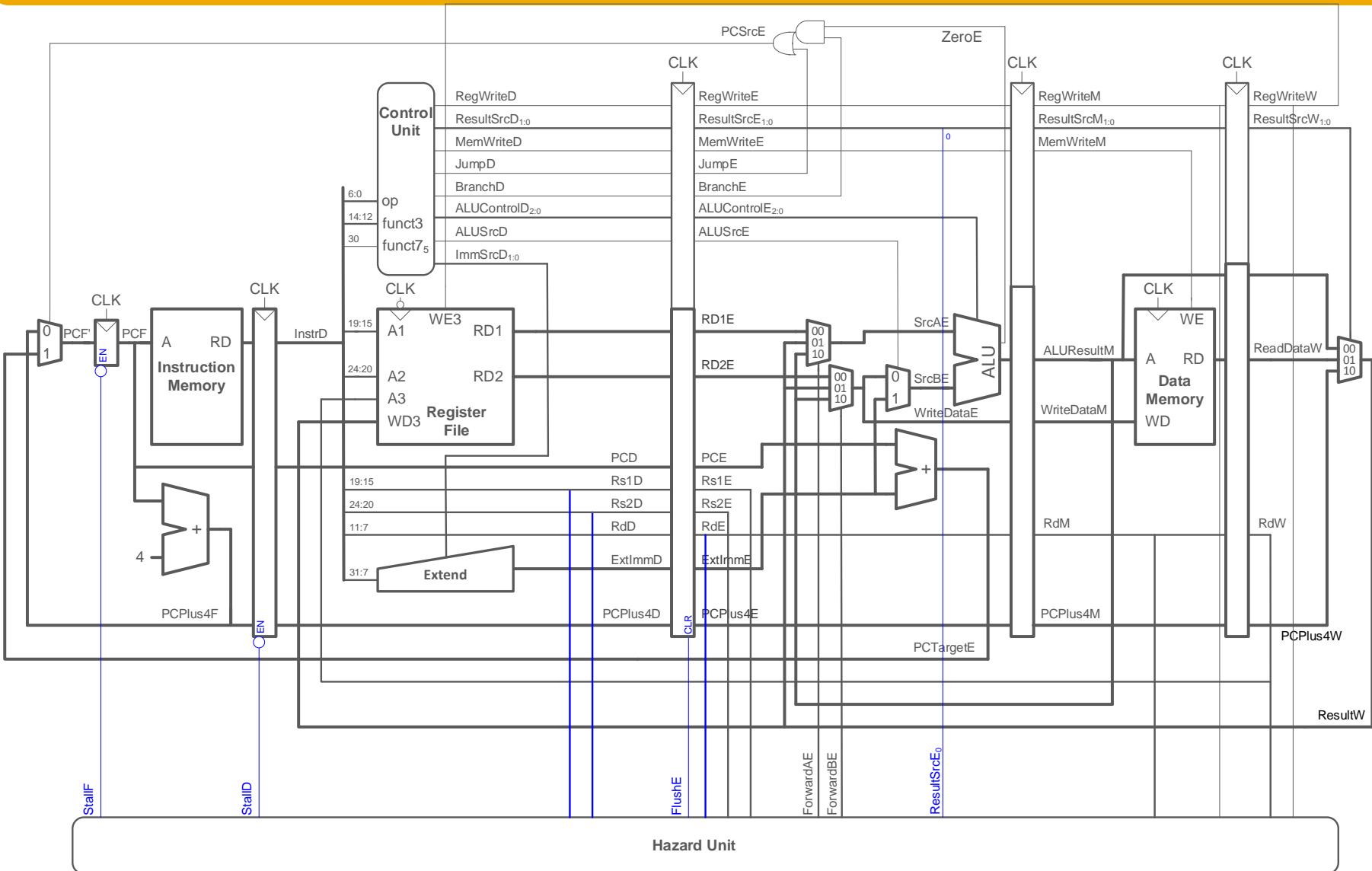
$StallIF = StallID = FlushE = IwStall$

(Stall the Fetch and Decode stages, and flush the Execute stage.)

Stalling Logic

- Stalls are supported by adding
 - enable inputs (*EN*) to the *Fetch* and *Decode* pipeline registers and
 - a synchronous reset/clear (*CLR*) input to the *Execute* pipeline register.
- When a load word (l_w) stall occurs
 - *StallD* and *StallF* are asserted to force the *Decode* and *Fetch* stage pipeline registers to retain their existing values.
 - *FlushE* is asserted to clear the contents of the *Execute* stage pipeline register, introducing a bubble.
- The Hazard Unit *lwStall* (load word stall) signal indicates when the pipeline should be stalled due to a load word dependency.
- Whenever *lwStall* is TRUE, all of the stall and flush signals are asserted.

Stalling Hardware



Chapter 7: Microarchitecture

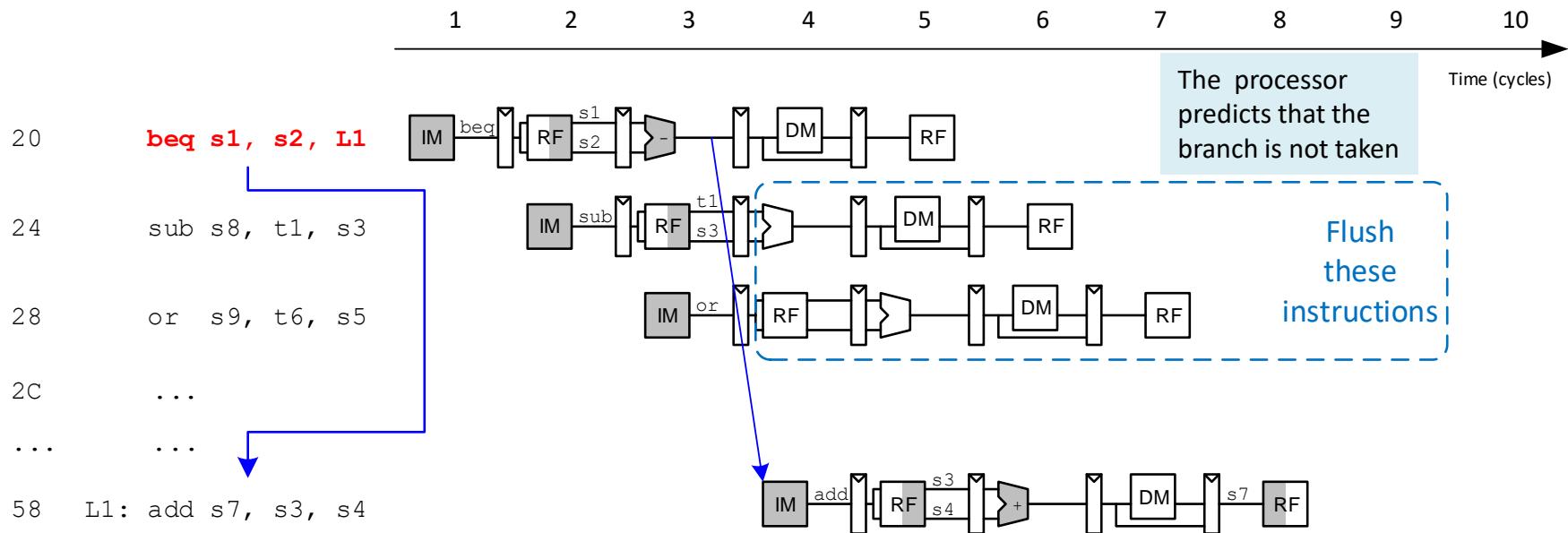
Pipelined Processor Control Hazards

Control Hazards

- **beq:**
 - Branch **not determined until the Execute stage** of pipeline
 - **Instructions** after branch **fetched** before branch occurs
 - These **2 instructions must be flushed** if branch happens

- One mechanism for dealing with this control hazard is to **stall** the pipeline until the branch decision is made (i.e., $PCSrcE$ is computed).
- Because the decision is made in the Execute stage, the pipeline must be stalled for two cycles at every branch.
- This would severely degrade the system performance if branches occur often, which is typically the case.

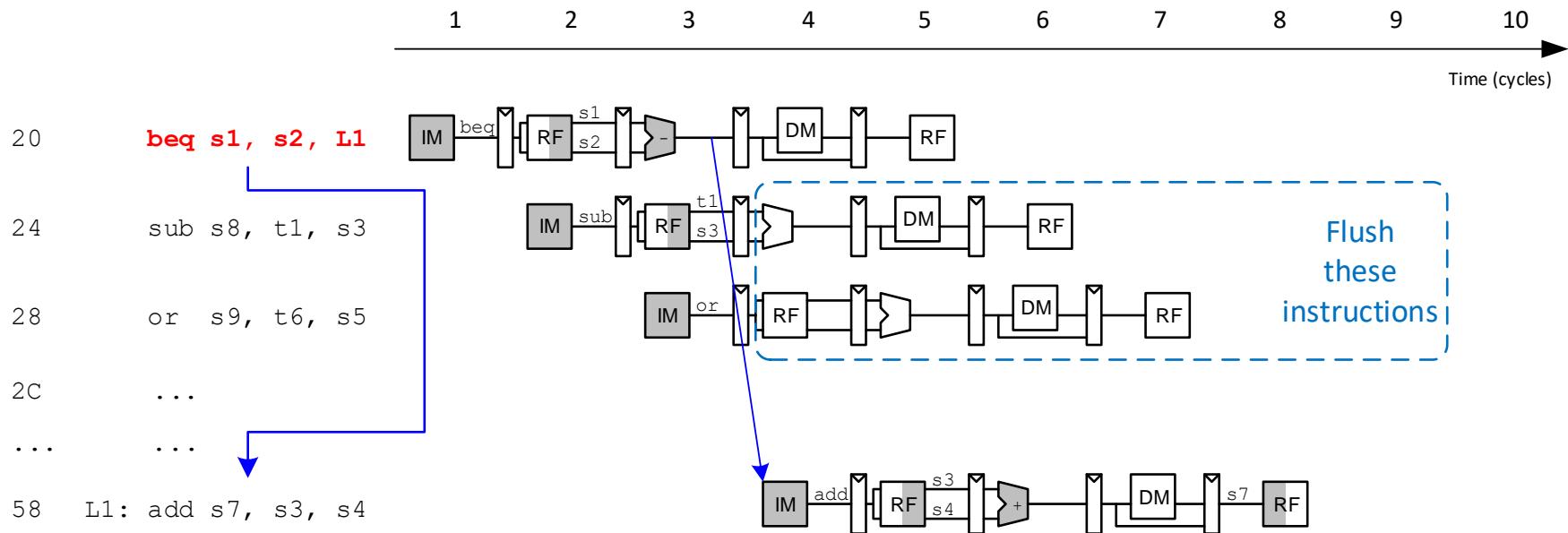
Control Hazards



Branch misprediction penalty:

The number of instructions *flushed* (discarded) when a branch is taken (in this case, 2 instructions)

Control Hazards



Branch misprediction penalty:

The number of instructions flushed when a branch is taken (in this case, 2 instructions)

Control Hazards: Flushing Logic

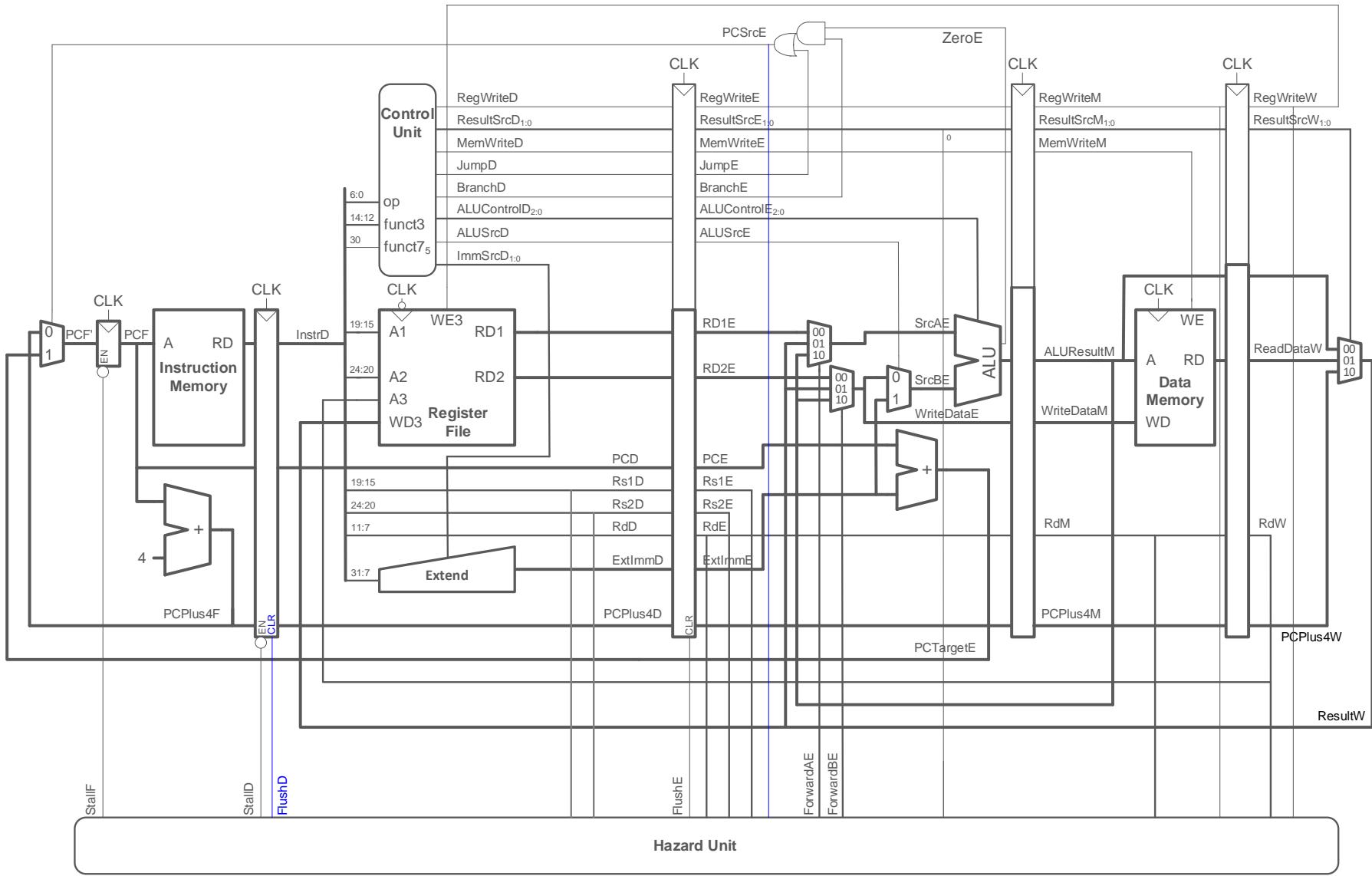
- If branch is taken in *execute* stage, need to flush the instructions in the *Fetch* and *Decode* stages
 - Do this by clearing Decode and Execute Pipeline registers using $FlushD$ and $FlushE$
- **Flushing Control Logic Equations:**

$$FlushD = PCSrcE$$

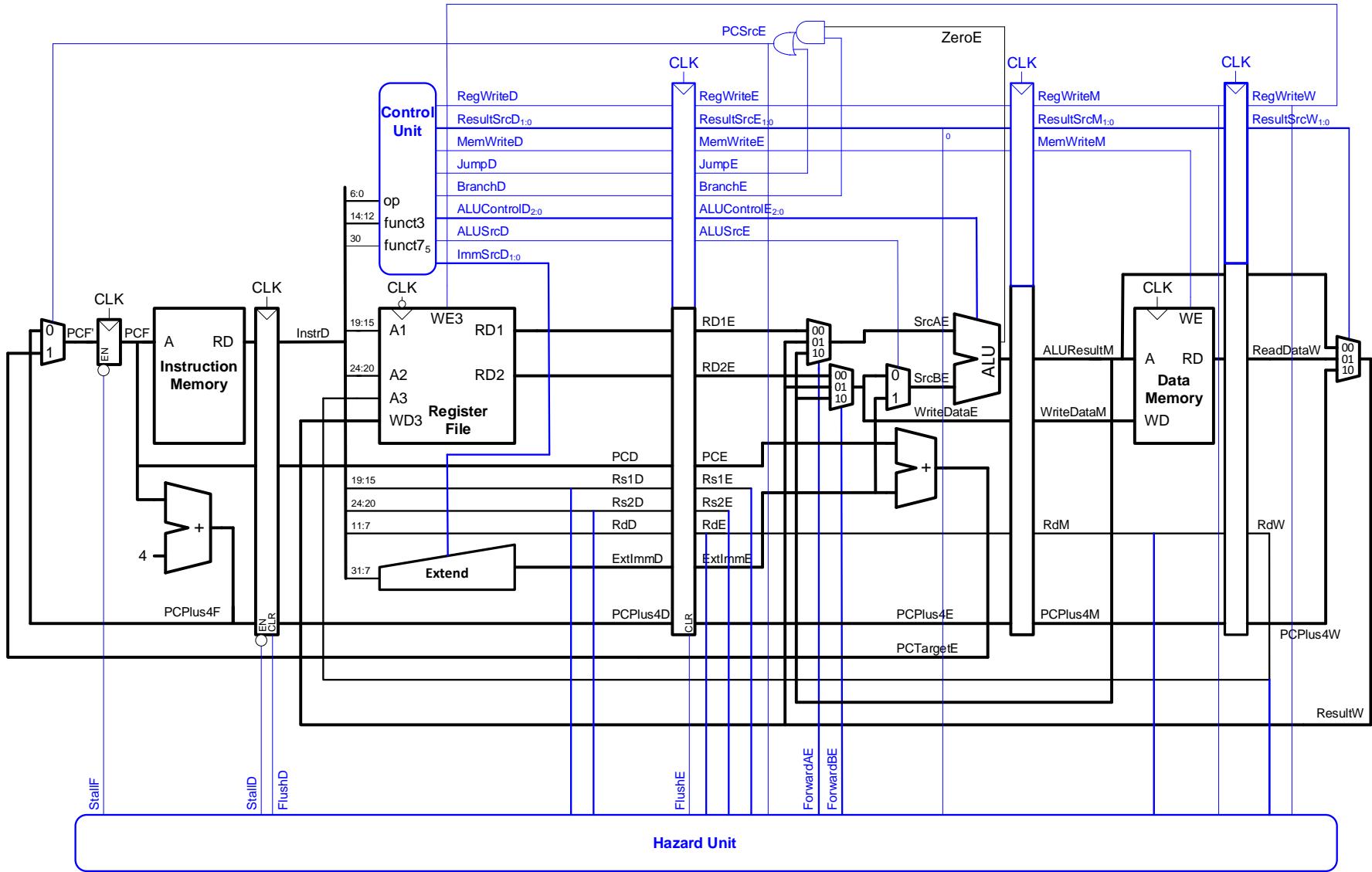
$$FlushE = lwStall \text{ OR } PCSrcE$$

- We add a synchronous clear input (CLR) to the *Decode* pipeline register and add the $FlushD$ output to the Hazard Unit. (When CLR = 1, the register contents are cleared, that is, become 0.) When a branch is taken (indicated by $PCSsrcE$ being 1), $FlushD$ and $FlushE$ must be asserted to flush the *Decode* and *Execute* pipeline registers.
- The next slide shows the enhanced pipelined processor for handling control hazards.

Control Hazards: Flushing Hardware



RISC-V Pipelined Processor with Hazard Unit



Summary of Hazard Logic

Data hazard logic (shown for SrcA of ALU):

```
if      ((Rs1E == RdM) AND RegWriteM) AND (Rs1E != 0)    // Forward from Memory stage
          ForwardAE = 10
else if ((Rs1E == RdW) AND RegWriteW) AND (Rs1E != 0)    // Forward from Writeback
stage
          ForwardAE = 01
else
          ForwardAE = 00                                     // No forwarding (use RF output)
```

Load word stall logic:

$lwStall = ((Rs1D == RdE) \text{ OR } (Rs2D == RdE)) \text{ AND } ResultSrcE_0$
 $StallF = StallID = lwStall$

Control hazard flush:

$FlushD = PCSrcE$
 $FlushE = lwStall \text{ OR } PCSrcE$

Chapter 7: Microarchitecture

Pipelined Performance

Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**

- 25% loads
- 10% stores
- 13% branches
- 52% R-type

The SPECint2000 benchmark is part of a suite of performance benchmarks developed by the Standard Performance Evaluation Corporation (SPEC), specifically targeting the evaluation of a computer's integer processing power. It was introduced as part of the SPEC CPU2000 suite and became widely used in the early 2000s for measuring and comparing the performance of CPUs, particularly for tasks that rely heavily on integer arithmetic rather than floating-point operations.

- **Suppose:**

- 40% of loads used by next instruction
- 50% of branches mispredicted

- **What is the average CPI? (Ideally it's 1, but...)**

Pipelined Processor Performance Example

- **SPECINT2000 benchmark:**

- 25% loads
- 10% stores
- 13% branches
- 52% R-type

- **Suppose:**

- 40% of loads used by next instruction
- 50% of branches mispredicted

- **What is the average CPI? (Ideally it's 1, but...)**

- Load CPI = 1 when not stalling, 2 when stalling
So, $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
- Branch CPI = 1 when not stalling, 3 when mispredict
So, $CPI_{beq} = 1(0.5) + 3(0.5) = 2$

$$\text{Average CPI} = (0.25)(1.4) + (0.1)(1) + (0.13)(2) + (0.52)(1) = 1.23$$

Pipelined Processor Performance Example

Pipelined processor critical path:

$T_{c_pipelined}$ = max of

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{RFread} + t_{setup})$$

$$t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup}$$

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{pcq} + t_{mux} + t_{RFwrite})$$

Fetch

Decode

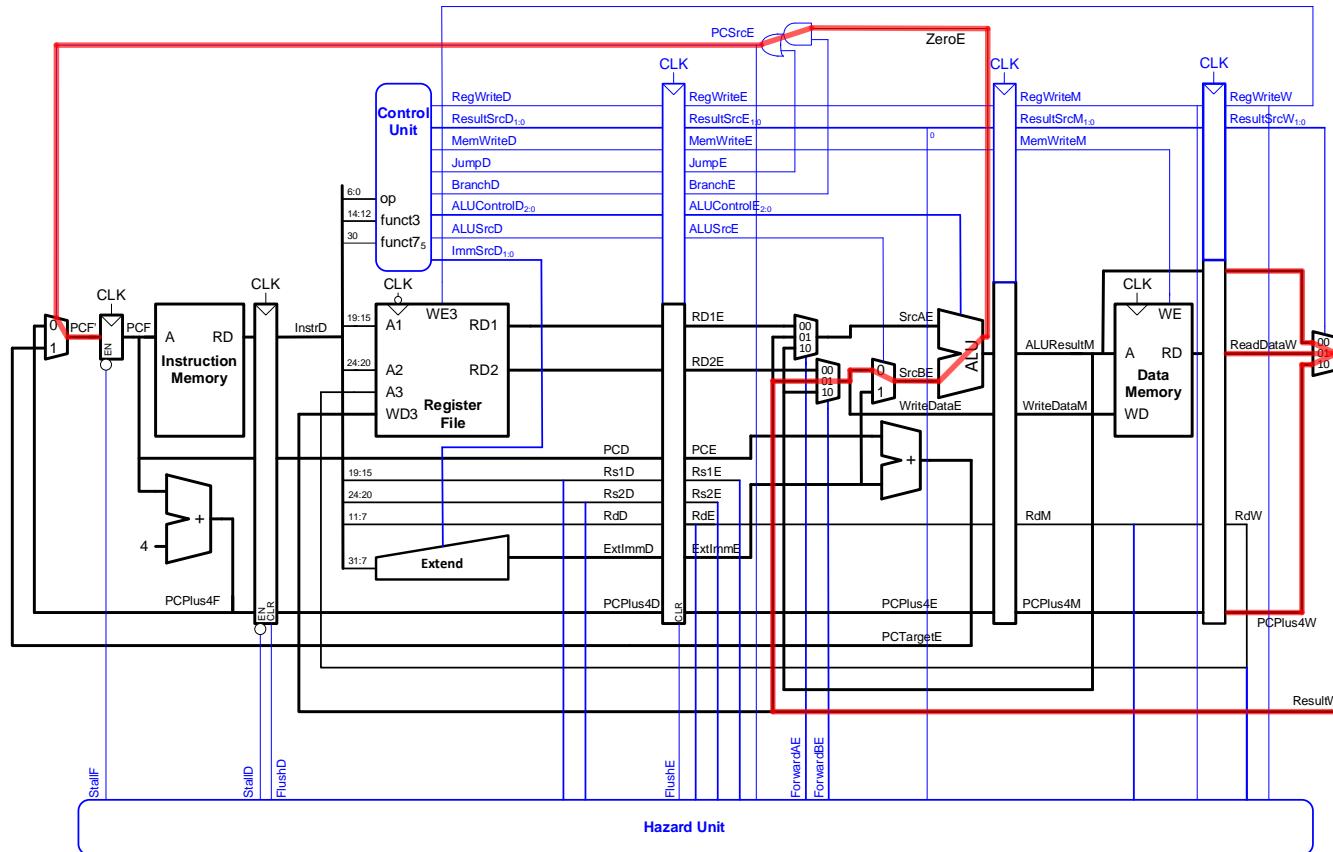
Execute

Memory

Writeback

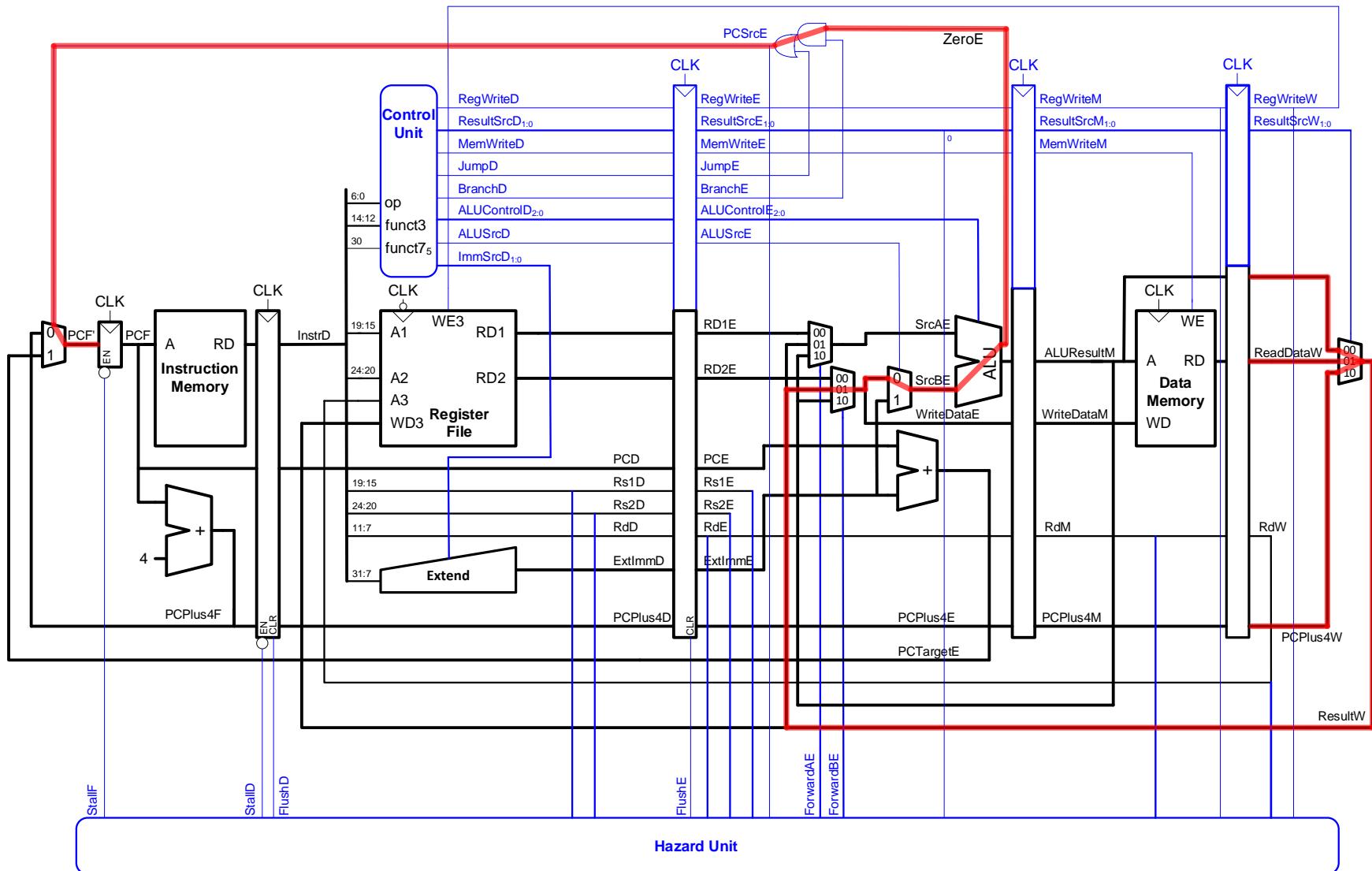
- Decode and Writeback stages **both use the register file** in each cycle.
- The RF is used twice in a single cycle, it is written in the 1st half of the Writeback cycle and read in the 2nd half of the Decode cycle.
- So each stage gets half of the cycle time ($T_c/2$) to do their work
- Or, stated a different way, **2x of their work** must fit in a cycle (T_c).

Pipelined Critical Path: Execute Stage



It occurs when a branch is in the Execute stage that requires forwarding from the Writeback stage: the path goes from the Writeback pipeline register, through the Result, ForwardBE, and SrcB multiplexers, through the ALU and AND-OR logic to the PC multiplexer and, finally, to the PC register.

Pipelined Critical Path: Execute Stage



Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$T_{c_pipelined} = t_{pcq} + 4t_{\text{mux}} + t_{\text{ALU}} + t_{\text{AND-OR}} + t_{\text{setup}}$$

=

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$\begin{aligned}T_{c_pipelined} &= t_{pcq} + 4t_{\text{mux}} + t_{\text{ALU}} + t_{\text{AND-OR}} + t_{\text{setup}} \\&= (40 + 4*30 + 120 + 20 + 50) \text{ ps} = 350 \text{ ps}\end{aligned}$$

Pipelined Performance Example

Program with 100 billion instructions

$$\begin{aligned}\text{Execution Time} &= (\# \text{ instructions}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(1.23)(350 \times 10^{-12}) \\ &= \mathbf{43 \text{ seconds}}\end{aligned}$$

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	75	1
Multicycle	155	0.5
Pipelined	43	1.7

- The pipelined processor is substantially faster than the others. However, its advantage over the single-cycle processor is nowhere near the fivefold speedup one might hope to get from a five-stage pipeline.
- Our pipelined processor is unbalanced, with branch resolution in the Execute stage taking much longer than any other stage. The pipeline could be balanced better by pushing the Result multiplexer back into the Memory stage, reducing the cycle time to 320 ps.
- The pipeline hazards introduce a small CPI penalty.
- More significantly, the **sequencing overhead** (*clk-to-Q* and *setup times*) of the registers applies to every pipeline stage, not just once to the overall datapath. Sequencing overhead limits the benefits one can hope to achieve from pipelining.
- Imbalanced delay in pipeline stages also decreases the benefits of pipelining.

Chapter 7: Microarchitecture

Advanced Microarchitecture

Advanced Microarchitecture

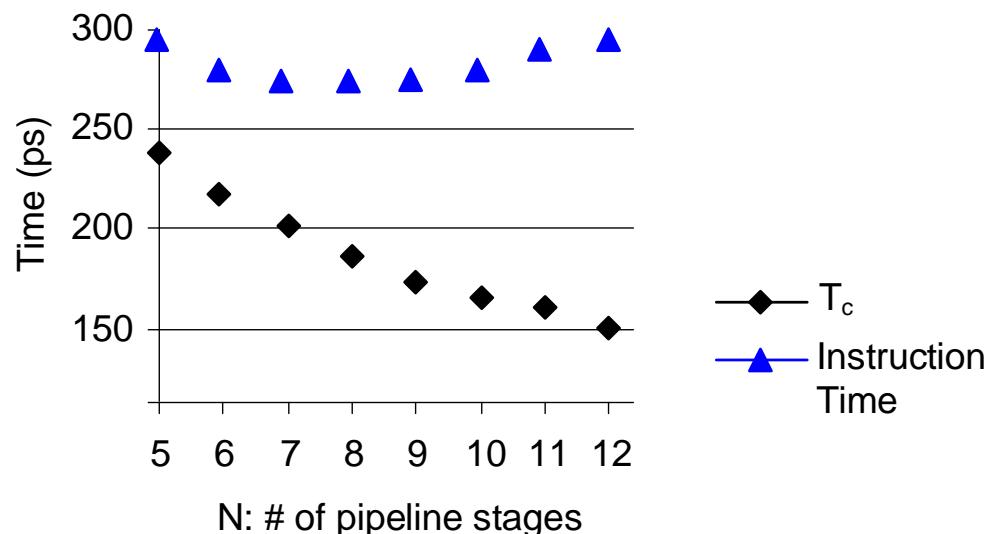
- Deep Pipelining
- Micro-operations
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- **10-20 stages typical**
- Number of stages limited by:
 - Pipeline hazards
 - Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding but others require stalls, which increase the CPI.
 - Sequencing overhead
 - The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). Due to this sequencing overhead, adding more pipeline stages gives diminishing returns.
 - Cost
 - Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Deep Pipelining

- **10-20 stages typical**
- Number of stages limited by:
 - Pipeline hazards
 - Sequencing overhead
 - Cost



Deep Pipelines Example

Example 7.11 DEEP PIPELINES

Consider building a pipelined processor by chopping up the single-cycle processor into N stages. The single-cycle processor has a propagation delay of 750 ps through the combinational logic. The sequencing overhead of a register is 90 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.9 has a CPI of 1.25.¹ Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Deep Pipelines Example

Solution The cycle time for an N -stage pipeline is $T_c = [(750/N) + 90]$ ps. The CPI is $1.25 + 0.1(N - 5)$, where $N \geq 5$. The time per instruction (i.e., instruction time) is the product of the cycle time T_c and the CPI. Figure 7.66 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 281 ps at $N = 8$ stages. This minimum is only slightly better than the 295 ps per instruction achieved with a five-stage pipeline, and the curve is almost flat between 7 to 10 stages.

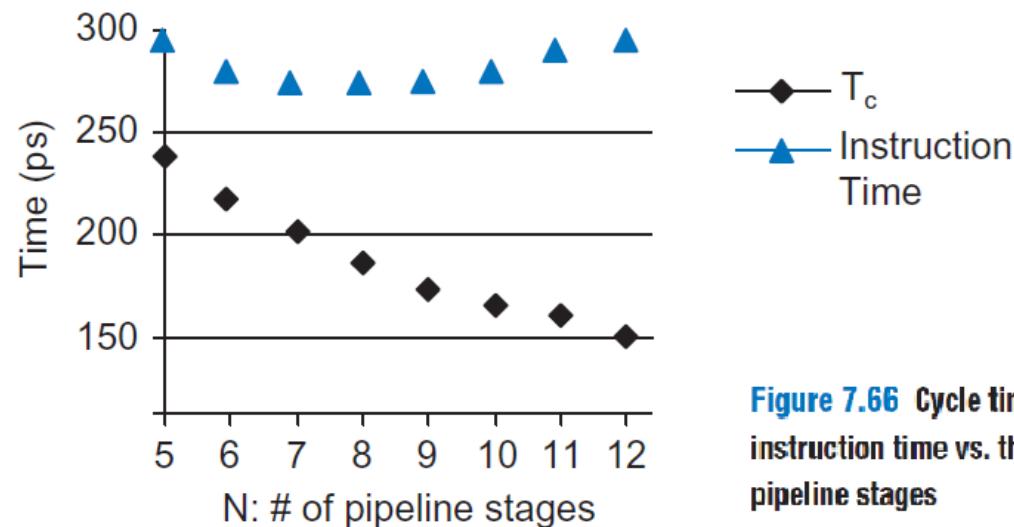


Figure 7.66 Cycle time and instruction time vs. the number of pipeline stages

Micro-operations

- Decompose complex instructions into series of simple instructions called ***micro-operations*** (*micro-ops* or μ -*ops*)
- **At run-time**, complex instructions are decoded into one or more micro-ops
- Used heavily in **CISC** (complex instruction set computer) architectures (e.g., x86)

Complex Op

lw s1, 0(s2), postincr 4

Micro-op Sequence

lw s1, 0(s2)
addi s2, s2, 4

Without μ -ops, would need 2nd write port on the register file

Micro-operations

- For example, the x86 instruction **ADD [ESP], [EDX+80+EDI*2]** involves
 - reading the three registers (ESP, EDX, and EDI),
 - adding the base (EDX), displacement (80), and scaled index (EDI*2), reading two memory locations, summing their values, and
 - writing the result back to memory.
- A microprocessor that could perform all of these functions at once would be unnecessarily slow when executing more common, simpler instructions.
- CISC Computer architect processors make the common case fast by defining a set of simple *micro-operations* (also known as *micro-ops* or *μops*) that can be executed on simple datapaths.
- Each CISC instruction is decoded into one or more micro-ops.
- For example, if we defined *μops* resembling RISC-V instructions, and used temporary registers t1 and t2 to hold intermediate results, then the x86 instruction above could become six μops:

Complex x86vOp

ADD [ESP],[EDX+80+EDI*2]

RISC-V like *μops* Sequence

```
slli t2, EDI, 1    # t2 = EDI*2  
add  t1, EDX, t2  # t1 = EDX + EDI*2  
lw   t1, 80(t1)  # t1 = MEM[EDX + EDI*2 + 80]  
lw   t2, 0(ESP)   # t2 = MEM[ESP]  
add  t1, t2, t1  # t1 = MEM[ESP] + MEM[EDX + EDI*2 + 80]  
sw   t1, 0(ESP)   # MEM[ESP] = MEM[ESP] + MEM[EDX + EDI*2 + 80]
```

Branch Prediction

- **Guess** whether branch will be taken
 - *Static branch prediction*: Backward branches are usually taken (loops).
 - *Dynamic branch prediction*: Consider history to improve guess.
- Good prediction **reduces fraction of branches requiring a flush**

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
 - Check direction of branch (forward or backward)
 - If backward, predict taken
 - Else, predict not taken
- **Dynamic branch prediction:**
 - Keep **history** of last several hundred (or thousand) branches in *branch target buffer*, record:
 - Branch destination
 - Whether branch was taken

Dynamic Branch Prediction

- 1-bit branch predictor
- 2-bit branch predictor

Branch Prediction Example

```
addi s1, zero, 0      # s1 = sum  
addi s0, zero, 0      # s0 = i  
addi t0, zero, 10     # t0 = 10
```

For: # for (i=0; i<10; i=i+1)

```
bge s0, t0, Done  
add s1, s1, s0        # sum = sum + i  
addi s0, s0, 1         # i = i + 1  
j For
```

Done:

1-Bit Branch Predictor

- **Remembers** whether branch was taken the last time and **does the same thing**
- Mispredicts *first* and *last* branch of loop

```
addi s1, zero, 0      # s1 = sum  
addi s0, zero, 0      # s0 = i  
addi t0, zero, 10    # t0 = 10
```

For: # for (i=0; i<10; i=i+1)

```
bge s0, t0, Done  
add s1, s1, s0        # sum = sum + i  
addi s0, s0, 1         # i = i + 1  
j   For
```

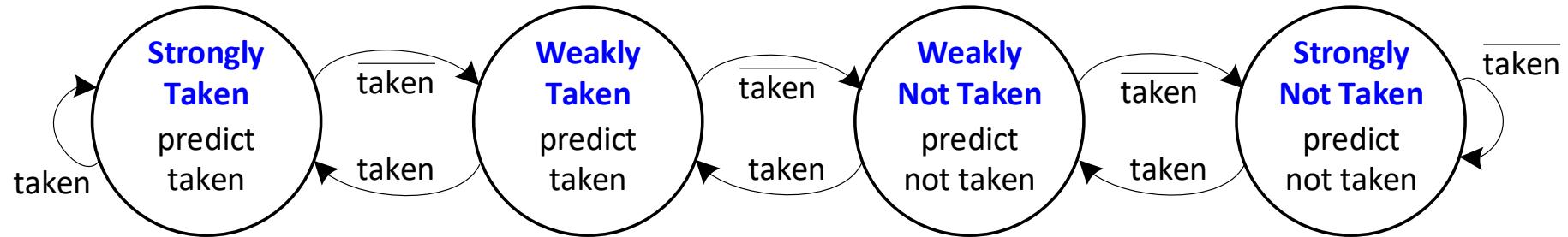
Done:

While the loop is repeating, it remembers that the `beg` was not taken last time and predicts that it should not be taken next time. *This is a correct prediction until the last branch of the loop, when the branch does get taken.*

Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore, *it incorrectly predicts that the branch should be taken when the loop is first run again.* In summary,

a 1-bit branch predictor mispredicts the first and last branches of a loop.

2-Bit Branch Predictor



```
addi s1, zero, 0      # s1 = sum  
addi s0, zero, 0      # s0 = i  
addi t0, zero, 10     # t0 = 10
```

The branch predictor operates in the pipeline *Fetch* stage so that it can determine which instruction to execute on the next cycle.

For: # for (i=0; i<10; i=i+1)

```
bge s0, t0, Done  
add s1, s1, s0      # sum = sum + i  
addi s0, s0, 1       # i = i + 1  
j    For
```

Done:

- Branch predictors may be used to track even more history of the program to increase the accuracy of predictions.
- Good branch predictors achieve better than 90% accuracy on typical programs.

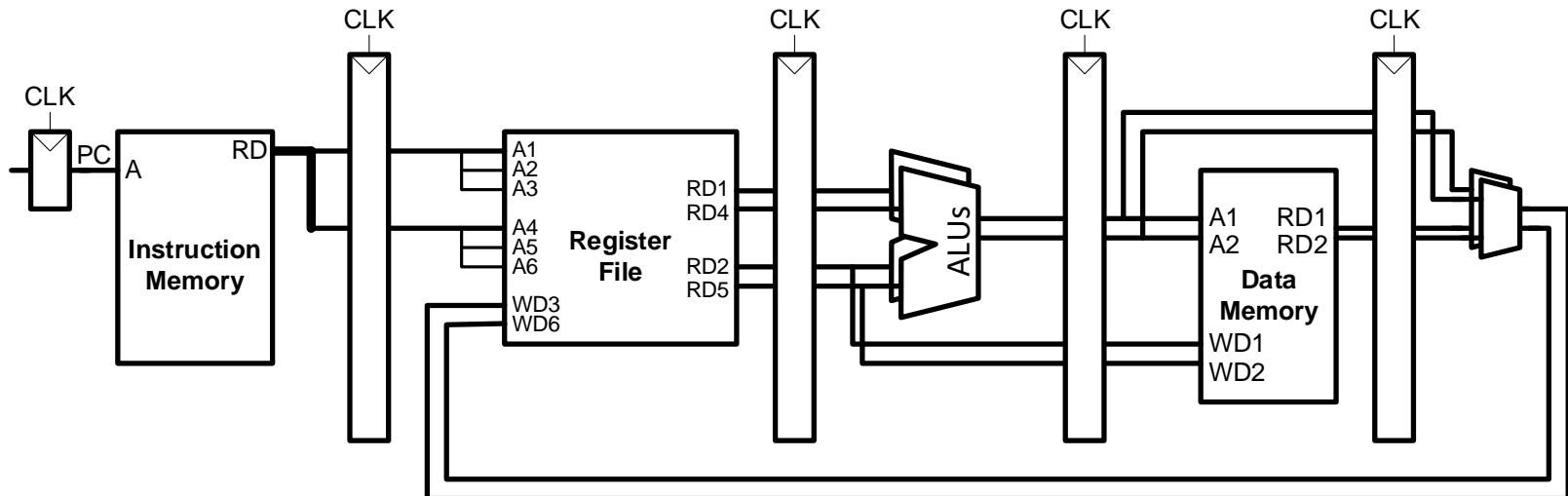
Only mispredicts last branch of loop

Chapter 7: Microarchitecture

Superscalar & Out of Order Processors

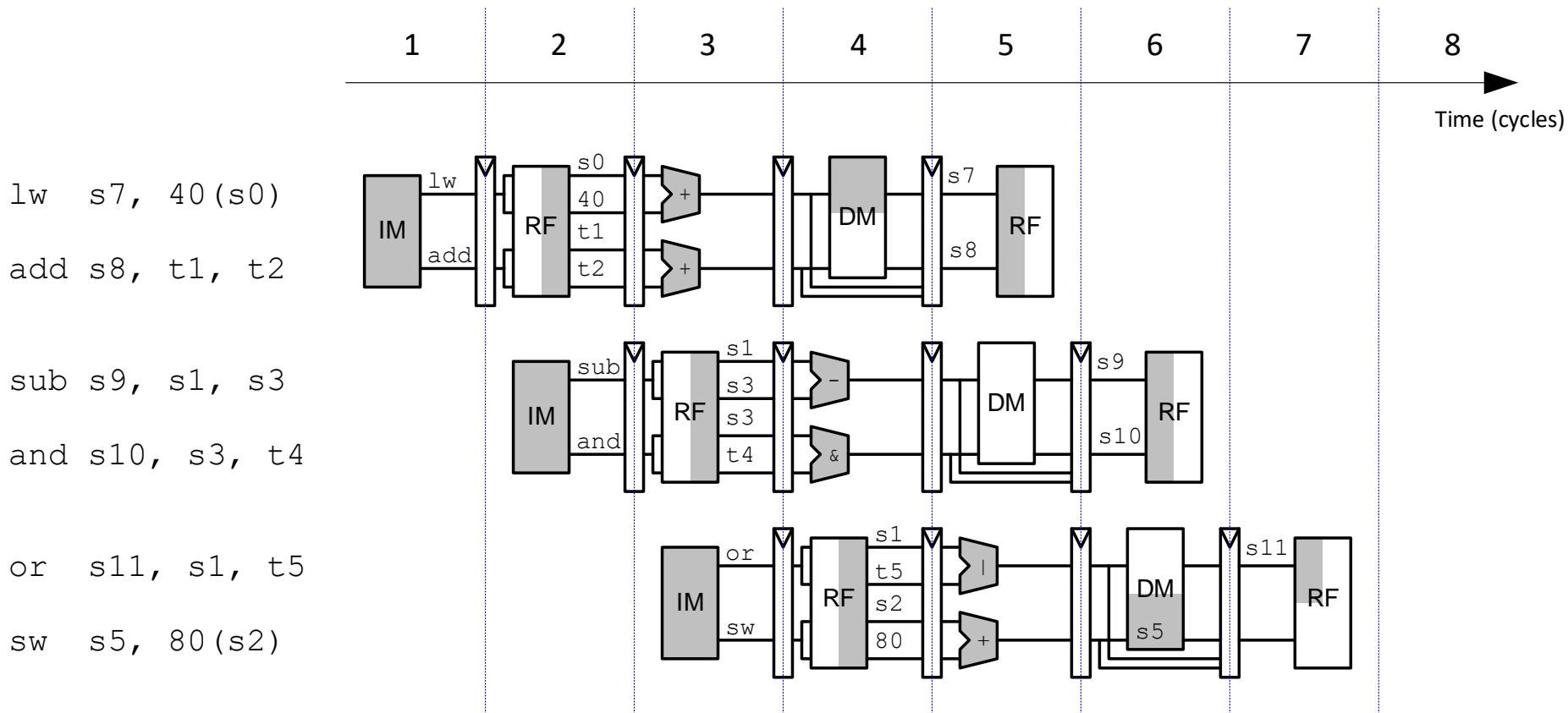
Superscalar Processors

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once



Superscalar Example

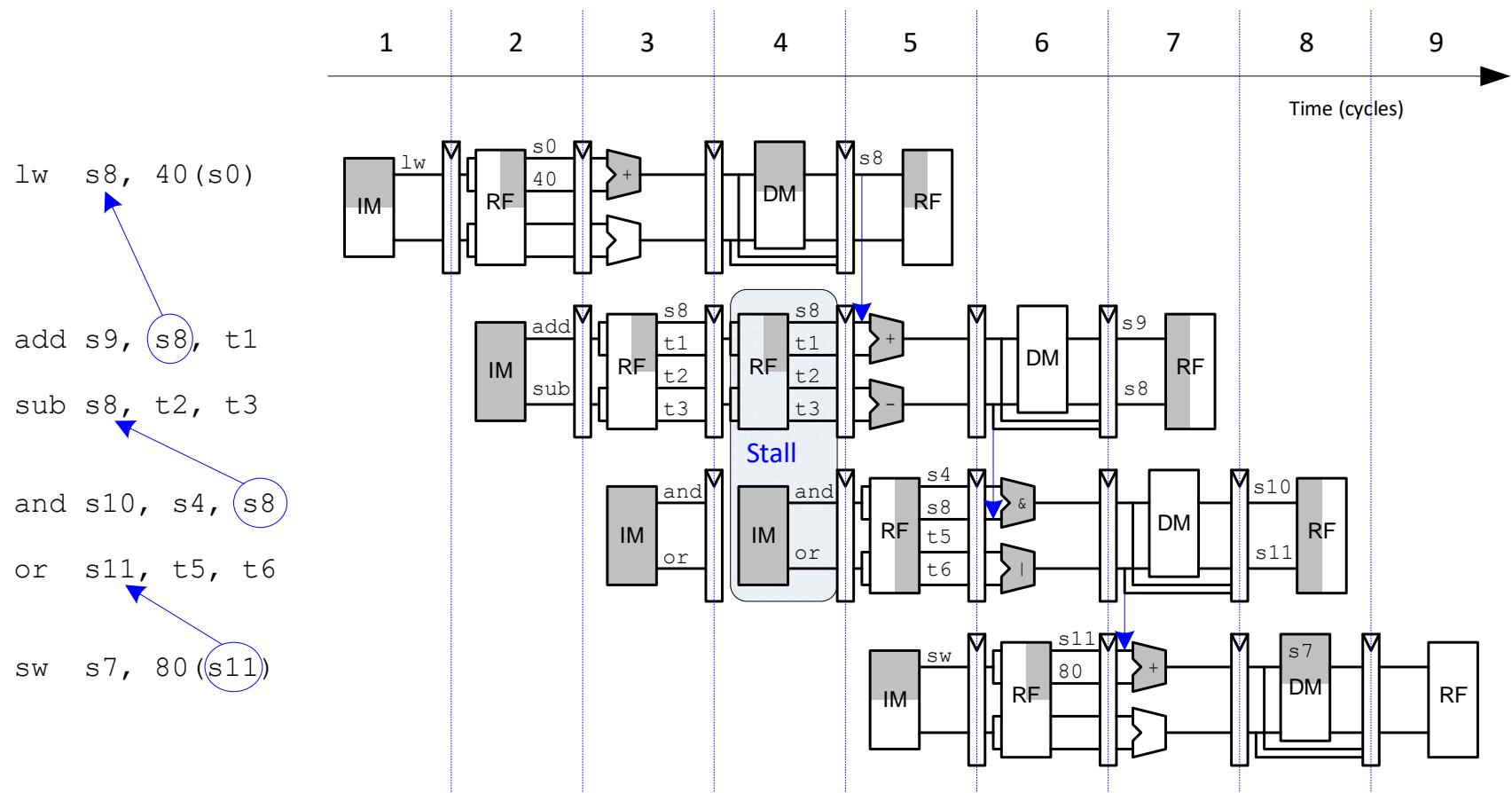
Ideal Instructions Per Cycle (IPC): 2
Actual IPC: 2



Superscalar with Dependencies

Ideal IPC: 2

Actual IPC: $6/5 = 1.2$ (need five cycles to issue 6 instructions)



Parallelism

- Parallelism comes in *temporal* and *spatial* forms.
- Pipelining is a case of temporal parallelism.
- Using multiple execution units is a case of spatial parallelism.
- Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.
- Commercial processors may be three-, four-, or even six-way superscalar.
 - They must handle control hazards such as branches as well as data hazards.
- Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units.
- Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

Out of Order (OOO) Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
 - **RAW** (read after write): one instruction writes, later instruction reads a register
 - **WAR** (write after read): one instruction reads, later instruction writes a register
 - **WAW** (write after write): one instruction writes, later instruction writes a register

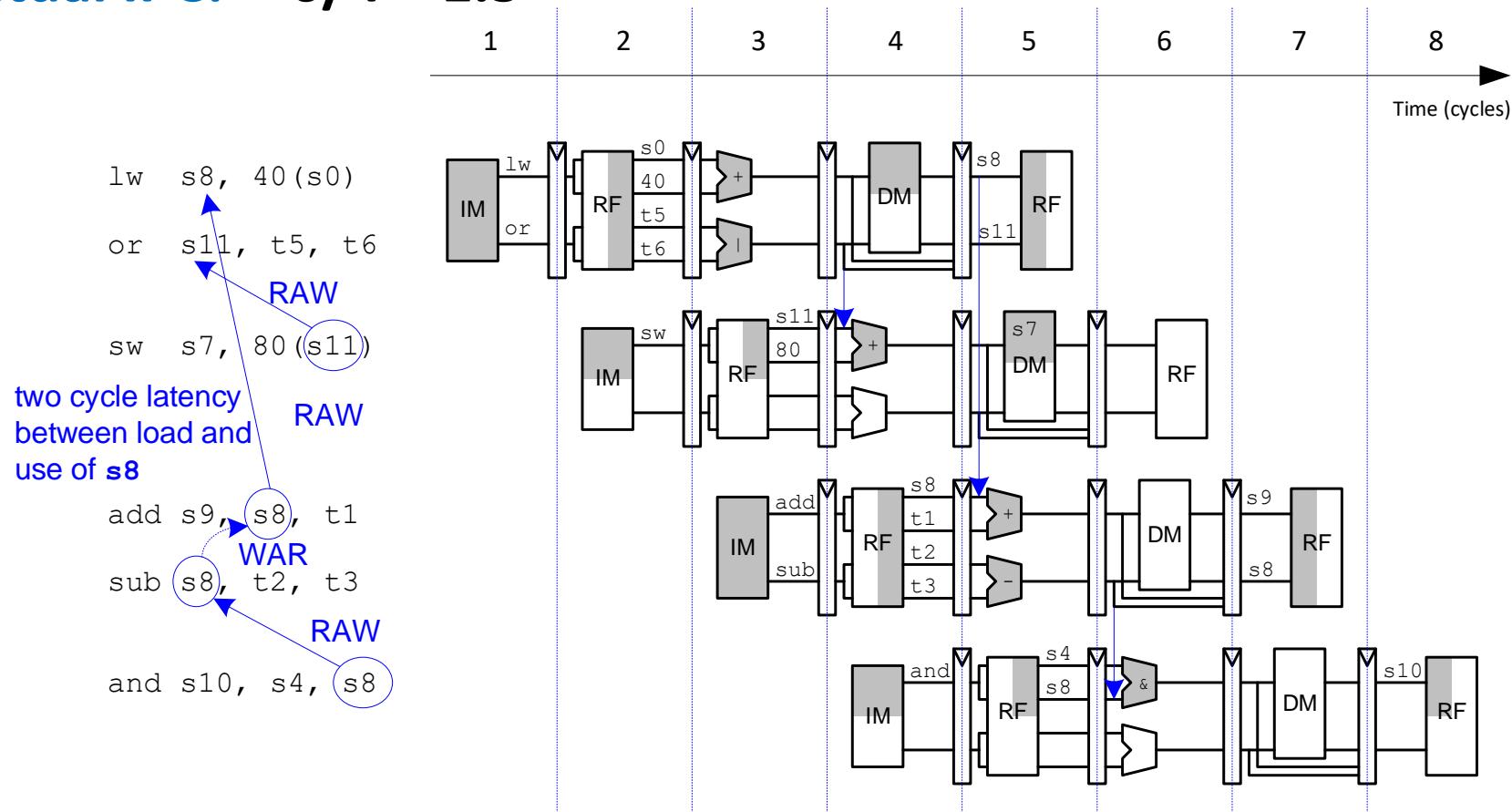
Out of Order (OOO) Processor

- **Instruction level parallelism (ILP):** number of instructions that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
 - Instructions waiting to issue
 - Available functional units
 - Dependencies

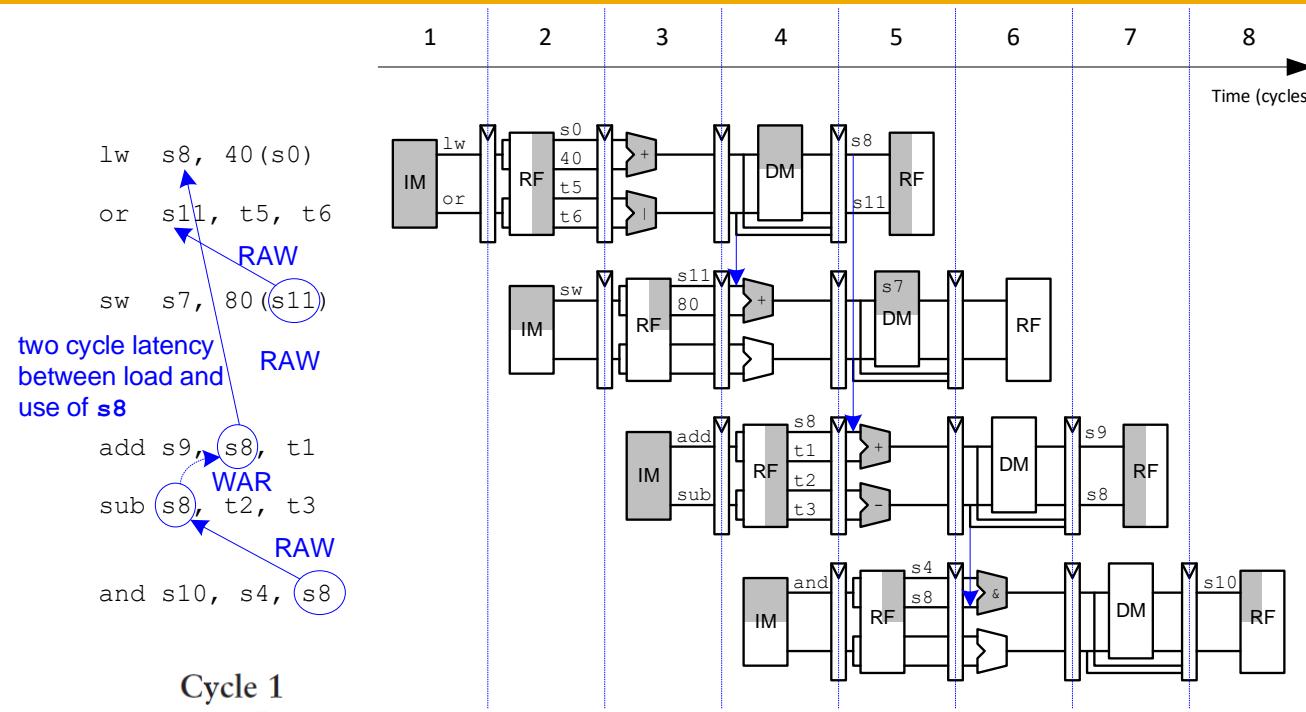
Out of Order Processor Example

Ideal IPC: 2

Actual IPC: $6/4 = 1.5$



Out of Order Processor Example



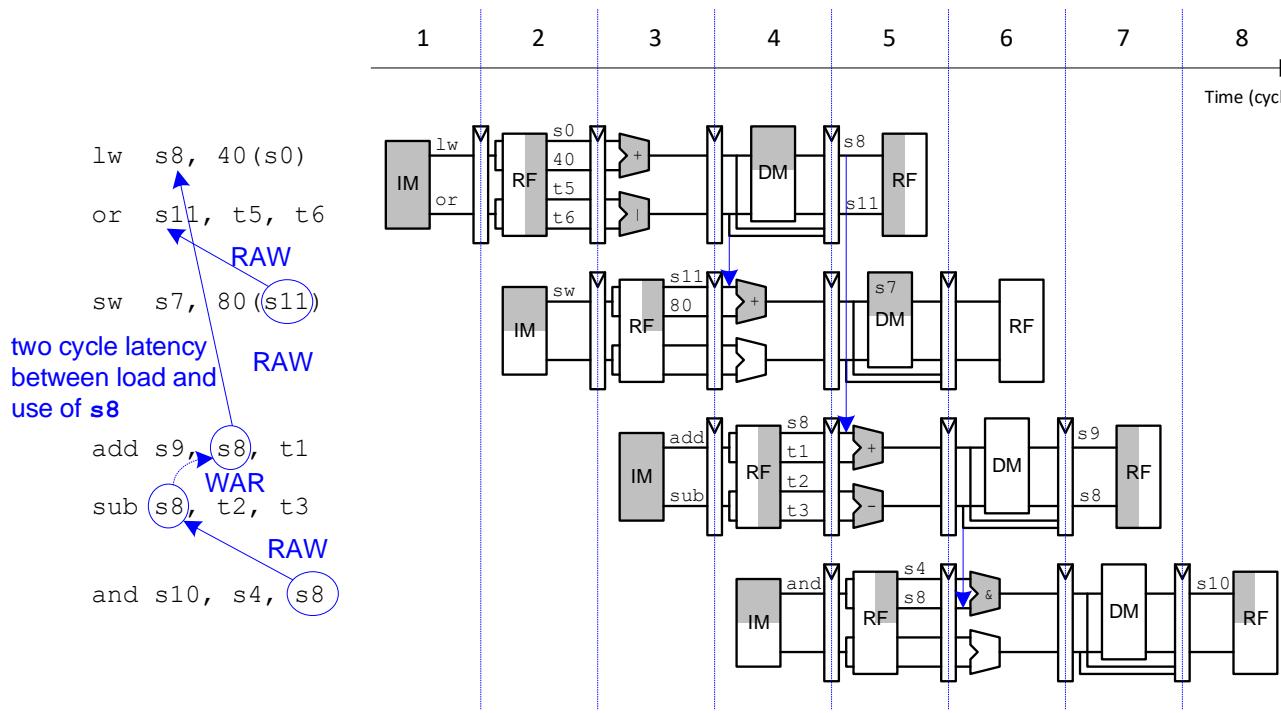
Cycle 1

- ▶ The `lw` instruction issues.
- ▶ The `add`, `sub`, and `and` instructions are dependent on `lw` by way of `s8`, so they cannot issue yet. However, the `or` instruction is independent, so it also issues.

Cycle 2

- ▶ Remember that a two-cycle latency exists between issuing `lw` and a dependent instruction, so `add` cannot issue yet because of the `s8` dependence. `sub` writes `s8`, so it cannot issue before `add`, lest `add` receive the wrong value of `s8`. `and` is dependent on `sub`.
- ▶ Only the `sw` instruction issues.

Out of Order Processor Example



- The dependence of add on lw by way of s8 is a *read after write (RAW)* hazard. add must not read s8 until after lw has written it.
- The dependence between sub and add by way of s8 is called a *write after read (WAR)* hazard or an *antidependence*. sub must not write s8 before add reads s8, so that add receives the correct value according to the original order of the program.
- WAR hazards could not occur in the simple pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, sub) is moved too early.

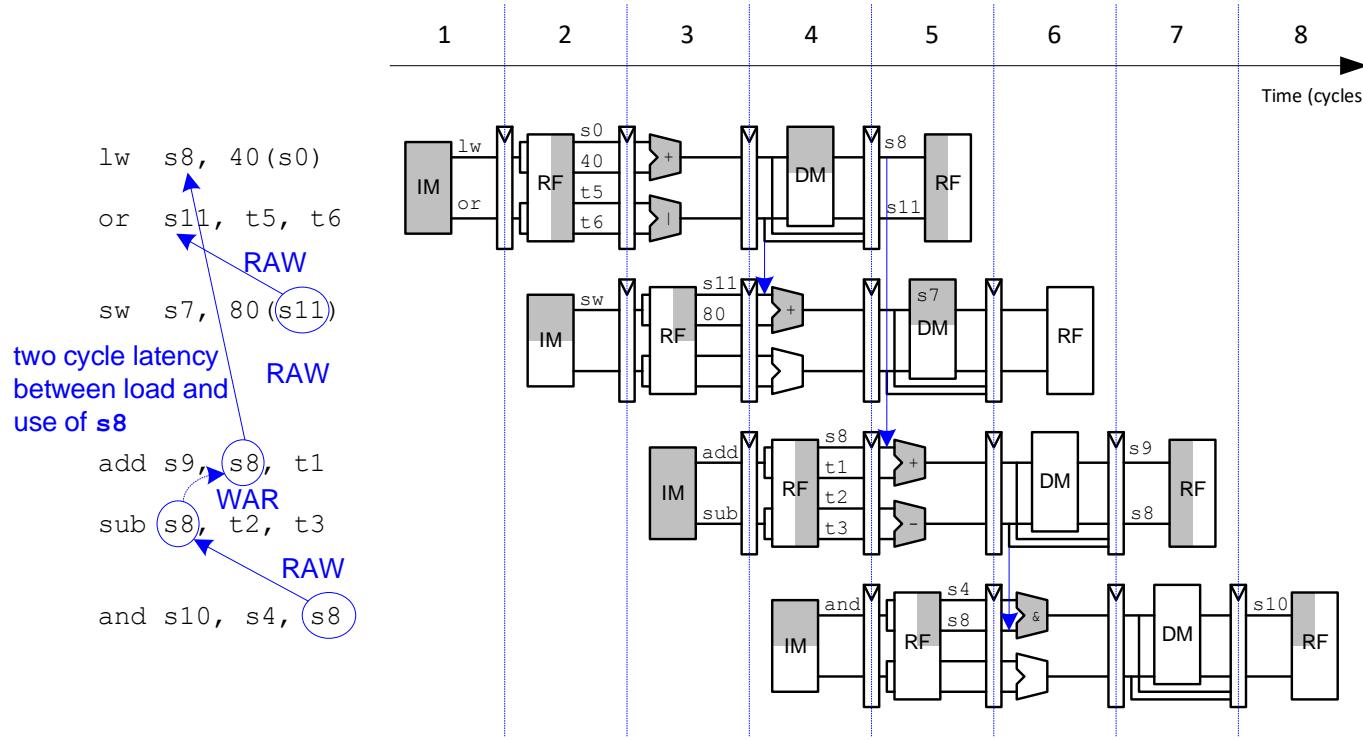
Cycle 3

- ▶ On cycle 3, s8 is available (or, rather, will be when add needs it), so the add issues. sub issues simultaneously, because it will not write s8 until after add consumes (i.e., reads) it.

Cycle 4

- ▶ The and instruction issues. s8 is forwarded from sub to and.

Out of Order Processor Example



- A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions.
- If the `sub` instruction had written `s3` instead of `s8`, then the dependency would disappear and `sub` could be issued before `add`.
- The RISC-V architecture has only 31 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all of the other registers are in use.

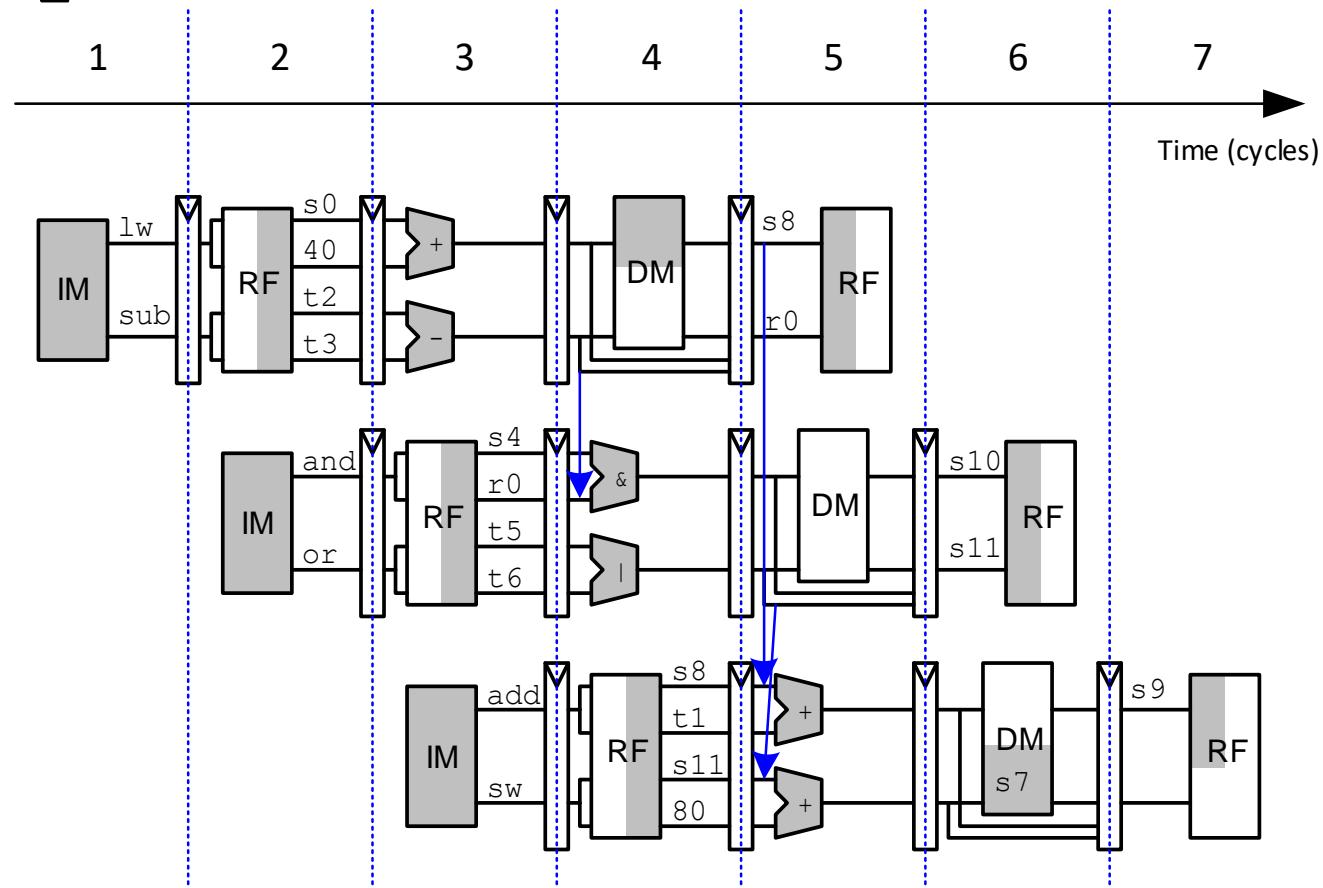
Register Renaming

Ideal IPC: 2

The sub instruction writes s3 instead of s8, the dependency disappears and sub could be issued before add.

Actual IPC: $6/3 = 2$

lw s8, 40 (s0)
sub s3, t2, t3
2-cycle RAW
and s10, s4, **s3**
or s11, t5, t6
add s9, **s8**, t1
sw s7, 80 (**s11**)



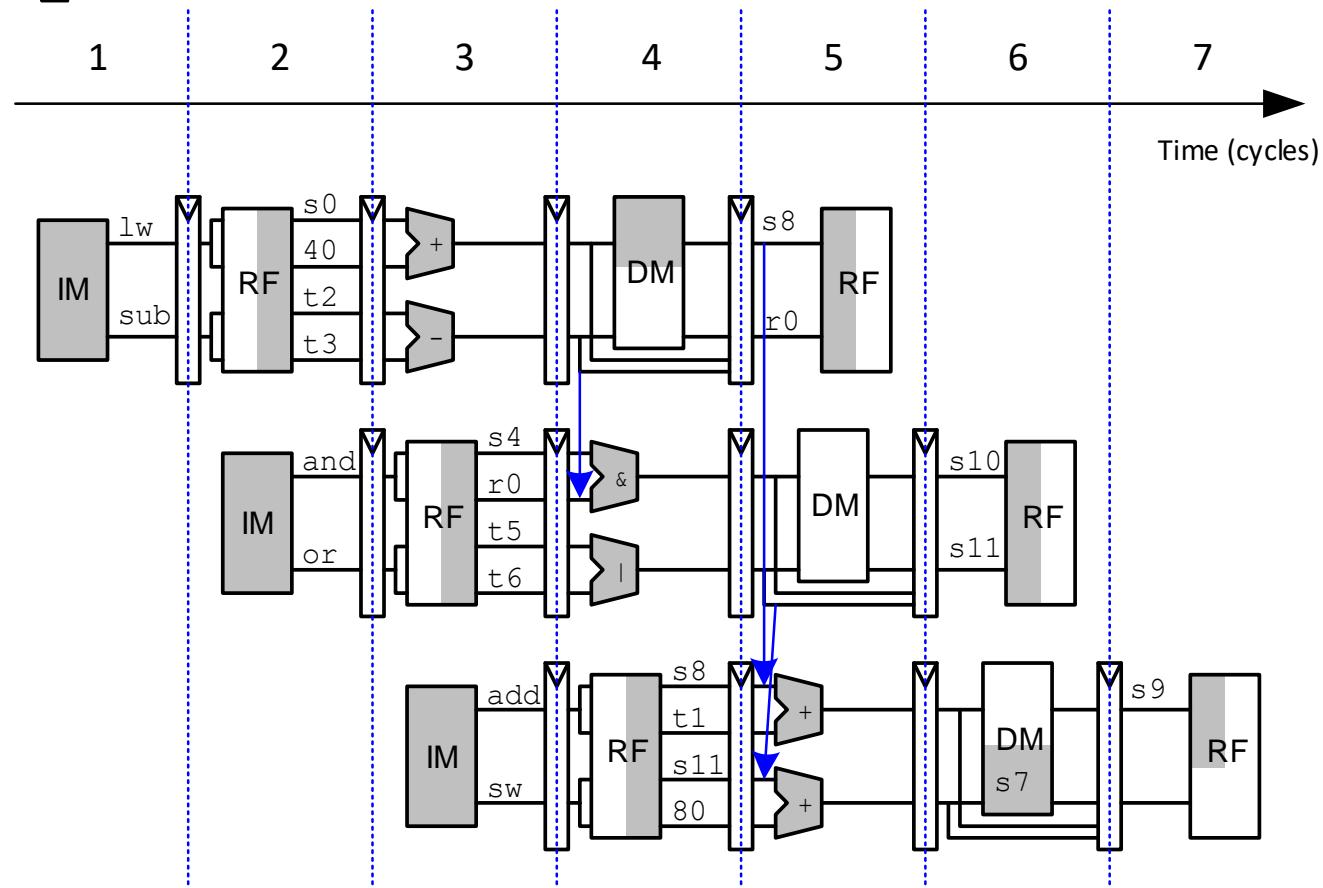
Register Renaming

Ideal IPC: 2

The sub instruction writes s3 instead of s8, the dependency disappears and sub could be issued before add.

Actual IPC: $6/3 = 2$

lw s8, 40 (s0)
sub s3, t2, t3
2-cycle RAW
and s10, s4, **s3**
or s11, t5, t6
add s9, **s8**, t1
sw s7, 80 (**s11**)



SIMD

- **Single Instruction Multiple Data (SIMD)**
 - Single instruction acts on multiple pieces of data at once
 - Common application: **graphics**
 - Can apply to short arithmetic operations (also called *packed arithmetic*)
- For example, add eight 8-bit elements

Bit position								
63	56 55	48 47	40 39	32 31	24 23	16 15	8 7	0
a ₇	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀	D0
+	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀
<hr/>								
a ₇ + b ₇	a ₆ + b ₆	a ₅ + b ₅	a ₄ + b ₄	a ₃ + b ₃	a ₂ + b ₂	a ₁ + b ₁	a ₀ + b ₀	D2

Chapter 7: Microarchitecture

Multithreading & Multiprocessors

Advanced Architecture Techniques

- **Multithreading**
 - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
 - Multiple processors (cores) on a single chip

Threading: Definitions

- **Process:** program running on a computer
 - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
 - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in a Conventional Processor

Single-core system:

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
 - Architectural state of that thread stored
 - Architectural state of waiting thread loaded into processor and it runs
 - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
 - When one thread stalls, another runs immediately
 - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this “hyperthreading”

Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
 - **Homogeneous:** multiple cores with shared main memory
 - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
 - **Clusters:** each core has own memory system

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.

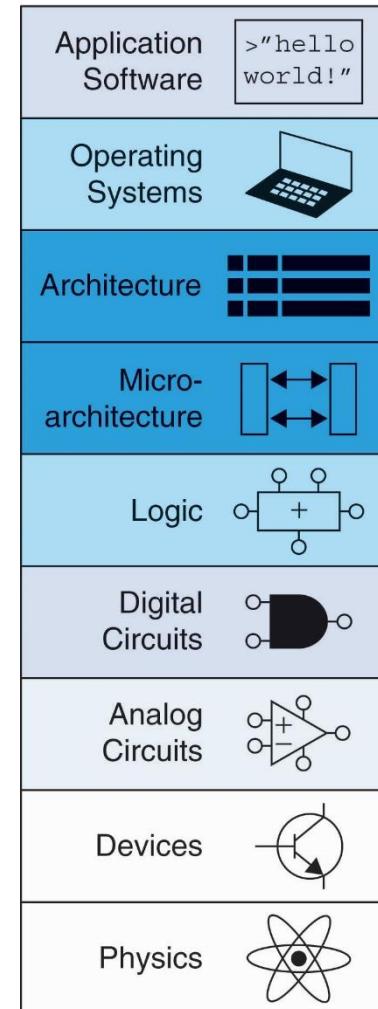
Digital Design & Computer Architecture

Sarah Harris & David Harris

Chapter 8: Memory Systems

Chapter 8 :: Topics

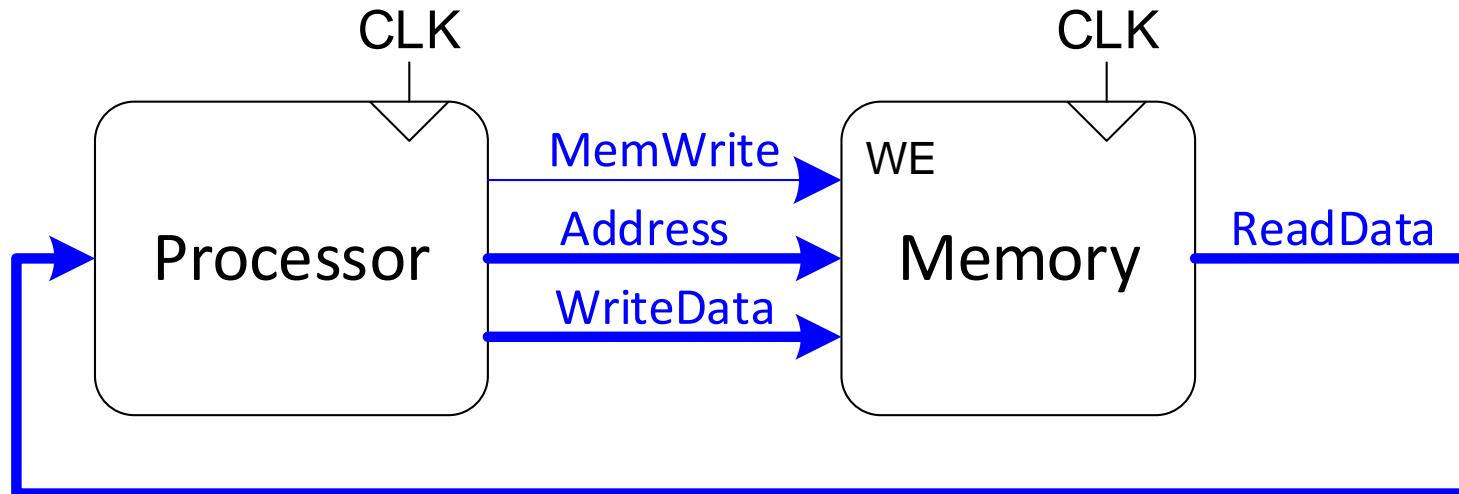
- **Introduction**
- **Memory System Performance Analysis**
- **Caches**
- **Virtual Memory**
- **Memory-Mapped I/O**
- **Summary**



Introduction

- Computer performance depends on:
 - Processor performance
 - Memory system performance

Processor / Memory Interface:

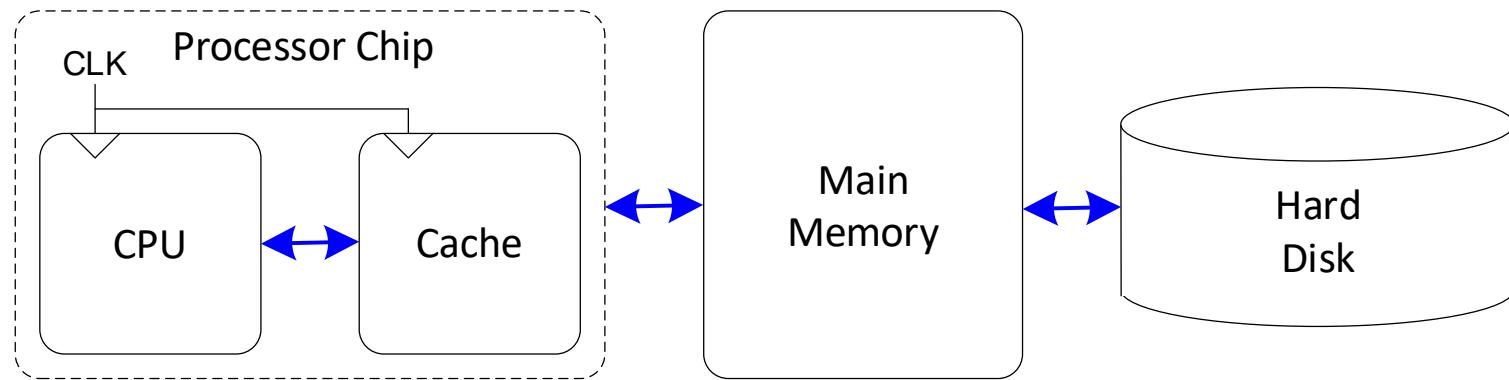


Memory System Challenge

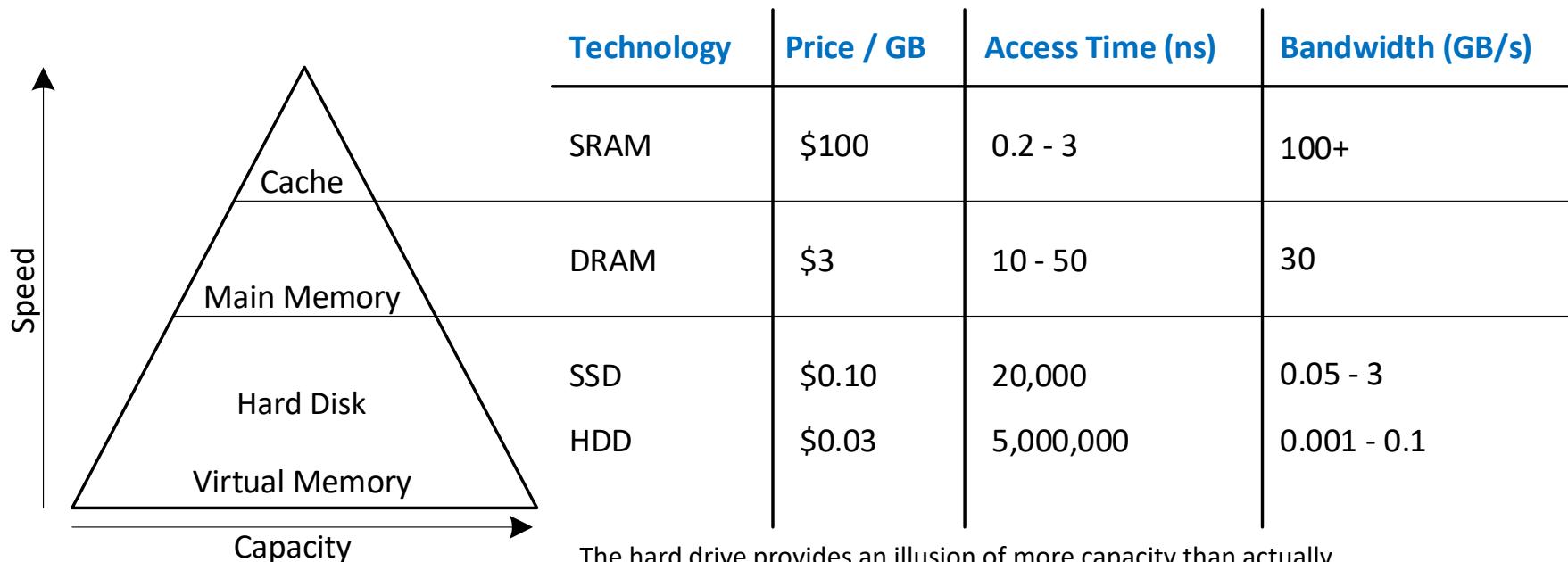
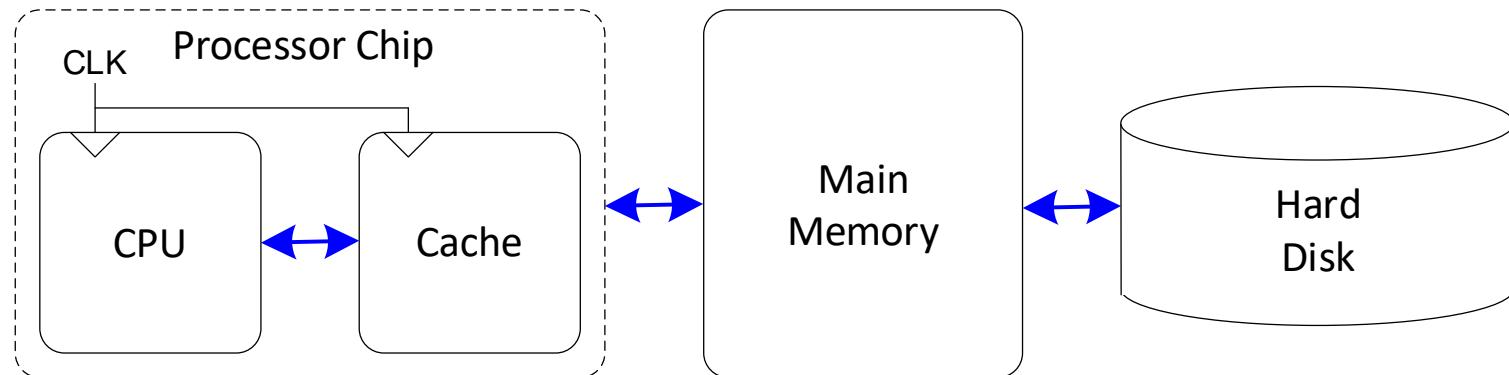
- Make memory system appear as fast as processor
- Use hierarchy of memories
- Ideal memory:
 - **Fast**
 - **Cheap** (inexpensive)
 - **Large** (capacity)
- **But can only choose two!**



Memory Hierarchy



Memory Hierarchy



Locality

Exploit locality to make memory accesses fast:

- **Temporal Locality:**
 - Locality in time
 - If data used recently, likely to use it again soon
 - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
- **Spatial Locality:**
 - Locality in space
 - If data used recently, likely to use nearby data soon
 - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too

Chapter 7: Microarchitecture

Memory Performance

Memory Performance

- **Hit:** data found in that level of memory hierarchy
- **Miss:** data not found (must go to next level)

$$\begin{aligned}\text{Hit Rate} &= \# \text{ hits} / \# \text{ memory accesses} \\ &= 1 - \text{Miss Rate}\end{aligned}$$

$$\begin{aligned}\text{Miss Rate} &= \# \text{ misses} / \# \text{ memory accesses} \\ &= 1 - \text{Hit Rate}\end{aligned}$$

- **Average memory access time (AMAT):** average time for processor to access data

$$\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$$

Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- What are the **cache hit and miss rates?**

Memory Performance Example 1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- What are the **cache hit and miss rates?**

$$\text{Hit Rate} = 1250/2000 = 0.625$$

$$\text{Miss Rate} = 750/2000 = 0.375 = 1 - \text{Hit Rate}$$

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1 \text{ cycle}$, $t_{MM} = 100 \text{ cycles}$
- What is the **AMAT** (average memory access time) of the program from Example 1?

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1 \text{ cycle}$, $t_{MM} = 100 \text{ cycles}$
- What is the **AMAT** (average memory access time) of the program from Example 1?

$$\begin{aligned}\text{AMAT} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= \mathbf{38.5 \text{ cycles}}\end{aligned}$$

Gene Amdahl, 16/11/1922 – 10/11/2015

- **Amdahl's Law:** the effort spent increasing the performance of a subsystem is wasted unless the subsystem affects a large percentage of overall performance
- Co-founded 3 companies, including one called Amdahl Corporation in 1970



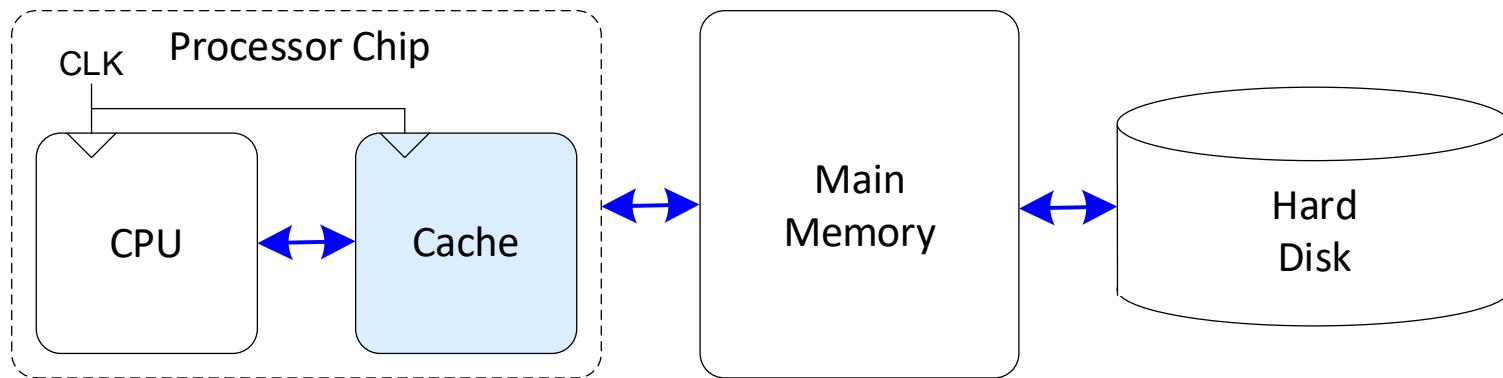
Making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's instructions are loads and stores, a tenfold memory system improvement means only a 1.82-fold improvement in program performance. This general principle is called *Amdahl's Law*.

Chapter 7: Microarchitecture

Caches

Cache

- Highest level in memory hierarchy
- Fast (typically ~ 1 cycle access time)
- Ideally supplies most data to processor
- Usually holds most recently accessed data



Cache Design Questions

- What data is held in the cache?
- How is data found?
- What data is replaced?

We focus on data loads, but stores follow the same principles.

What data is held in the cache?

- Ideally, cache anticipates needed data and puts it in cache
- But impossible to predict future
- Use past to predict future – temporal and spatial locality:
 - **Temporal locality:** copy newly accessed data into cache
 - **Spatial locality:** copy neighboring data into cache too

Cache Terminology

- **Capacity (C):**
 - number of data bytes in cache
- **Block size (b):**
 - bytes of data brought into cache at once
- **Number of blocks ($B = C/b$):**
 - number of blocks in cache: $B = C/b$
- **Degree of associativity (N):**
 - number of blocks in a set
- **Number of sets ($S = B/N$):**
 - each memory address maps to exactly one cache set

How is data found?

- Cache organized into S sets
- Each memory address maps to exactly one set
- Caches categorized by # of blocks in a set:
 - **Direct mapped:** 1 block per set
 - **N -way set associative:** N blocks per set
 - **Fully associative:** all cache blocks in 1 set
- Examine each organization for a cache with:
 - Capacity ($C = 8$ words)
 - Block size ($b = 1$ word)
 - So, number of blocks ($B = 8$)

Example Cache Parameters

- $C = 8$ words (capacity)
- $b = 1$ word (block size)
- So, $B = 8$ (# of blocks)

Ridiculously small, but will illustrate organizations

Chapter 7: Microarchitecture

Direct-Mapped Caches

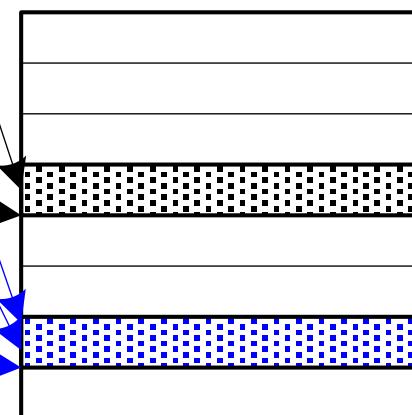
Direct Mapped Cache

Address	
11...11111100	mem[0xFF...FC]
11...11111000	mem[0xFF...F8]
11...11110100	mem[0xFF...F4]
11...11110000	mem[0xE8...E0]
11...11101100	mem[0xFF...EC]
11...11101000	mem[0xFF...E8]
11...11100100	mem[0xFF...E4]
11...11100000	mem[0xFF...E0]
⋮	⋮
00...00100100	mem[0x00...24]
00...00100000	mem[0x00...20]
00...00011100	mem[0x00...1C]
00...00011000	mem[0x00...18]
00...00010100	mem[0x00...14]
00...00010000	mem[0x00...10]
00...00001100	mem[0x00...0C]
00...00001000	mem[0x00...08]
00...00000100	mem[0x00...04]
00...00000000	mem[0x00...00]

2^{30} Word Main Memory

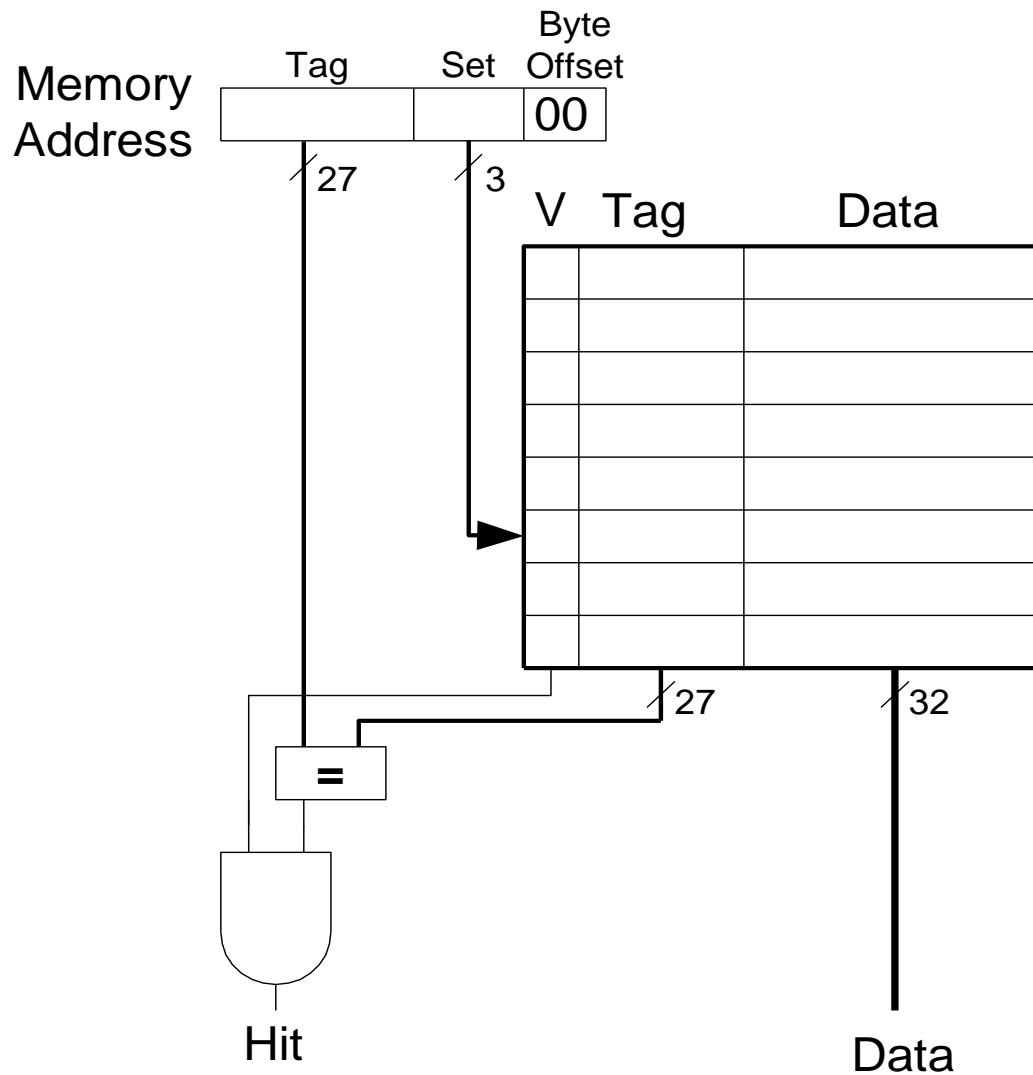
2^3 Word Cache

Cache Terminology:
 $C = 8$ words (capacity)
 $b = 1$ word (block size)
 $B = 8$ (# of blocks)
 $N = 1$ (# of blocks in a set)
 $S = B/N = 8$ (# of sets)



Set Number
7 (111)
6 (110)
5 (101)
4 (100)
3 (011)
2 (010)
1 (001)
0 (000)

Direct Mapped Cache Hardware



8-entry x
(1+27+32)-bit
SRAM

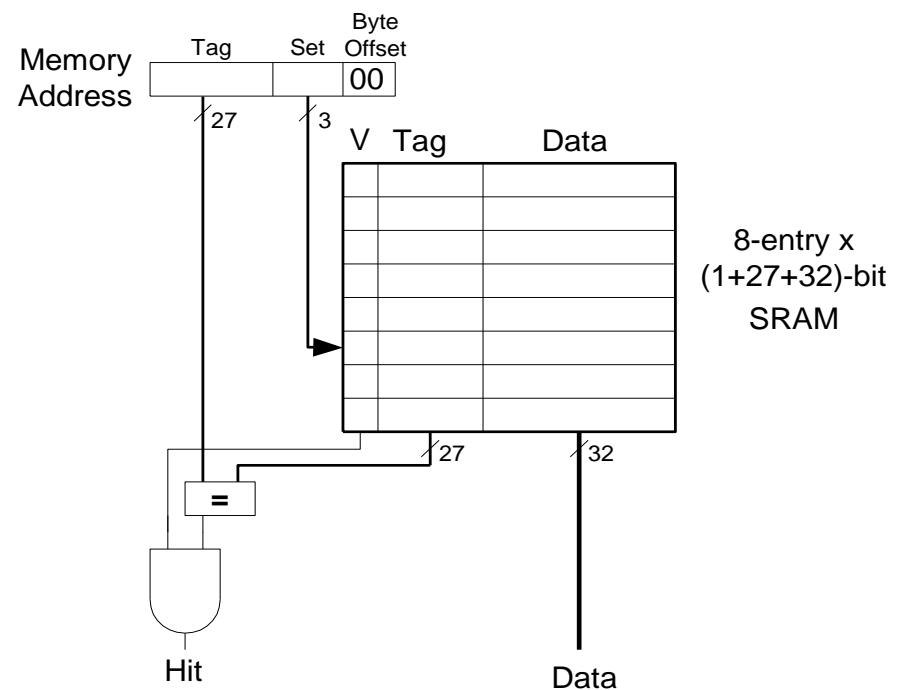
Sometimes, such as when the computer first starts up, the cache sets contain no data at all. **The cache uses a valid bit, V, for each set to indicate whether the set holds meaningful data.** If the valid bit is 0, the contents are meaningless.

Direct Mapped Cache Performance

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in RISC-V assembly code. Assume that the cache is initially empty. What is the miss rate?

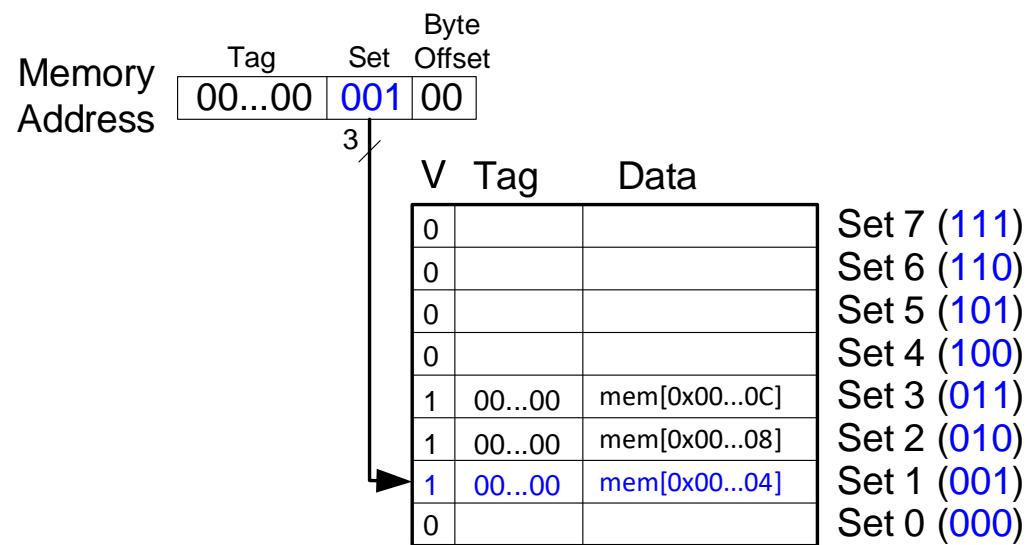
```
addi s0, zero, 5
addi s1, zero, 0
LOOP: beq s0, zero, DONE
      lw s2, 4(s1)
      lw s3, 12(s1)
      lw s4, 8(s1)
      addi s0, s0, -1
      j LOOP
DONE:
```



Direct Mapped Cache Performance

Solution The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is $3/15 = 20\%$.

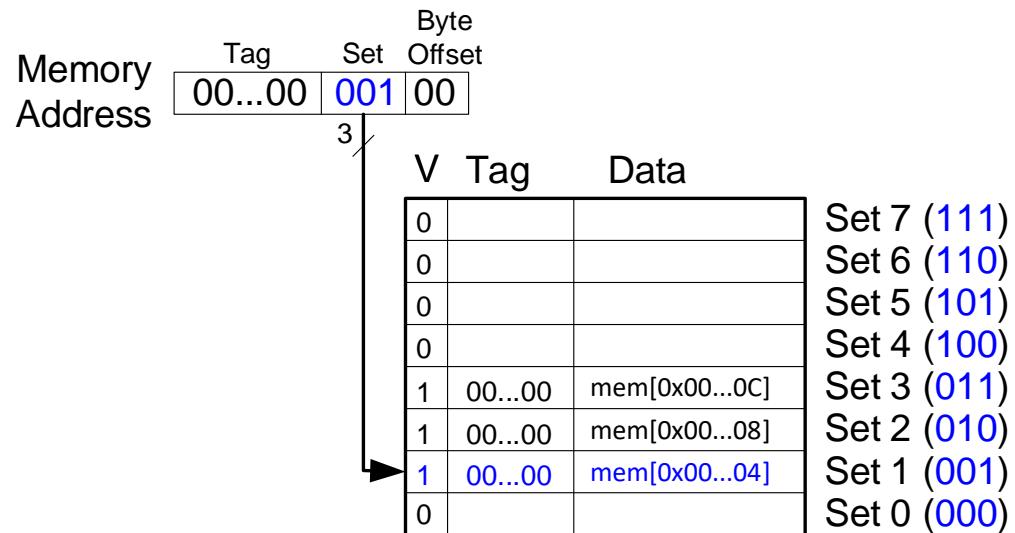
```
# RISC-V assembly code
      addi s0, zero, 5
      addi s1, zero, 0
LOOP:   beq s0, zero, DONE
      lw    s2, 4(s1)
      lw    s3, 12(s1)
      lw    s4, 8(s1)
      addi s0, s0, -1
      j     LOOP
DONE:
```



Direct Mapped Cache Performance

RISC-V assembly code

```
addi s0, zero, 5  
addi s1, zero, 0  
LOOP: beq s0, zero, DONE  
      lw s2, 4(s1)  
      lw s3, 12(s1)  
      lw s4, 8(s1)  
      addi s0, s0, -1  
      j LOOP  
  
DONE:
```

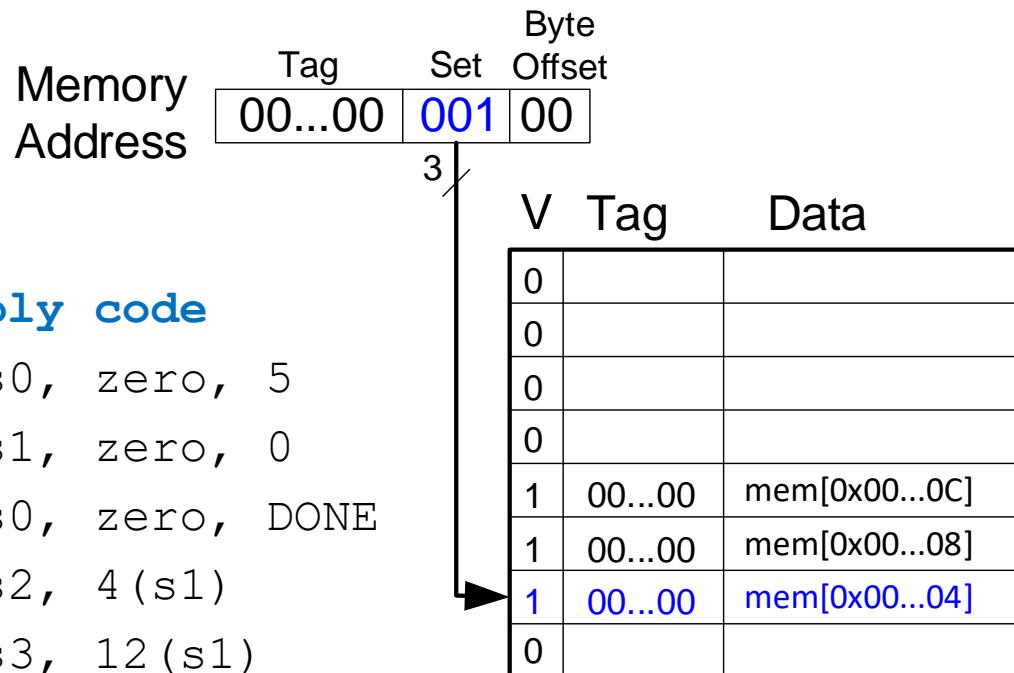


The cache is accessed using the 32-bit address.

- The two least significant bits, the *byte offset* bits, are ignored for word accesses.
- The next three bits, the *set* bits, specify the entry or set in the cache.
- A load instruction reads the specified entry from the cache and checks the tag and valid bits.
- If the tag matches the most significant 27 bits of the requested address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Direct Mapped Cache Performance

```
# RISC-V assembly code
addi s0, zero, 5
addi s1, zero, 0
LOOP:   beq s0, zero, DONE
        lw s2, 4(s1)
        lw s3, 12(s1)
        lw s4, 8(s1)
        addi s0, s0, -1
        j LOOP
DONE:
```

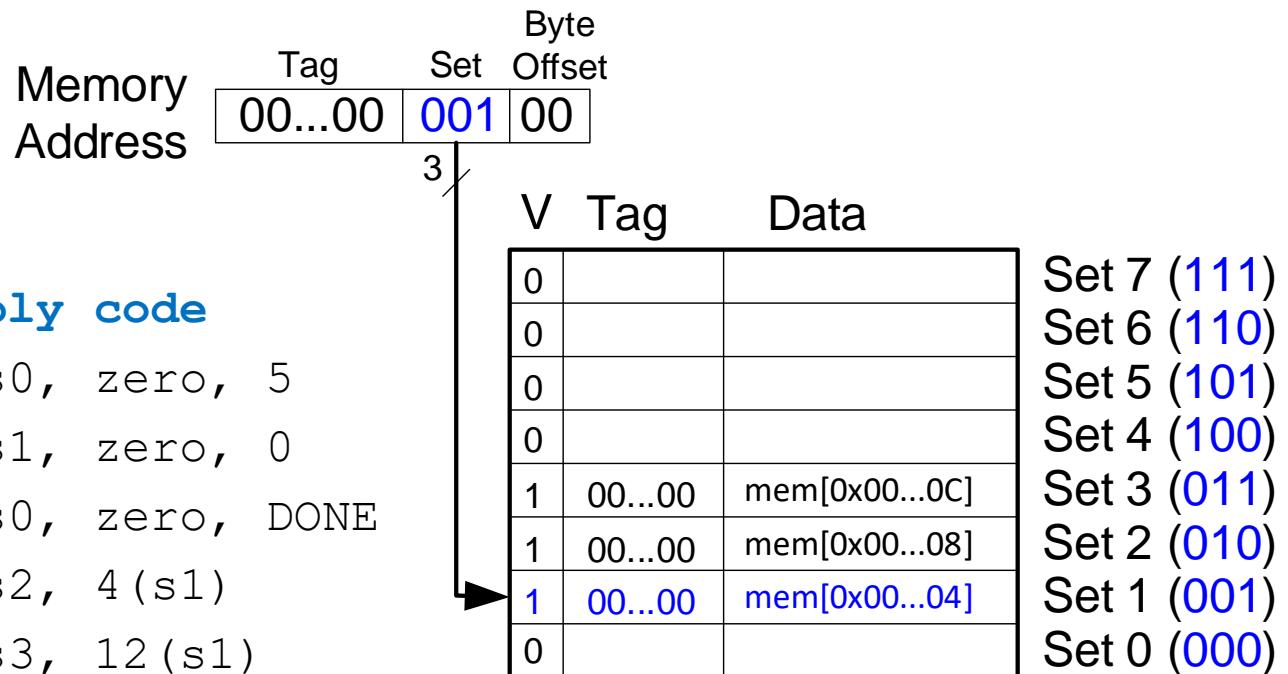


$$\text{Miss Rate} = \frac{3}{15} \\ = 20\%$$

Assume that
the cache is
initially empty

Direct Mapped Cache Performance

```
# RISC-V assembly code
addi s0, zero, 5
addi s1, zero, 0
LOOP:   beq s0, zero, DONE
        lw s2, 4(s1)
        lw s3, 12(s1)
        lw s4, 8(s1)
        addi s0, s0, -1
        j LOOP
DONE:
```

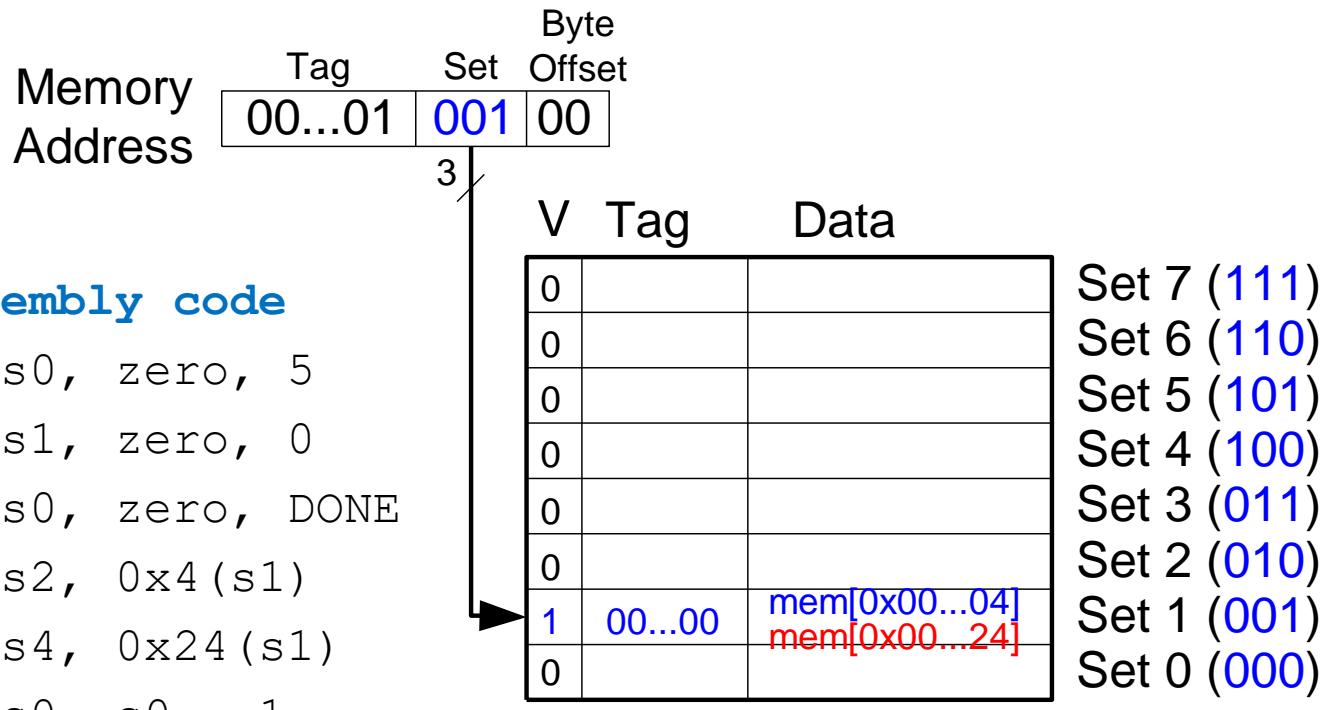


$$\begin{aligned}\text{Miss Rate} &= 3/15 \\ &= 20\%\end{aligned}$$

Temporal Locality
Compulsory Misses

Direct Mapped Cache: Conflict Miss

```
# RISC-V assembly code
    addi s0, zero, 5
    addi s1, zero, 0
LOOP:   beq s0, zero, DONE
        lw    s2, 0x4(s1)
        lw    s4, 0x24(s1)
        addi s0, s0, -1
        j     LOOP
DONE:
```



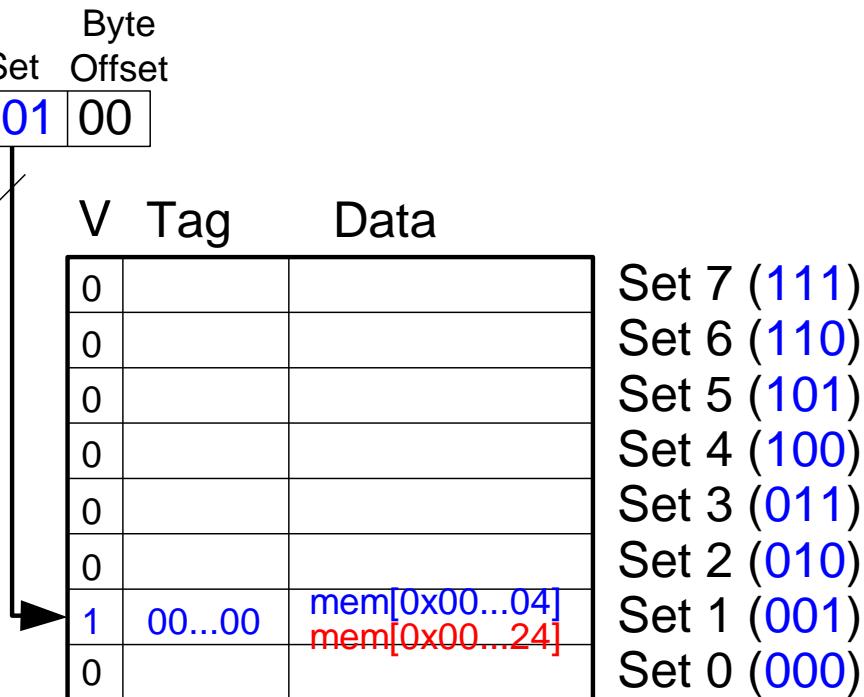
Miss Rate

Direct Mapped Cache: Conflict Miss

RISC-V assembly code

```
addi s0, zero, 5
addi s1, zero, 0
LOOP: beq s0, zero, DONE
      lw    s2, 0x4(s1)
      lw    s4, 0x24(s1)
      addi s0, s0, -1
      j     LOOP
```

DONE:



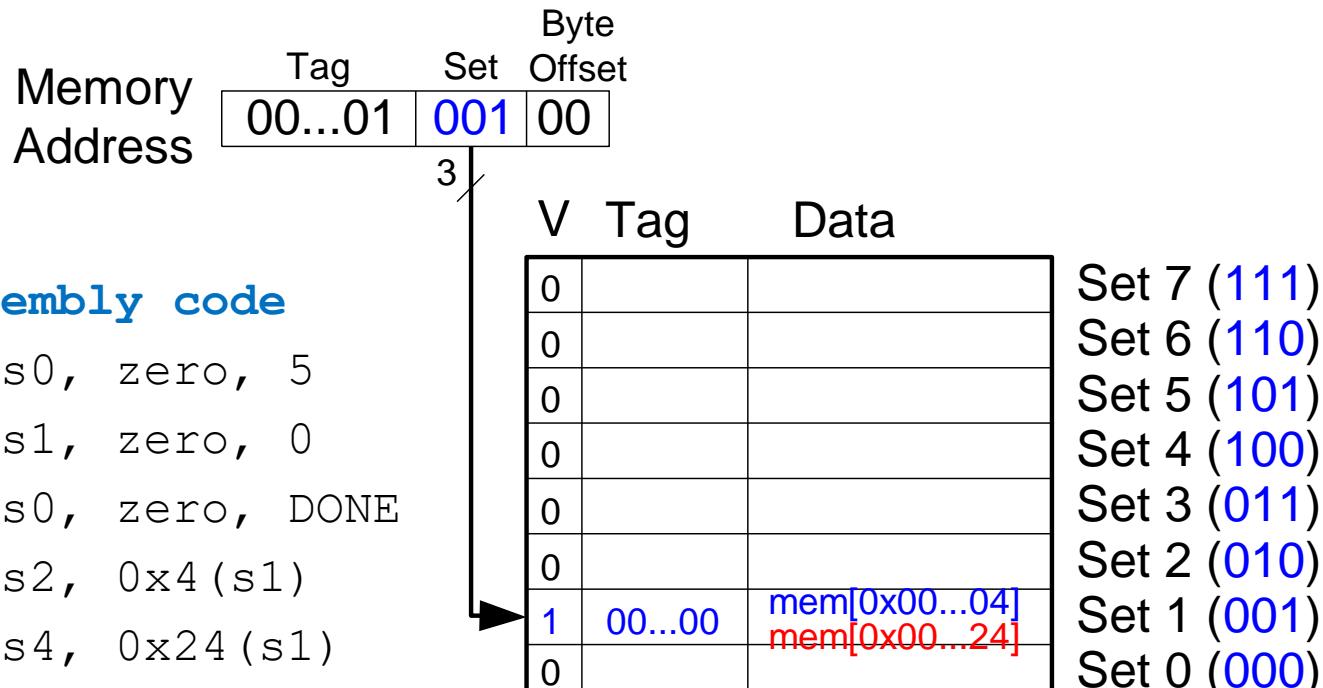
$$\text{Miss Rate} = 10/10 = 100\%$$

Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then, data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must re-fetch data at address 0x4, evicting data from address 0x24. The two addresses conflict and the miss rate is 100%.

Direct Mapped Cache: Conflict Miss

RISC-V assembly code

```
addi s0, zero, 5
addi s1, zero, 0
LOOP: beq s0, zero, DONE
      lw    s2, 0x4(s1)
      lw    s4, 0x24(s1)
      addi s0, s0, -1
      j     LOOP
DONE:
```



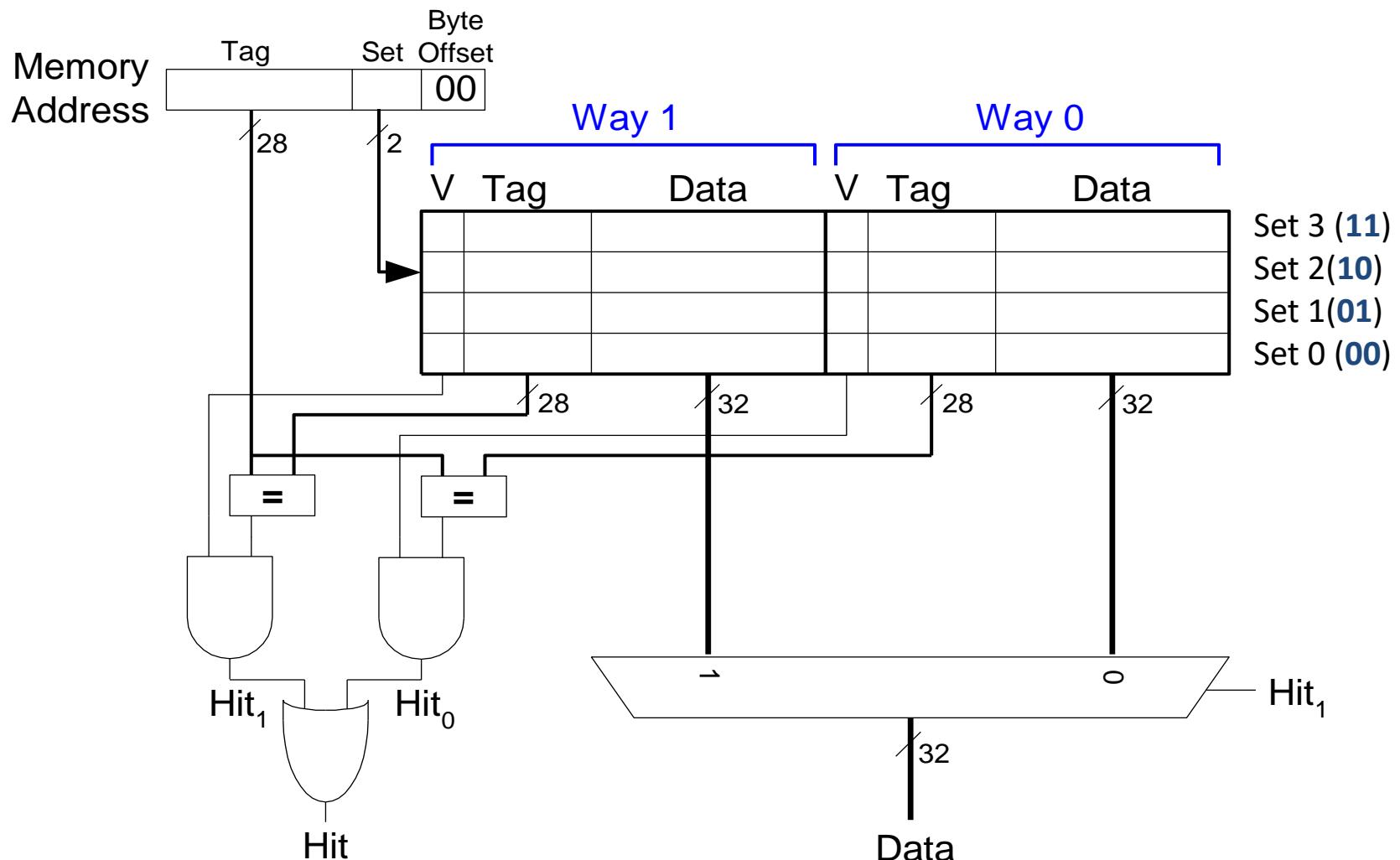
$$\begin{aligned} \text{Miss Rate} &= 10/10 \\ &= 100\% \end{aligned}$$

Conflict Misses

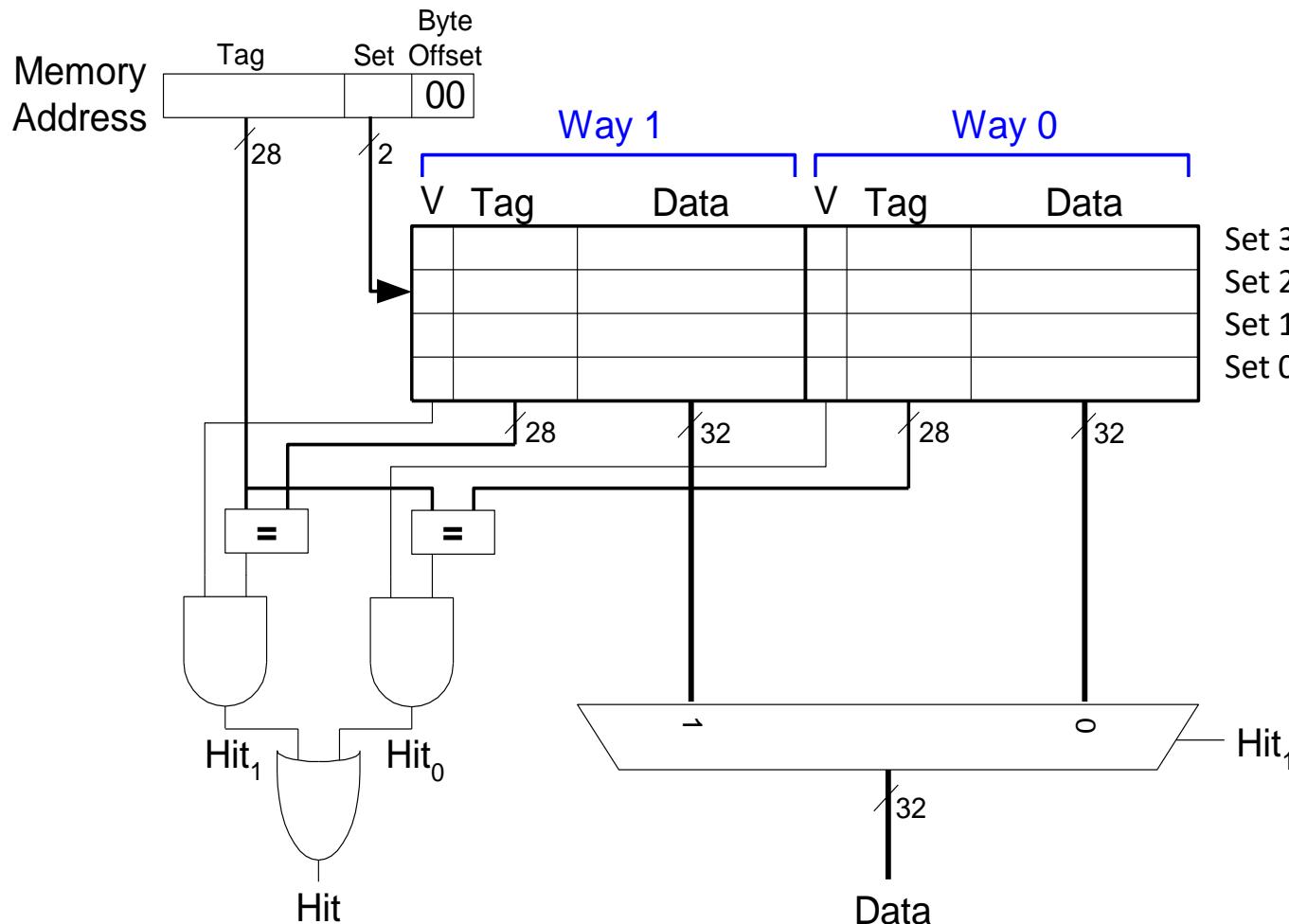
Chapter 7: Microarchitecture

Associative Caches

N-Way Set Associative Cache



N-Way Set Associative Cache



An *N-way set associative cache* reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found.

Each memory address still maps to a specific set, but it can map to any one of the *N* blocks in the set.

N-Way Set Assoc. Cache Performance

RISC-V assembly code

```
addi s0, zero, 5  
addi s1, zero, 0  
LOOP: beq s0, zero, DONE  
      lw   s2, 0x4(s1)  
      lw   s4, 0x24(s1)  
      addi s0, s0, -1  
      j    LOOP  
  
DONE :
```

Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1. In the next four iterations, the cache hits. Hence, the miss rate is $2/10 = 20\%$, which is a significant improvement over the 100% miss rate of the direct mapped cache of the same size in the previous example.

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

N-Way Set Assoc. Cache Performance

RISC-V assembly code

```
addi s0, zero, 5  
addi s1, zero, 0  
LOOP: beq s0, zero, DONE  
      lw   s2, 0x4(s1)  
      lw   s4, 0x24(s1)  
      addi s0, s0, -1  
      j    LOOP  
  
DONE:
```

$$\begin{aligned}\text{Miss Rate} &= 2/10 \\ &= 20\%\end{aligned}$$

Associativity reduces conflict misses

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0		Conflict Misses	Set 0

Fully Associative Cache

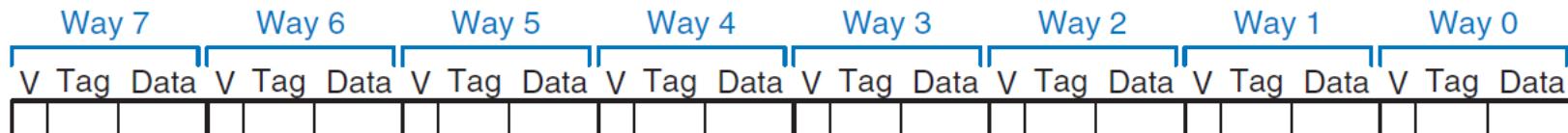


Reduces conflict misses

Expensive to build

Fully Associative Cache

- A *fully associative* cache contains a single set with B ways, where B is the number of blocks.
- A memory address can map to a block in any of these ways. A fully associative cache is another name for a B -way set associative cache with one set.
- Upon a data request, eight tag comparisons (not shown) must be made because the data could be in any block.
- An 8:1 multiplexer chooses the proper data if a hit occurs.
- Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.



Reduces conflict misses
Expensive to build

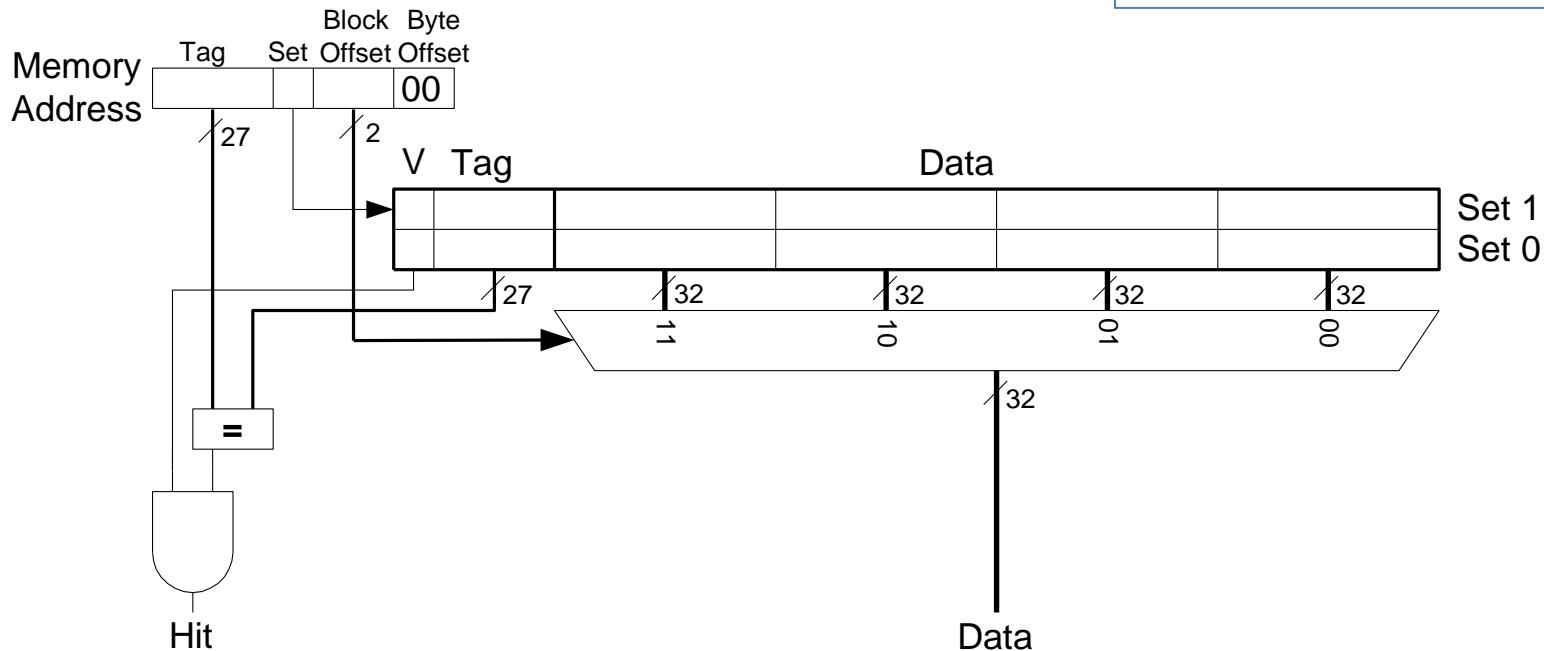
Chapter 7: Microarchitecture

Spatial Locality

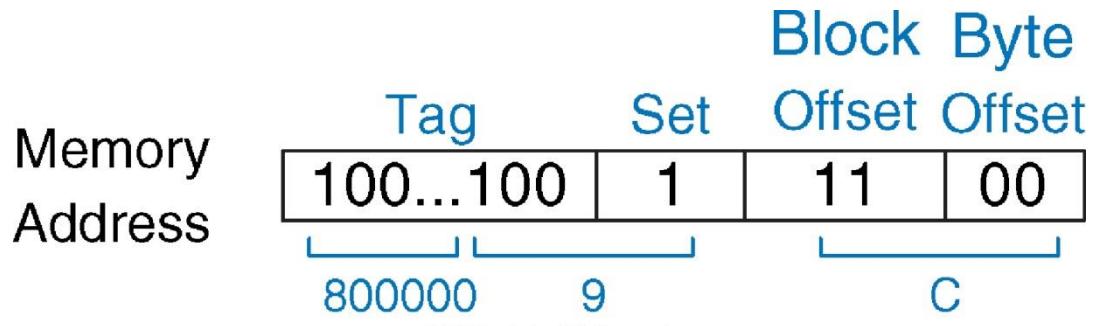
Spatial Locality

- Increase block size:
 - Block size, $b = 4 \text{ words}$
 - $C = 8 \text{ words}$
 - Direct mapped (1 block per set)
 - Number of blocks, $B = 2$ ($C/b = 8/4 = 2$)

- Only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block.
- The multiplexer is controlled by the $\log_2 4 = 2$ block offset bits of the address.
- The most significant 27 address bits form the tag.
- Only one tag is needed for the entire block because the words in the block are at consecutive addresses.



Cache with Larger Block Size



The diagram illustrates the memory address format and its mapping to a cache. The **Memory Address** is divided into several fields:

- Tag**: 27 bits, used for tag comparison.
- Set Offset**: 2 bits, used to select a set in the cache.
- Offset**: 8 bits, used to index into a block of data.

The **Set Offset** field is further divided into:

- V**: 1 bit, indicating if the offset is valid.
- Tag**: 2 bits, part of the tag for tag comparison.
- Data**: 32 bits, the actual data stored in the cache.

The cache organization is shown as two sets of four blocks each:

- Set 1**: Contains four blocks, indexed by the Set Offset field.
- Set 0**: Contains four blocks, indexed by the Set Offset field.

A logic diagram at the bottom shows the comparison of the **Set Offset** bits with the **Set** bits of the cache blocks. A **Hit** signal is generated if there is a match. The **Data** output is also indicated.

- The *byte offset* bits are always 0 for word accesses.
 - The next $\log_2 b = 2$ *block offset* bits indicate the word within the block and
 - the next bit indicates the *set*.

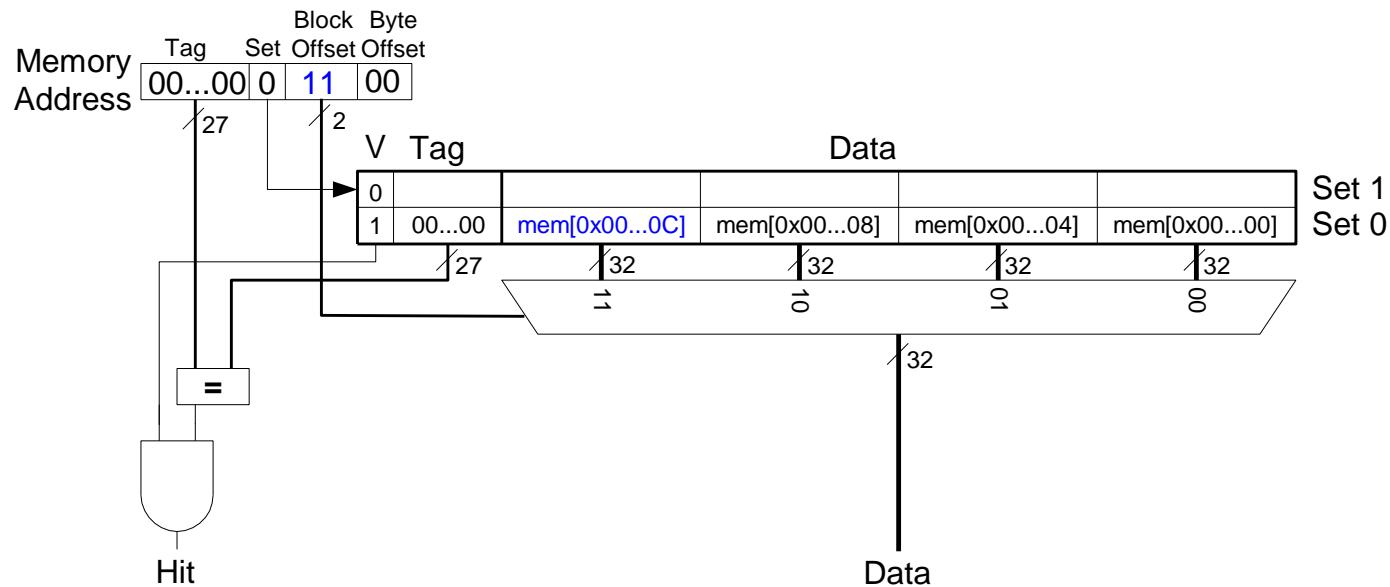
- Word 0x8000009C maps to set 1, word 3 in the cache.
 - The principle of *using larger block sizes to exploit spatial locality* also applies to associative caches.

Cache Perf. with Spatial Locality

```
addi s0, zero, 5  
addi s1, zero, 0  
LOOP:    beq s0, zero, DONE  
          lw   s2, 4(s1)  
          lw   s3, 12(s1)  
          lw   s4, 8(s1)  
          addi s0, s0, -1  
          j    LOOP
```

DONE:

Miss Rate



Cache Perf. with Spatial Locality

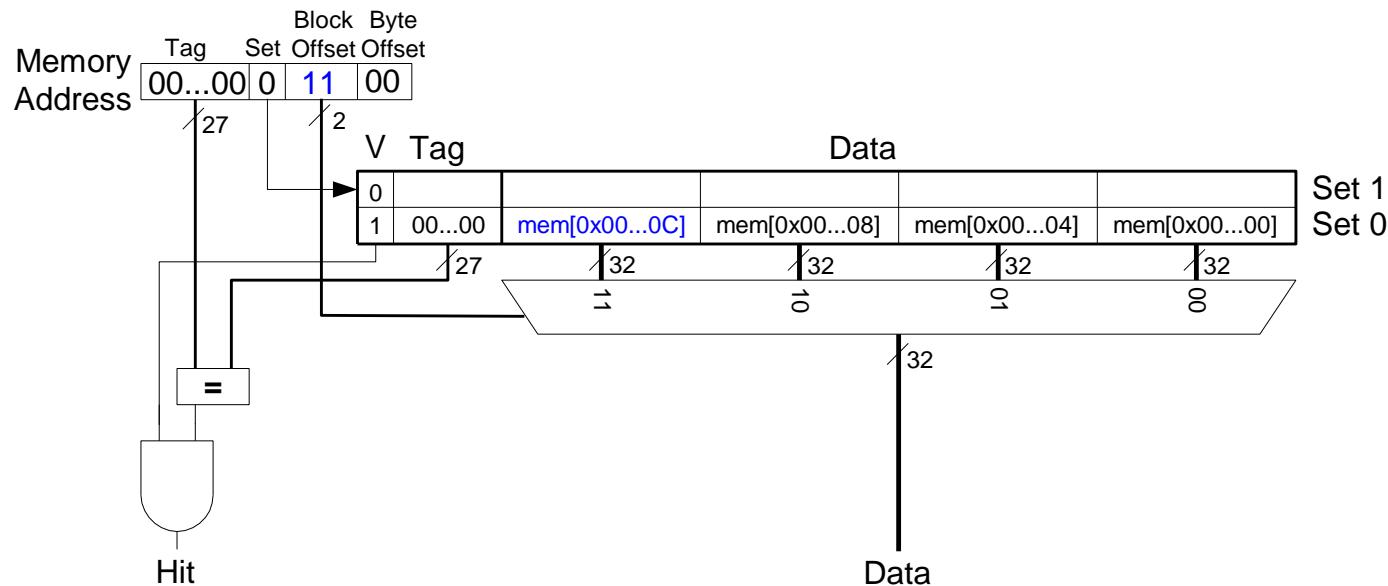
```
addi s0, zero, 5  
addi s1, zero, 0  
LOOP:   beq s0, zero, DONE  
        lw   s2, 4 (s1)  
        lw   s3, 12 (s1)  
        lw   s4, 8 (s1)  
        addi s0, s0, -1  
        j    LOOP
```

DONE:

$$\text{Miss Rate} = 1/15$$

$$= 6.67\%$$

On the first loop iteration, the cache misses the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is $1/15 = 6.67\%$.



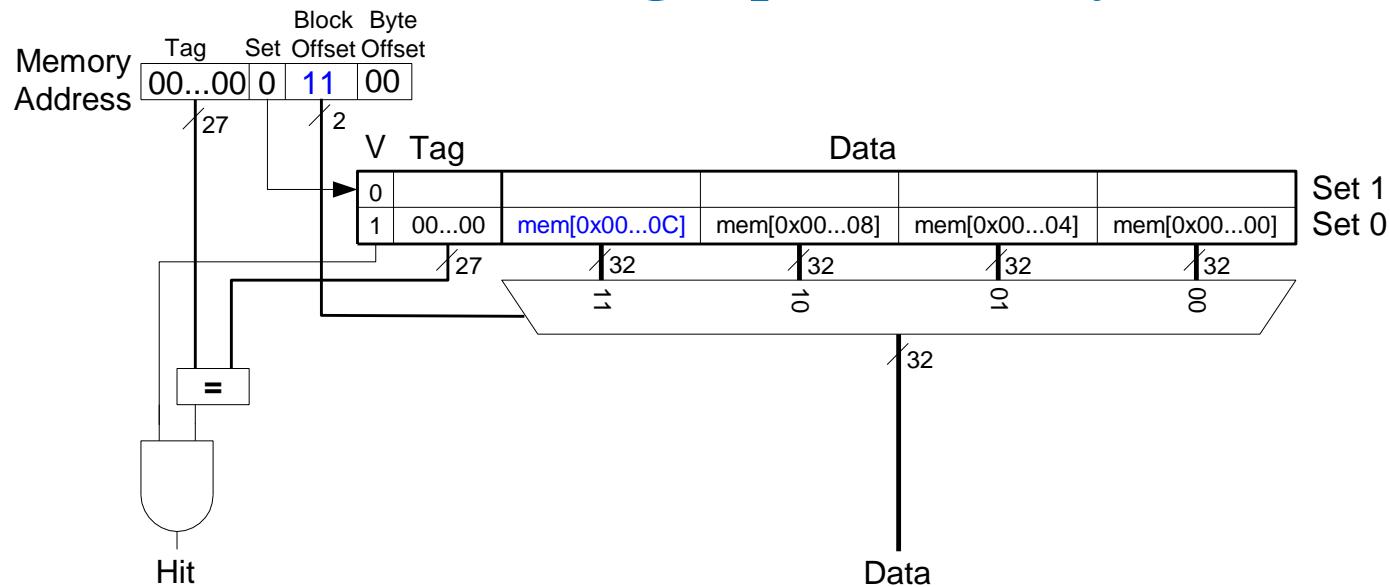
Cache Perf. with Spatial Locality

```
addi s0, zero, 5  
addi s1, zero, 0  
LOOP:    beq s0, zero, DONE  
          lw   s2, 4(s1)  
          lw   s3, 12(s1)  
          lw   s4, 8(s1)  
          addi s0, s0, -1  
          j    LOOP
```

DONE:

$$\text{Miss Rate} = \frac{1}{15} \\ = 6.67\%$$

Larger blocks reduce compulsory misses through spatial locality



Types of Misses

- **Compulsory:** first time data accessed
- **Capacity:** cache too small to hold all data of interest
- **Conflict:** data of interest maps to same location in cache

Miss penalty: time it takes to retrieve a block from lower level of hierarchy

Cache Organization Recap

- **Capacity:** C
- **Block size:** b
- **Number of blocks in cache:** $B = C/b$
- **Number of blocks in a set:** N
- **Number of sets:** $S = B/N$

Organization	Number of Ways (N)	Number of Sets ($S = B/N$)
Direct Mapped	1	B
N-Way Set Associative	$1 < N < B$	B / N
Fully Associative	B	1

Chapter 7: Microarchitecture

Cache Replacement Policy

Replacement Policy

- Cache is too small to hold all data of interest at once
- If cache full: program accesses data X and evicts data Y
- ***Capacity miss*** when access Y again
- How to choose Y to minimize chance of needing it again?
 - **Least recently used (LRU) replacement:** the least recently used block in a set evicted

LRU Replacement

RISC-V assembly

```
lw s1, 0x04(zero)  
lw s2, 0x24(zero)  
lw s3, 0x54(zero)
```

Way 1			Way 0			
V	U	Tag	Data	V	Tag	
0	0			0		Set 3 (11)
0	0			0		Set 2 (10)
0	0			0		Set 1 (01)
0	0			0		Set 0 (00)

LRU Replacement

RISC-V assembly

```
lw s1, 0x04(zero)  
lw s2, 0x24(zero)  
lw s3, 0x54(zero)
```

Way 1			Way 0			
V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
0	0			0		
0	0			0		

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

Way 1			Way 0			
V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]
0	0			0		

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

- The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache.
- $U = 0$ indicates that data in way 0 was the least recently used.

LRU Replacement

RISC-V assembly

```
lw s1, 0x04(zero)  
lw s2, 0x24(zero)  
lw s3, 0x54(zero)
```

Way 1			Way 0		
V	U	Tag	Data	V	Tag
0	0			0	
0	0			0	
1	0	00...010	mem[0x00...24]	1	00...000
0	0			0	mem[0x00...04]

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

- The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache.
- $U = 0$ indicates that data in way 0 was the least recently used.

Way 1			Way 0		
V	U	Tag	Data	V	Tag
0	0			0	
0	0			0	
1	1	00...010	mem[0x00...24]	1	00...101
0	0			0	mem[0x00...54]

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

- The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0.
- The use bit U is set to 1 to indicate that data in way 1 was the least recently used.

The use of bit U

- For a **two-way** set associative cache, the U bit indicates which way within a set was least recently used.
 - Each time one of the ways is used, U is adjusted to indicate the other way.
- For set-associative caches with more than two ways, tracking the least recently used way becomes complicated.
 - To simplify the problem, the ways are often divided into **two groups**, and U indicates which group of ways was least recently used.
 - The new block replaces a random block within the least recently used group upon replacement.
 - Such a policy is called *pseudo-LRU* and is good enough in practice.

RISC-V assembly

```
lw s1, 0x04(zero)
```

```
lw s2, 0x24(zero)
```

```
lw s3, 0x54(zero)
```

Way 1			Way 0			Set 3 (11) Set 2 (10) Set 1 (01) Set 0 (00)
V	U	Tag	Data	V	Tag	
0	0			0		
0	0			0		
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]
0	0			0		

(a)

Way 1			Way 0			Set 3 (11) Set 2 (10) Set 1 (01) Set 0 (00)
V	U	Tag	Data	V	Tag	
0	0			0		
0	0			0		
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]
0	0			0		

(b)

Chapter 7: Microarchitecture

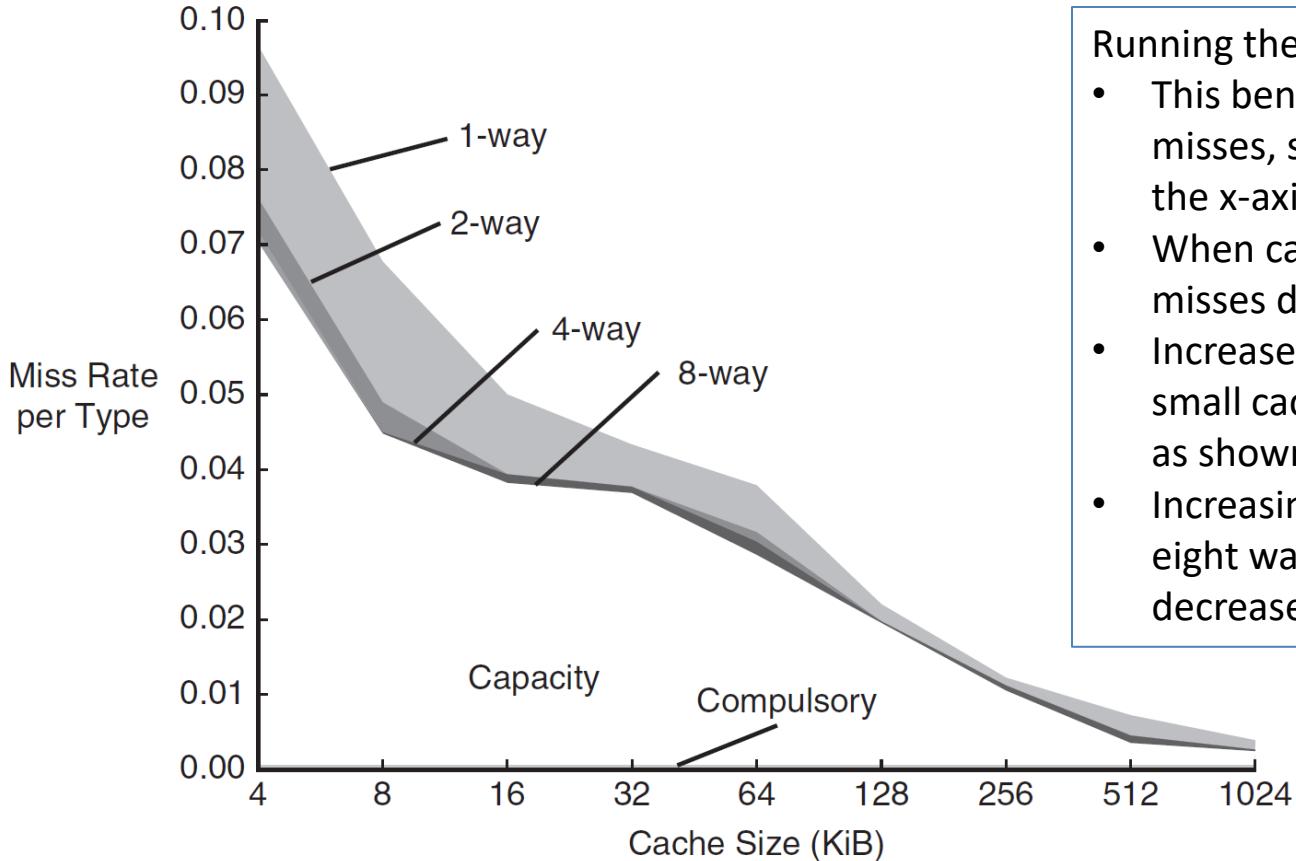
Cache Summary

Cache Summary

- **What data is held in the cache?**
 - Recently used data (temporal locality)
 - Nearby data (spatial locality)
- **How is data found?**
 - Set is determined by address of data
 - Word within block also determined by address
 - In associative caches, data could be in one of several ways
- **What data is replaced?**
 - Least-recently used way in the set

Miss Rate Trends

- Bigger caches reduce capacity misses
- Greater associativity reduces conflict misses



Running the SPEC2000 benchmark.

- This benchmark has few compulsory misses, shown by the dark region near the x-axis.
- When cache size increases, capacity misses decrease.
- Increased associativity, especially for small caches, decreases conflict misses as shown at the top of the curve.
- Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

Adapted from Patterson & Hennessy, *Computer Architecture: A Quantitative Approach*, 2011

Miss Rate Trends

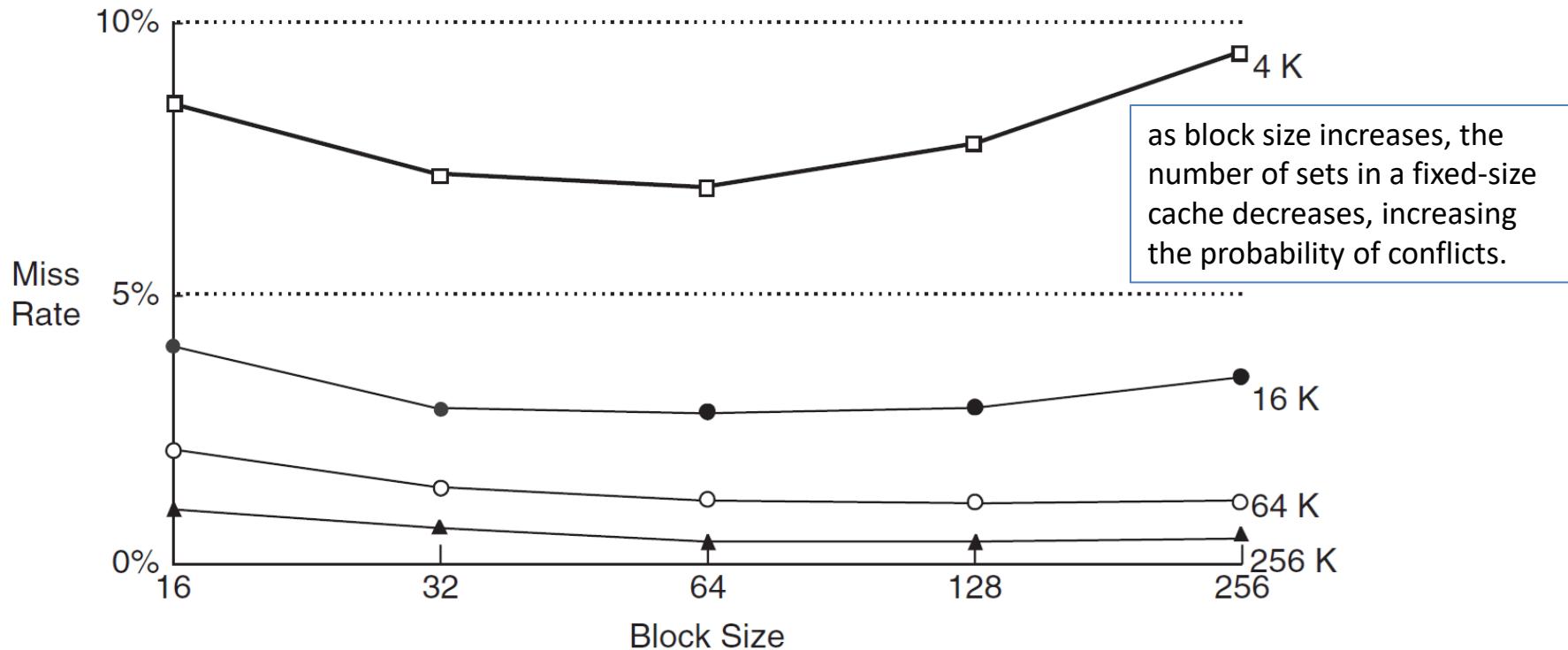


Figure 8.18 Miss rate versus block size and cache size on SPEC92 benchmark

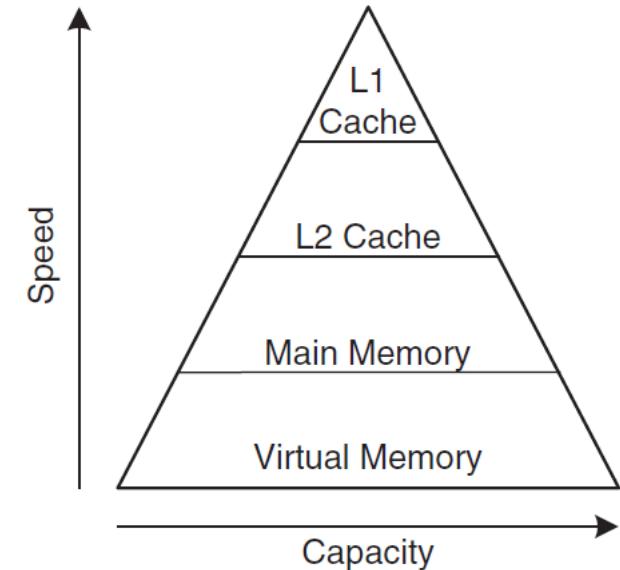
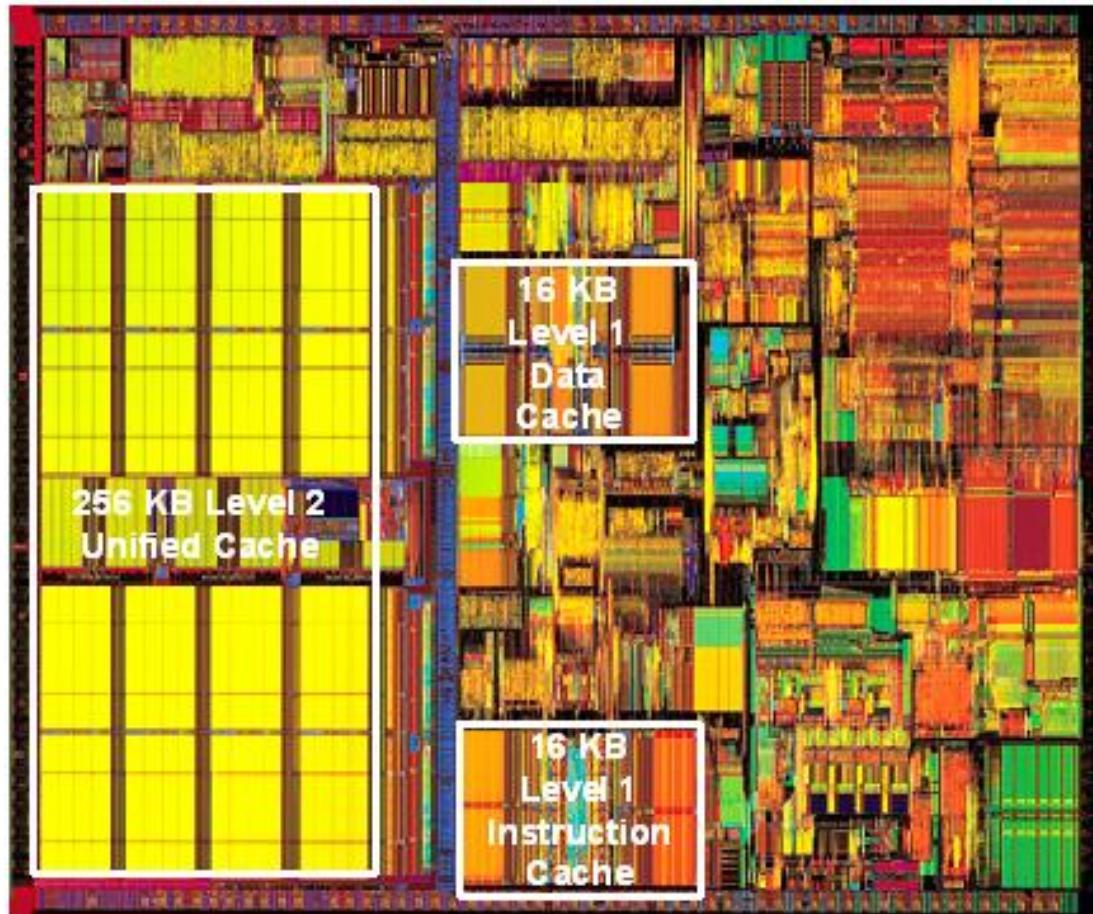
Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012

- Bigger blocks reduce compulsory misses
- Bigger blocks increase conflict misses

Multilevel Caches

- Larger caches have lower miss rates, longer access times
- Expand memory hierarchy to multiple levels of caches
- Level 1: small and fast (e.g. 16 KB, 1 cycle)
- Level 2: larger and slower (e.g. 256 KB, 2-6 cycles)
- Most modern PCs have L1, L2, and L3 cache

Intel Pentium III Die

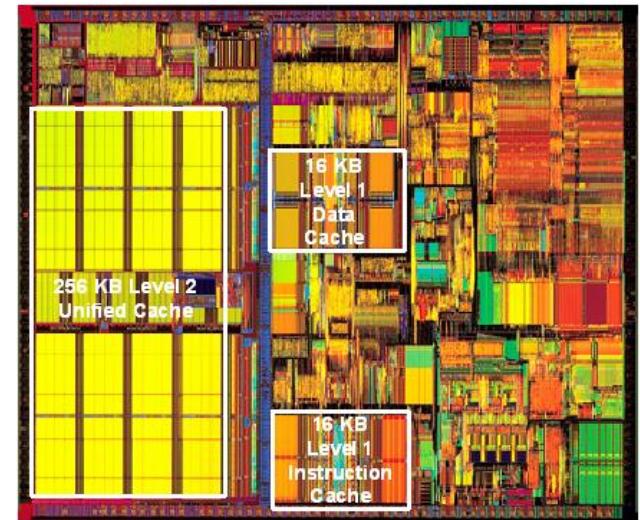
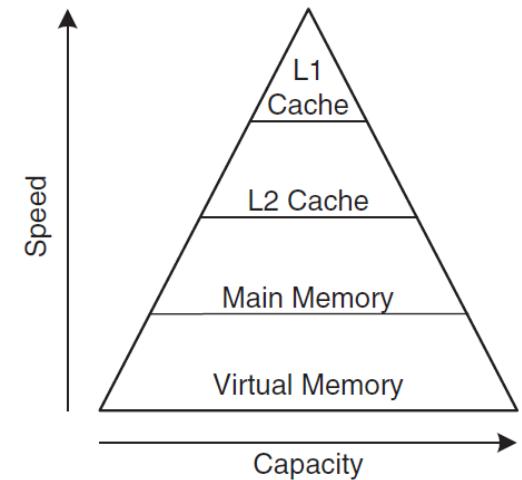


© Intel Corp.

System with Two Cache Levels

Question

Suppose a memory system with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (AMAT)?



System with Two Cache Levels

Question:

Suppose a memory system with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (AMAT)?

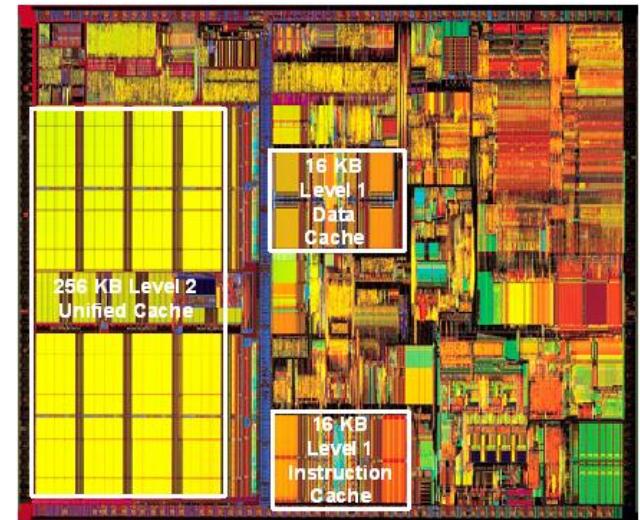
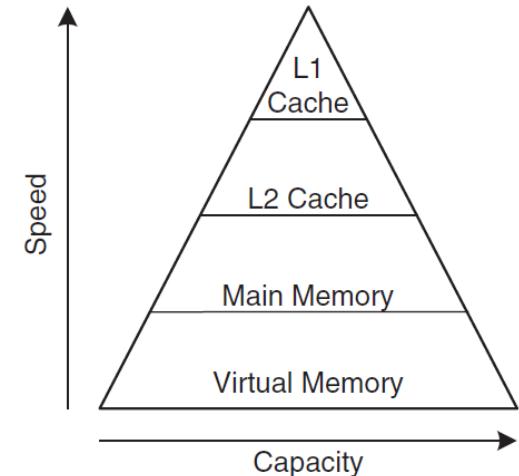
Solution

Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from the main memory. Using equation;

$$AMAT = t_{L1\text{cache}} + MR_{L1\text{cache}}(t_{L2\text{cache}} + MR_{L2\text{cache}}t_{MM})$$

we calculate the average memory access time as follows:

$$1 \text{ cycle} + 0.05[10 \text{ cycles} + 0.2(100 \text{ cycles})] = 2.5 \text{ cycles.}$$



Memory Write Policy

- The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads.
- Upon a memory store (write), the processor checks the cache. If the cache misses, the cache block is fetched from the main memory into the cache, and then the appropriate word in the cache block is written.
- If the cache hits, the word is simply written to the cache block.
- Caches are classified as either **write-through** or **write-back**.
- In a *writethrough* cache, the data written to a cache block is simultaneously written to the main memory.
- In a *write-back* cache, a **dirty bit** (D) is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise.
- Dirty cache blocks are written back to main memory only when they are evicted from the cache.
- A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache.
- *Modern caches are usually write-back because the main memory access time is so large.*

Memory Write-Through vs Write-Back

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
addi t5, zero, 0  
sw    t1, 0(t5)  
sw    t2, 12(t5)  
sw    t3, 8(t5)  
sw    t4, 4(t5)
```

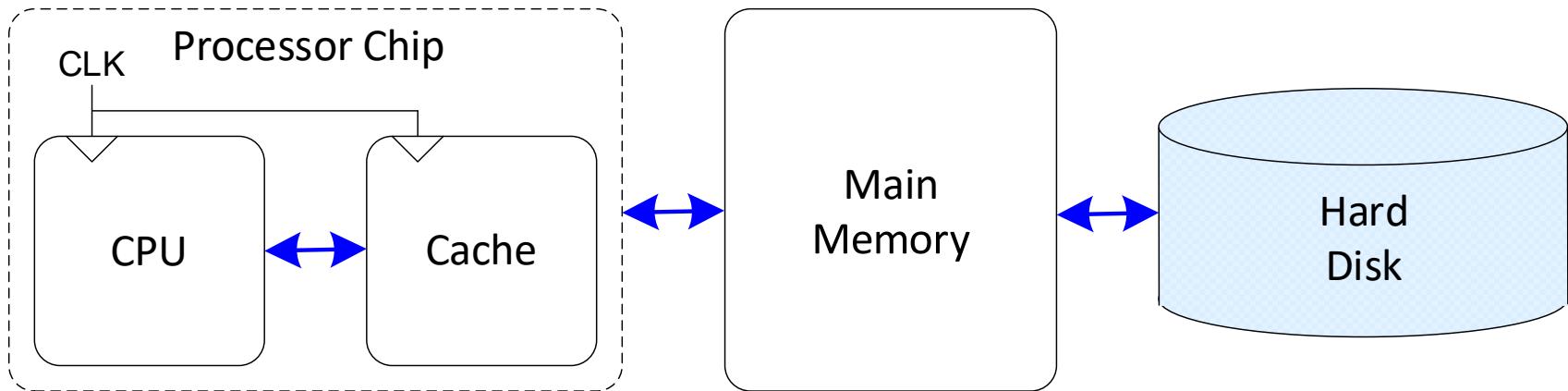
Solution All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

Chapter 7: Microarchitecture

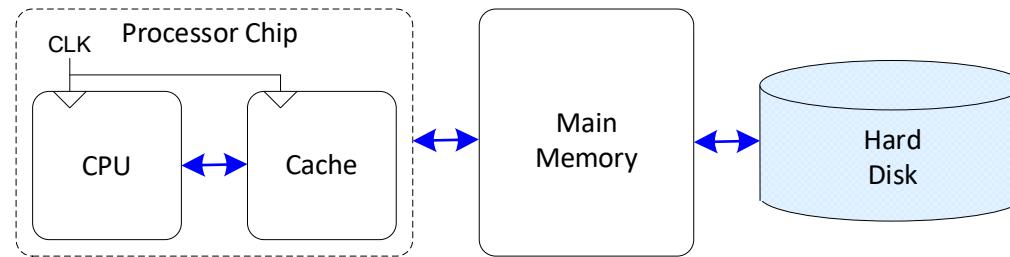
Virtual Memory

Virtual Memory

- Gives the illusion of bigger memory
- Main memory (DRAM) acts as cache for hard disk

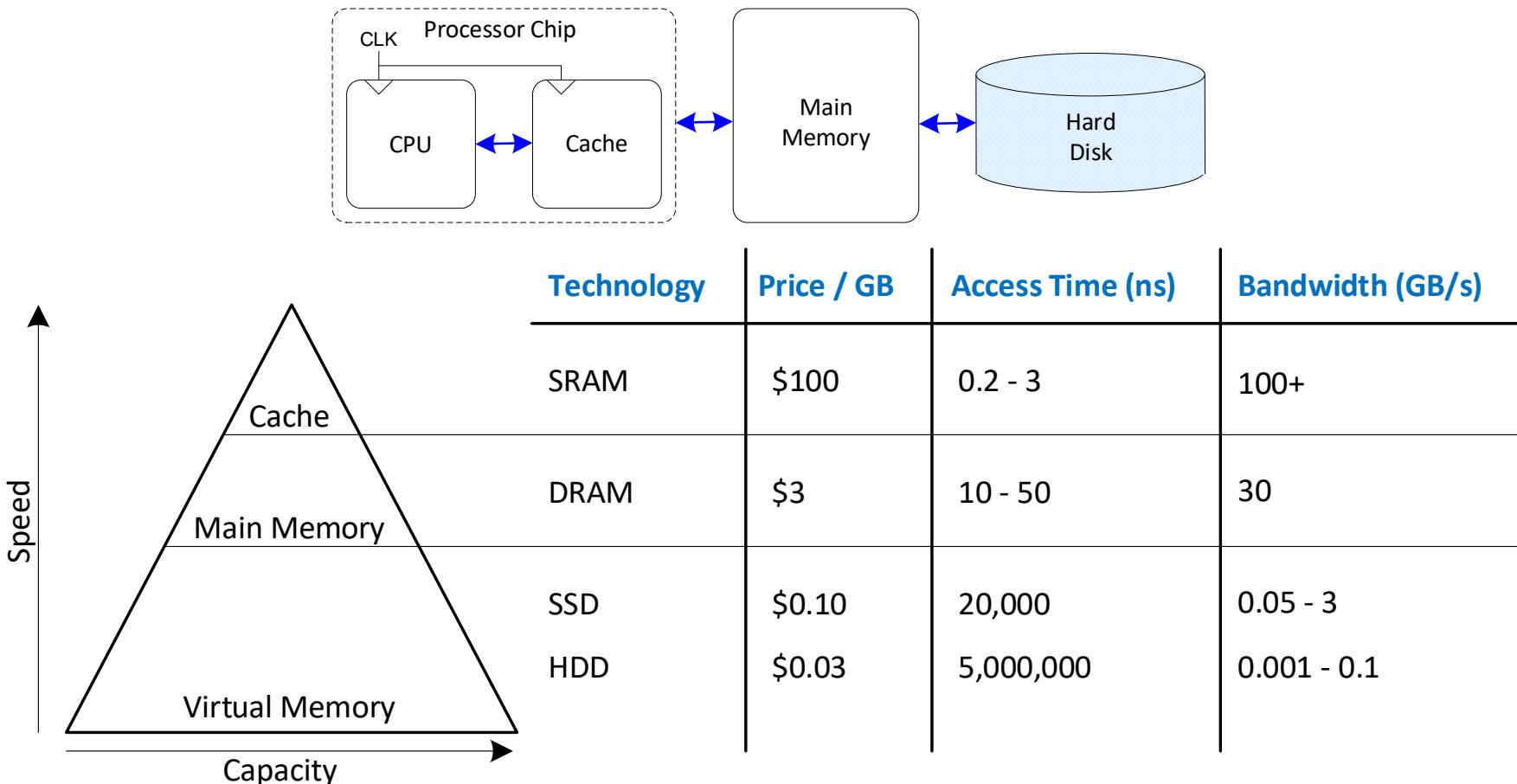


Memory Hierarchy



- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
 - Slow, Large, Cheap

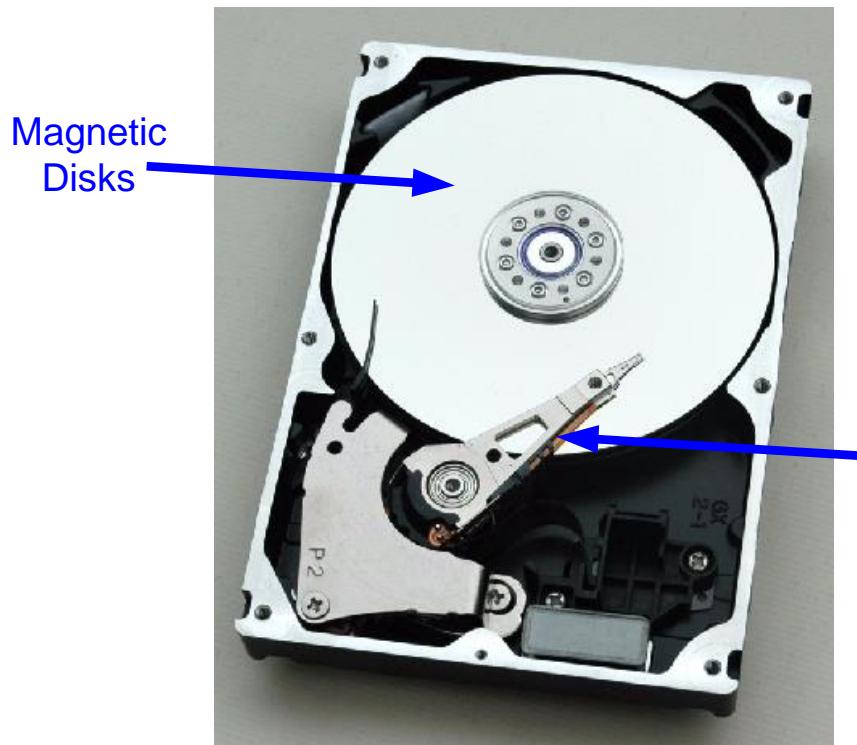
Memory Hierarchy



- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
 - Slow, Large, Cheap

Memory Hierarchy

Hard Disk Drive



Solid State Drive



Takes milliseconds to seek correct location on disk

Arshane88 / CC BY-SA 4.0 / Wikimedia Commons

Virtual Memory

- **Virtual addresses**
 - Programs use virtual addresses
 - Entire virtual address space stored on a hard drive
 - Subset of virtual address data in DRAM
 - CPU translates virtual addresses into ***physical addresses*** (DRAM addresses)
 - Data not in DRAM fetched from hard drive

Virtual Memory

- **Virtual addresses**
 - Programs use virtual addresses
 - Entire virtual address space stored on a hard drive
 - Subset of virtual address data in DRAM
 - CPU translates virtual addresses into ***physical addresses*** (DRAM addresses)
 - Data not in DRAM fetched from hard drive
- **Memory Protection**
 - Each program has own virtual to physical mapping
 - Two programs can use same virtual address for different data
 - Programs don't need to be aware others are running
 - One program (or virus) can't corrupt memory used by another

Virtual Memory

Analogous terms

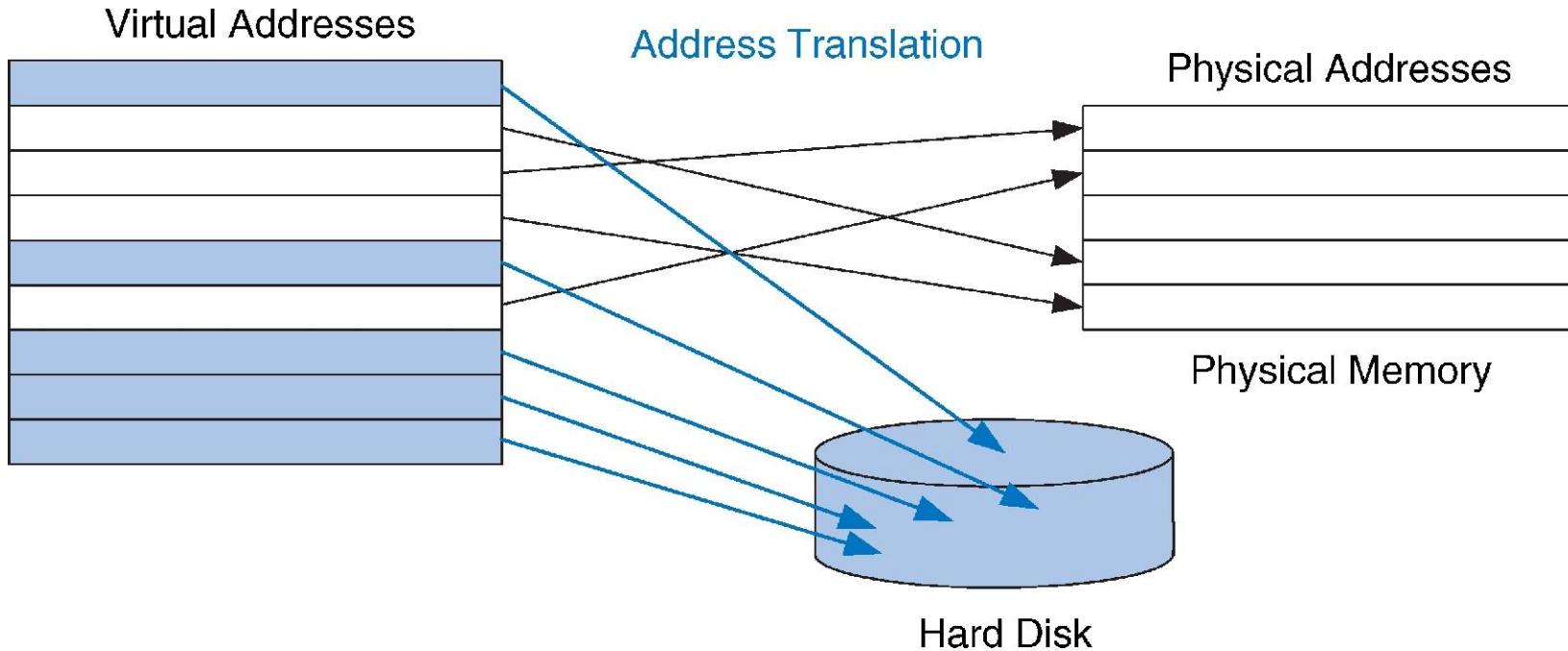
Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

Physical memory acts as cache for virtual memory

Virtual Memory Definitions

- **Page size:** amount of memory transferred from hard disk to DRAM at once
- **Address translation:** determining physical address from virtual address
- **Page table:** lookup table used to translate virtual addresses to physical addresses

Virtual Memory Definitions



© 2007 Elsevier, Inc. All rights reserved

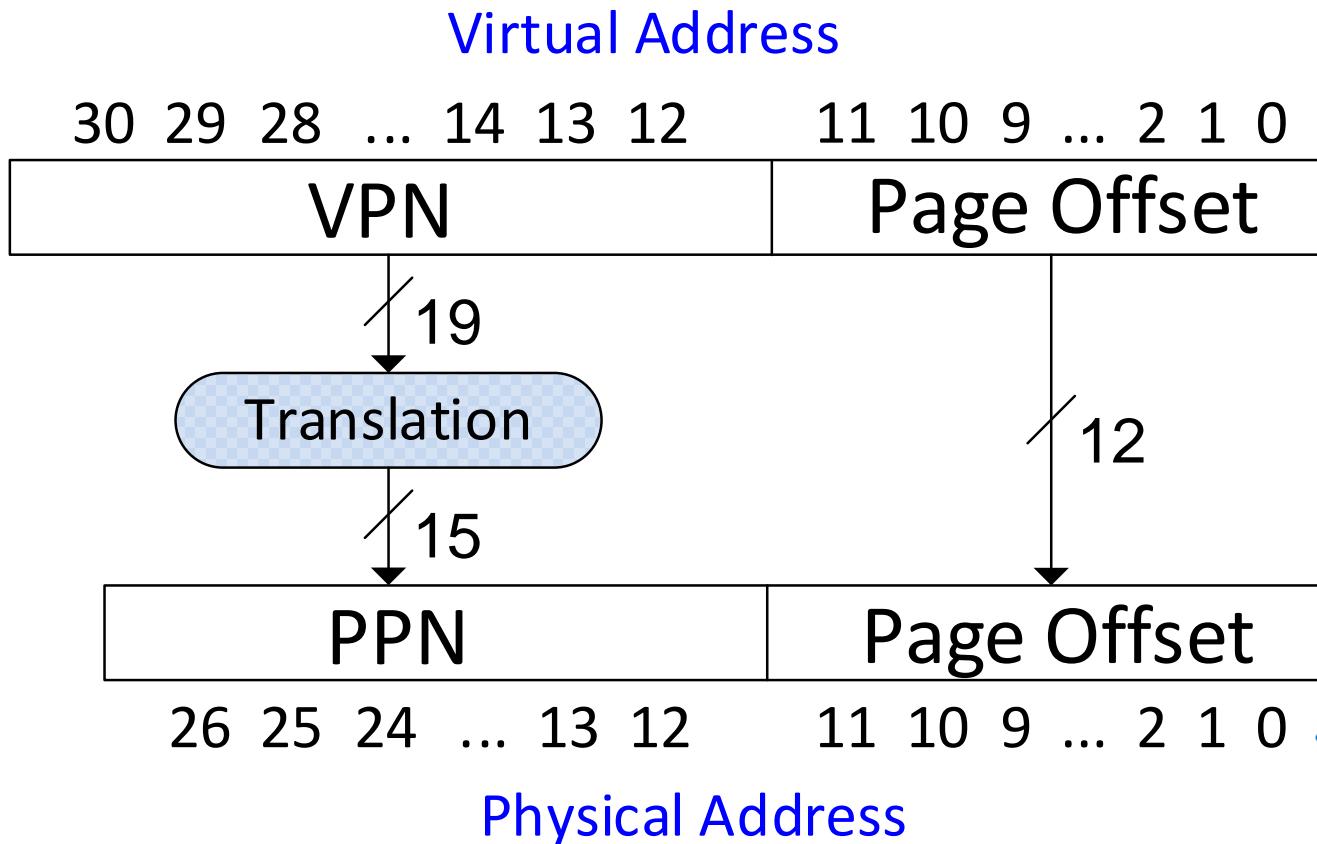
Most accesses hit in physical memory

But programs have the **large capacity** of virtual memory

Chapter 7: Microarchitecture

Address Translation

Address Translation



- **System:**
 - Virtual memory size: 2^{31} bytes
 - Physical memory size: 128 MB = 2^{27} bytes
 - Page size: 4 KB = 2^{12} bytes
- **Organization:**
 - Virtual address: **31** bits
 - Physical address: **27** bits
 - Page offset: **12** bits
 - # Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = **19** bits)
 - # Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = **15** bits)

Virtual Memory Example

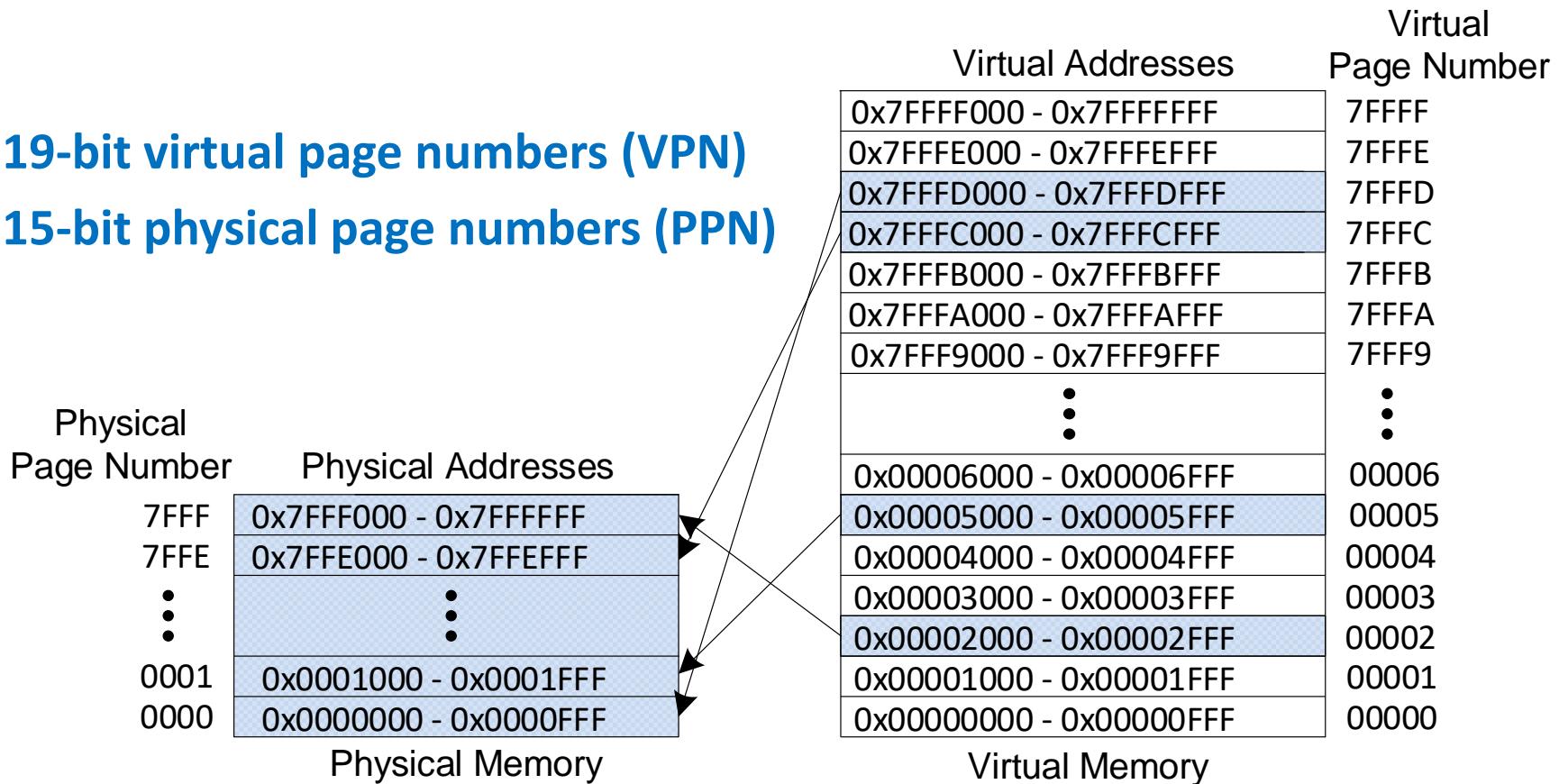
- **System:**
 - Virtual memory size: 2 GB = 2^{31} bytes
 - Physical memory size: 128 MB = 2^{27} bytes
 - Page size: 4 KB = 2^{12} bytes

Virtual Memory Example

- **System:**
 - Virtual memory size: $2 \text{ GB} = 2^{31}$ bytes
 - Physical memory size: $128 \text{ MB} = 2^{27}$ bytes
 - Page size: $4 \text{ KB} = 2^{12}$ bytes
- **Organization:**
 - Virtual address: **31** bits
 - Physical address: **27** bits
 - Page offset: **12** bits
 - # Virtual pages = $2^{31}/2^{12} = \mathbf{2^{19}}$ (VPN = **19** bits)
 - # Physical pages = $2^{27}/2^{12} = \mathbf{2^{15}}$ (PPN = **15** bits)

Virtual Memory Example

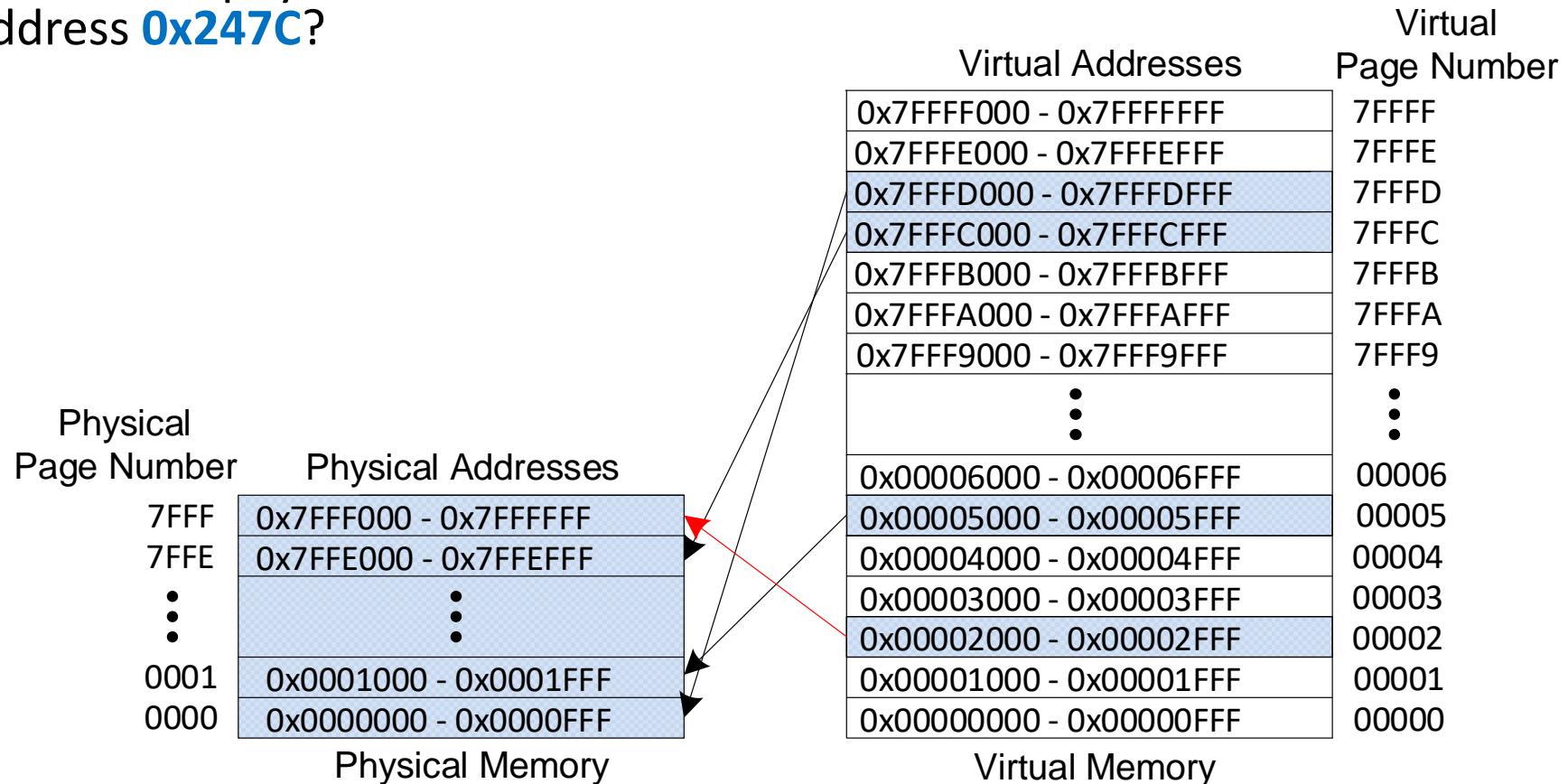
- **19-bit virtual page numbers (VPN)**
- **15-bit physical page numbers (PPN)**



Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

Virtual Memory Example

What is the physical address of virtual address **0x247C**?



Virtual Memory Example

What is the physical address of virtual address **0x247C**?

- VPN = **0x2**
- VPN 0x2 maps to PPN **0x7FFF**
- 12-bit page offset: **0x47C**
- Physical address = **0x7FFF47C**

Physical Page Number	Physical Addresses
7FFF	0x7FFF000 - 0x7FFFFFFF
7FFE	0x7FFE000 - 0x7FFEFFFF
:	:
0001	0x0001000 - 0x0001FFFF
0000	0x0000000 - 0x00000FFF

Physical Memory

Virtual Addresses	Virtual Page Number
0x7FFFF000 - 0x7FFFFFFF	7FFFF
0x7FFFE000 - 0x7FFEFFFF	7FFFE
0x7FFFD000 - 0x7FFFDFFF	7FFFD
0x7FFFC000 - 0x7FFFCFFF	7FFFC
0x7FFFB000 - 0x7FFFBFFF	7FFFB
0x7FFFA000 - 0x7FFFAFFF	7FFFA
0x7FFF9000 - 0x7FFF9FFF	7FFF9
⋮	⋮
0x00006000 - 0x00006FFF	00006
0x00005000 - 0x00005FFF	00005
0x00004000 - 0x00004FFF	00004
0x00003000 - 0x00003FFF	00003
0x00002000 - 0x00002FFF	00002
0x00001000 - 0x00001FFF	00001
0x00000000 - 0x00000FFF	00000

Virtual Memory

Chapter 7: Microarchitecture

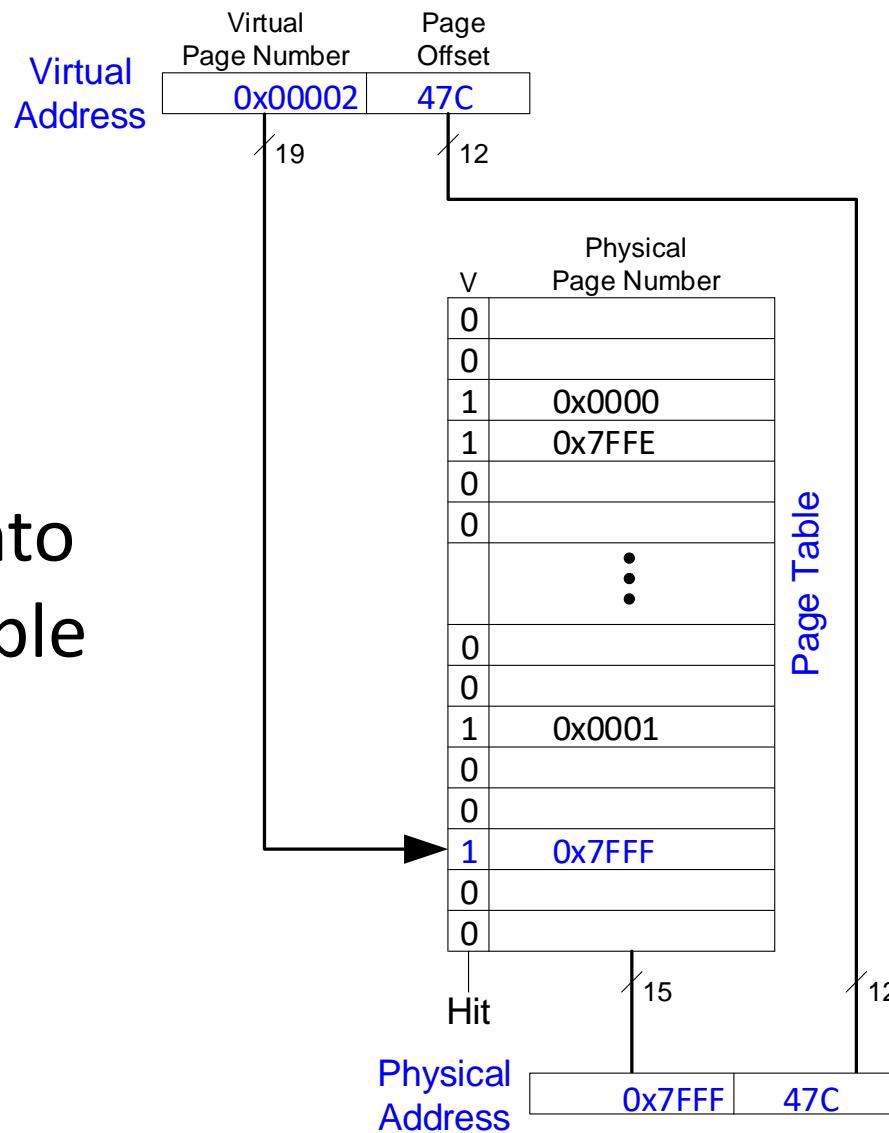
Page Table

How to Perform Translation

- **Page table**
 - Entry for each virtual page
 - Entry fields:
 - **Valid bit:** 1 if page in physical memory
 - **Physical page number:** where the page is located

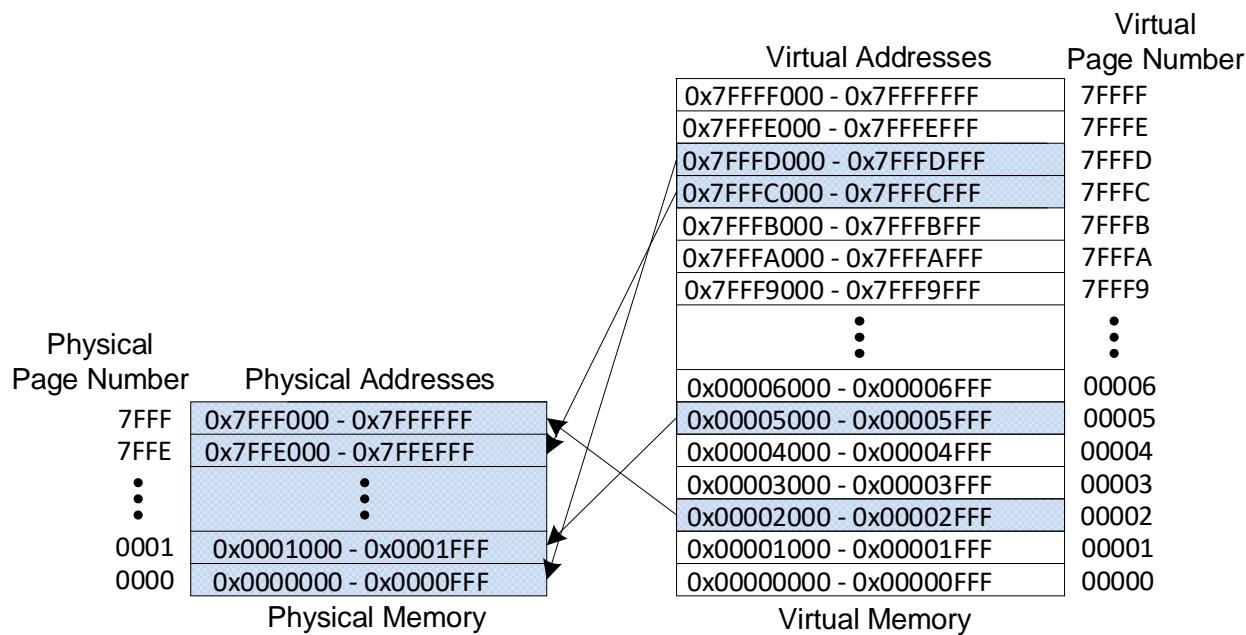
Page Table Example

VPN is
index into
page table



Page Table Example 1

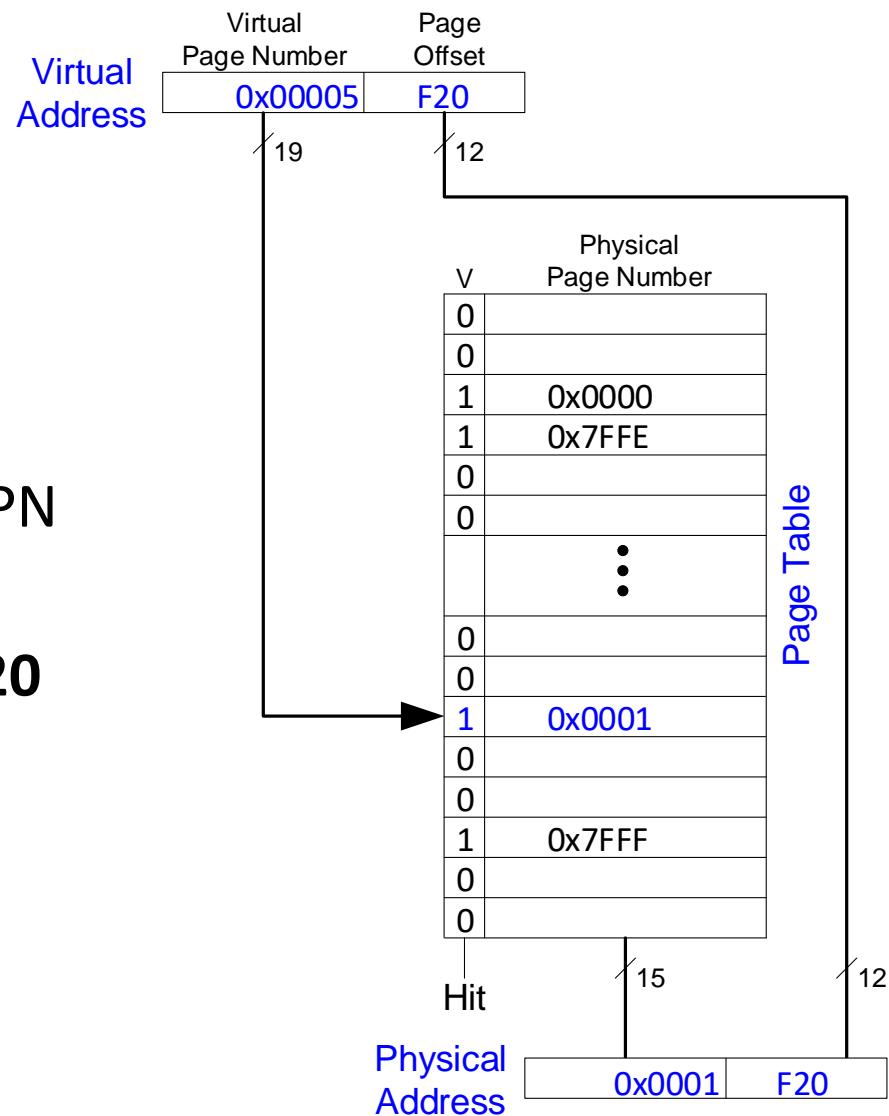
What is the physical address of virtual address **0x5F20**?



Page Table Example 1

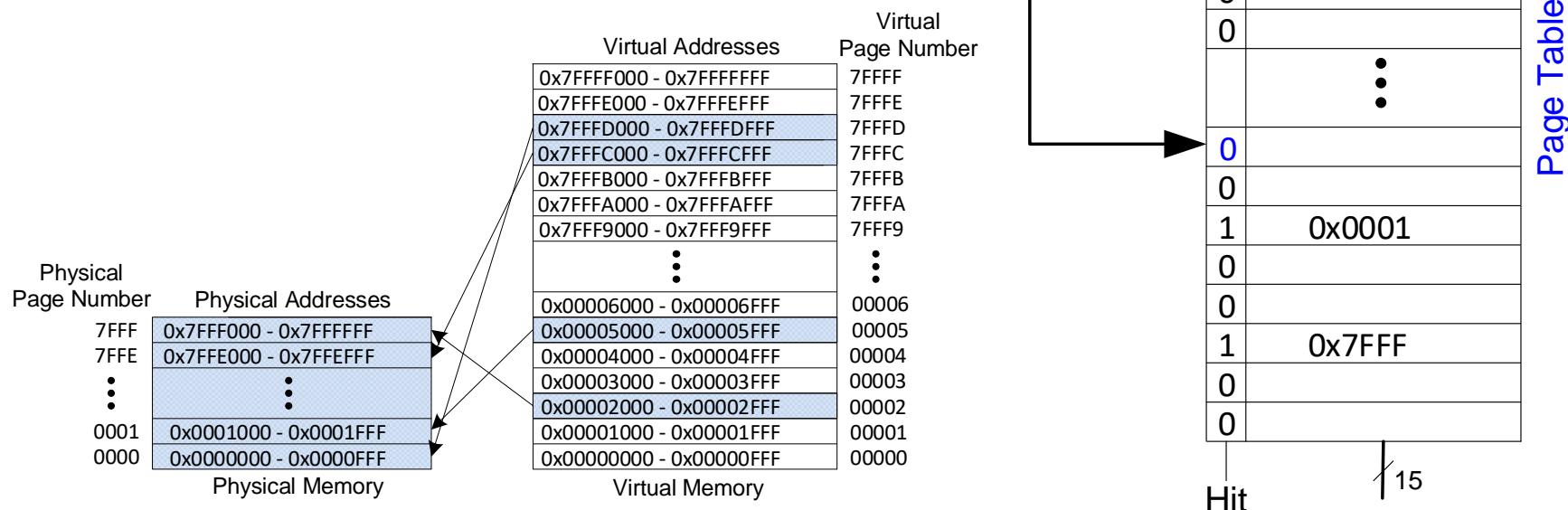
What is the physical address of virtual address **0x5F20**?

- VPN = **5**
- Entry 5 in page table VPN
5 => physical page **1**
- Physical address: **0x1F20**



Page Table Example 2

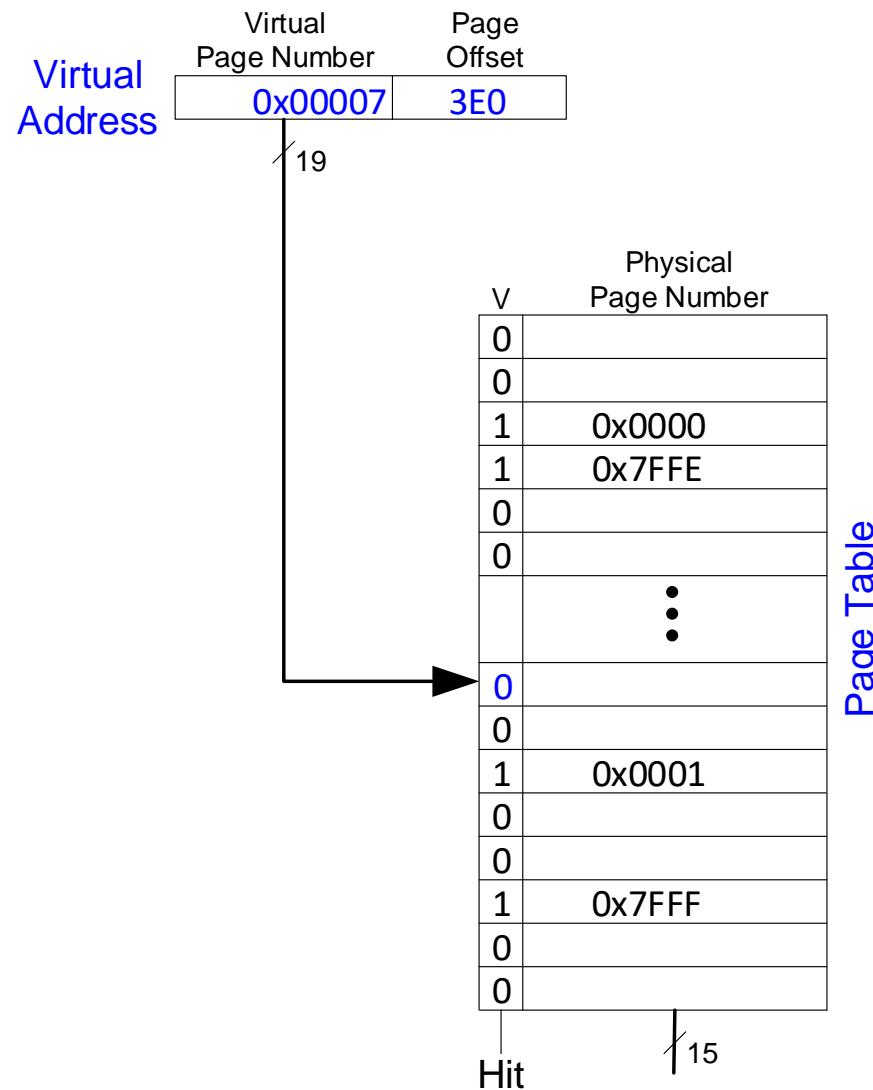
What is the physical address of virtual address **0x73E4**?



Page Table Example 2

What is the physical address of virtual address **0x73E4**?

- VPN = **7**
- Entry 7 is invalid
- Virtual page must be **paged** into physical memory from disk



Page Table Challenges

- Page table is **large**
 - usually located in physical memory
- Load/store requires **2 main memory accesses:**
 - one for translation (page table read)
 - one to access data (after translation)
- Cuts memory performance in half
 - *Unless we get clever...*

Chapter 7: Microarchitecture

Translation Lookaside Buffer (TLB)

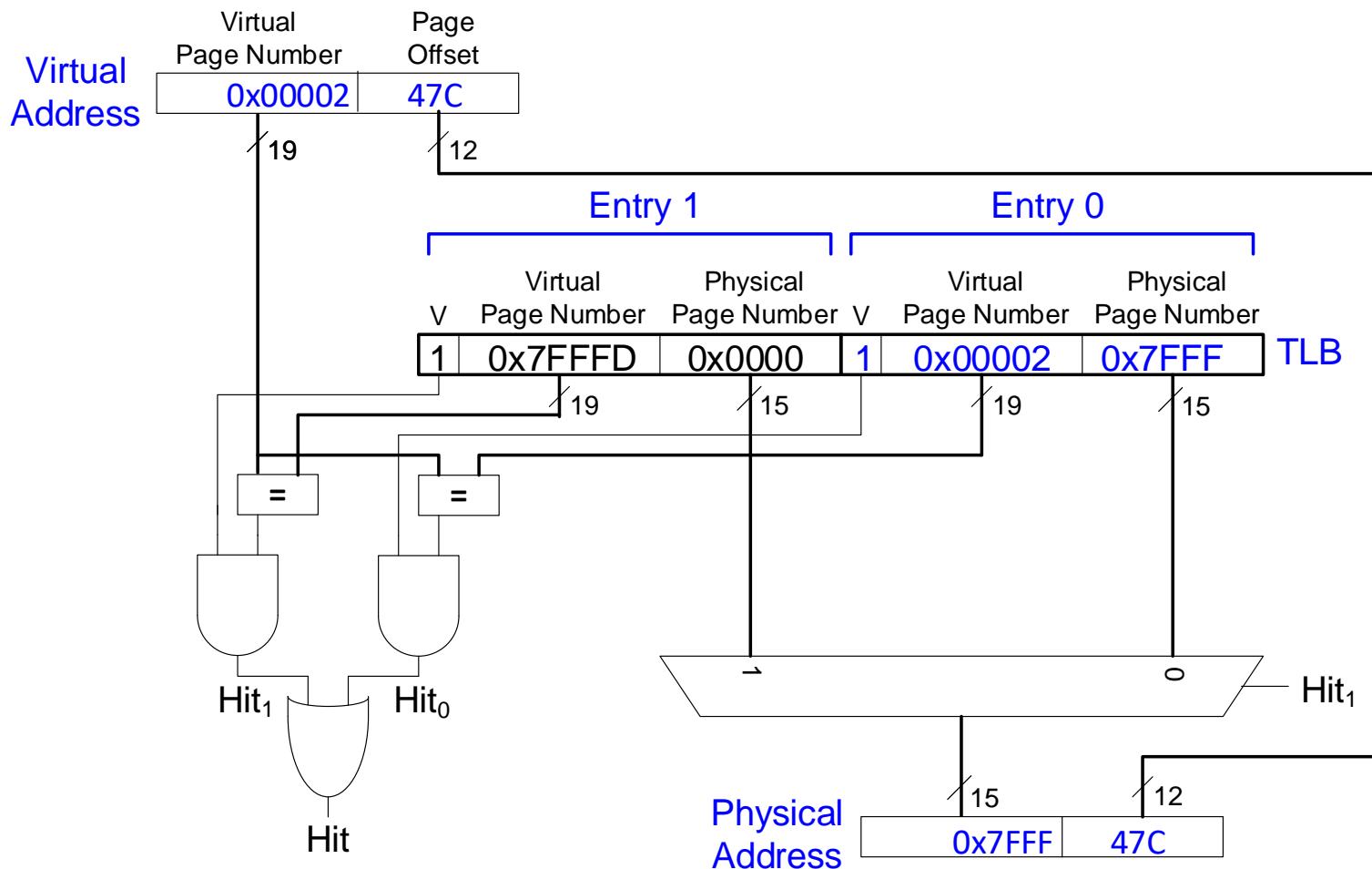
Translation Lookaside Buffer (TLB)

- Small cache of most recent translations
- Reduces number of memory accesses for *most* loads/stores from 2 to **1**

TLB

- **Page table accesses:** high temporal locality
 - Large page size, so consecutive loads/stores likely to access same page
- **TLB**
 - **Small:** accessed in < 1 cycle
 - Typically **16 - 512 entries**
 - **Fully associative**
 - > **99%** hit rates typical
 - **Reduces number of memory accesses** for most loads/stores from 2 to 1

Example: 2-entry TLB



Chapter 7: Microarchitecture

Virtual Memory Summary

Memory Protection

- **Multiple processes** (programs) run at once
- Each process has its **own page table**
- Each process can use **entire virtual address space**
- A process can only access a **subset of physical pages**: those mapped in its own page table

Virtual Memory Summary

- Virtual memory increases **capacity**
- A subset of virtual pages in physical memory
- **Page table** maps virtual pages to physical pages – address translation
- A **TLB** speeds up address translation
- Different page tables for different programs provides **memory protection**

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

**These notes may be used and modified for educational
and/or non-commercial purposes so long as the source is
attributed.**

Digital Design & Computer Architecture

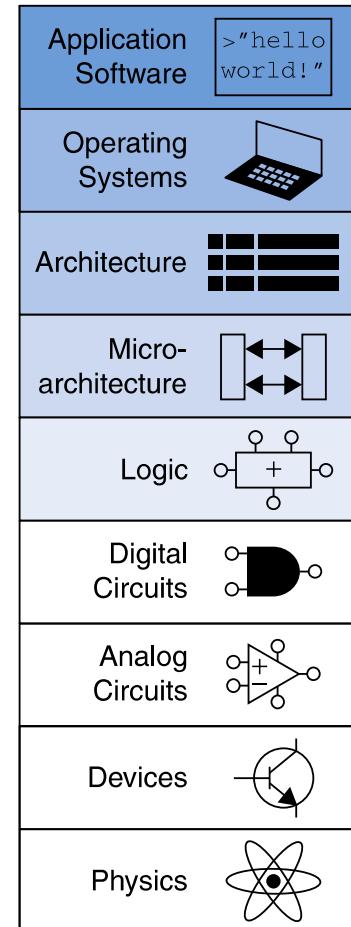
Sarah Harris & David Harris

with Joshua Brake

Chapter 9: Embedded I/O Systems

Chapter 9 :: Topics

- Microcontrollers
- RISC-V Microcontrollers
- Memory-Mapped I/O
- General-Purpose I/O
- Device Drivers
- Delays & Timers
- Example: Morse Code
- Interfacing
- Serial Peripheral Interface
- Example: SPI Accelerometer



Input/Output Systems

- Are used to connect a computer with external devices called peripherals.
- In a personal computer
 - Keyboards, monitors, printers, wireless networks, etc.
- In embedded systems
 - A toaster's heating element, a doll's speech synthesiser, an engine's fuel injector, a satellite's solar panel positioning motors, etc.
- In memory-mapped I/O, a processor can access an I/O device using the address and data buses in the same way as it accesses the main memory

Embedded Systems

- Microcontrollers are commonly used in embedded systems
- An embedded system is a system whose user may be unaware there is a computer inside
- **Examples:**
 - Microwave oven
 - Clock/radio
 - Electronic fuel injection
 - Annoying toys with batteries for toddlers
 - Implantable glucose monitor
 - Cochlear Implants:
 - Cardiac Pacemaker

Embedded Processors

- Embedded processors are so named because they are typically embedded within a larger system (such as a toy or an automobile) and have a limited user interface.
- In contrast, processors found in PCs have interfaces such as keyboards and screens that make them accessible to program or run applications.
- But all types of processors are essentially the same—they all execute instructions.
- Only the interfaces and peripheral devices used by embedded and traditional processors differ.

Microcontroller Economics

- In 2019, about **26 billion microcontrollers sold**
 - Average price: **60 cents**
 - About **70 microcontrollers in an average new car**
- **Biggest markets:**
 - Automotive (~70 microcontrollers in an average new car)
 - Consumer electronics, industrial, medical, military
- **Highly cost-sensitive market**
 - Often < \$0.40; pennies matter
 - Cheaper than using a cable or a pushrod in a system
 - A microcontroller integrated on a larger chip can have a manufacturing cost of < \$0.001
- **Memory tends to dominate the cost**
 - Select one with no more memory than necessary
- **Classified as 8, 16, 32-bit based on internal bus**
 - 1970s vintage 8-bit processors can still be adequate
 - 16-bit have the highest revenue in 2019
 - 32-bit processors are faster and more flexible but use more code memory

Microprocessor Architectures

- The architecture is the native machine language a microprocessor understands.
- **Commercial Examples:**
 - RISC-V
 - ARM
 - PIC (Peripheral Interface Controller)
 - 8051 (Introduced by Intel in 1980, Harvard architecture, separate program and data memories)

Embedded Systems

RISC-V

Microcontrollers

RISC-V Microcontrollers

- RISC-V is an **open-standard** architecture
 - Developed at Berkeley starting in 2010
 - No licensing fees
 - Increasingly popular, especially for system-on-chip
- **SiFive Freedom E310-G002**
 - Second generation RISC-V microcontroller (2019)
 - Found on several low-cost boards
- **RISC-V processor runs at 3.6 GHz in 5nm** (Dec 2022)

SiFive Freedom E310 Microprocessor

- E31 microprocessor with **5-stage pipeline**
 - Similar to the one we discussed in class
 - RV32IMAC architecture
 - Baseline **32-bit RISC-V Integer** instructions
 - Plus **Multiply/Divide, Atomic Memory, Compressed** operations
 - 2.73 Coremark/MHz
- **Electrical Specs**
 - Up to 320 MHz
 - Built in 180 nm process
 - 1.8V core and 3.3V I/O

- **CoreMark** points are derived from running the CoreMark benchmark, where a higher score indicates better performance.
- Measuring per MHz normalises the result, making comparing processors of different clock speeds easier.
- **CoreMark** is a useful metric for comparing processors in embedded systems, as it focuses on real-world tasks and eliminates the influence of clock speed alone.
- The E31 microprocessor runs at 320 MHz and achieves 2.73 CoreMark/MHz, the total CoreMark score is $2.73 \text{ Coremark/MHz} * 320 \text{ MHz} = 873.6 \text{ Coremarks}$.

Coremark Benchmark

- **Coremark:** a benchmark designed to evaluate the performance of CPUs in embedded systems and other constrained environments.
 - It consists of a set of tasks that emulate the workload of a typical embedded application.
 - Measuring Coremark per MHz normalises the result, making comparing processors of different clock speeds easier.
 - The more Coremarks a system achieves, the better its performance in handling these specific tasks.

RV32IMAC architecture

- **RV32IMAC** is a specific variant of the RISC-V instruction set architecture.
- **RV32:** This indicates a 32-bit base integer instruction set width.
 - RISC-V supports 32-bit (RV32), 64-bit (RV64), and 128-bit (RV128).
- **I:** Stands for the "Integer", the base integer instruction set.
 - It includes essential instructions for integer arithmetic and logic operations and control flow.
- **M:** Stands for the "Multiply" and "Divide" extensions. This extension adds instructions for multiplication and division operations, often used in applications requiring efficient arithmetic operations.
- **A:** Stands for the "Atomic" extension. This extension introduces atomic memory operations, vital for concurrent programming and ensuring data consistency in multi-threaded environments.
 - Atomic operations are the operations that execute as a single unified operation. In simple terms, when an atomic operation is being executed, no other process can read through or modify the data that is currently used by the atomic operation.
- **C:** Stands for the "Compressed" extension. introducing a set of 16-bit instructions to minimise code size while preserving performance.

FE310 Memory & I/O

- **Onboard Memory**
 - 16 KB Data SRAM
 - 16 KB Instruction Cache
 - 8 KB Mask ROM & 8 KB OTP (one-time-programmable) Program Boot Memory
 - Most instructions located on external Flash
- **I/O Peripherals**
 - 19 GPIO Pins
 - Serial Ports: SPI x2, I2C, UART x2
 - PWM x3
 - Timer
 - JTAG Debugger Interface

GPIO: General-Purpose Input/Output.

SPI: Serial Peripheral Interface.

I2C: Inter-Integrated Circuit: A synchronous serial communication protocol for connecting low-speed peripherals to microcontrollers.

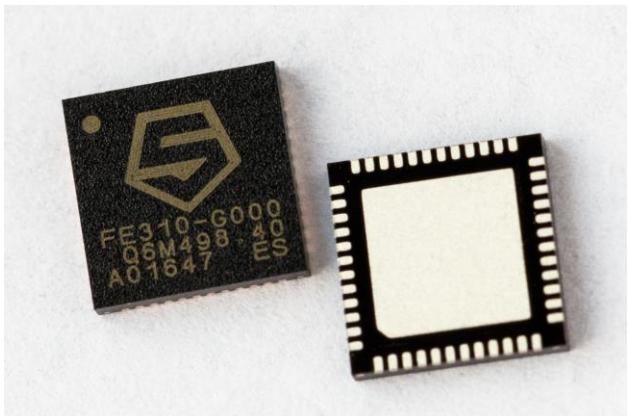
UART: Universal Asynchronous Receiver-Transmitter.

PWM: Pulse Width Modulation. A technique used to encode a signal into a pulsing waveform by varying the width of the pulses to represent the desired output.

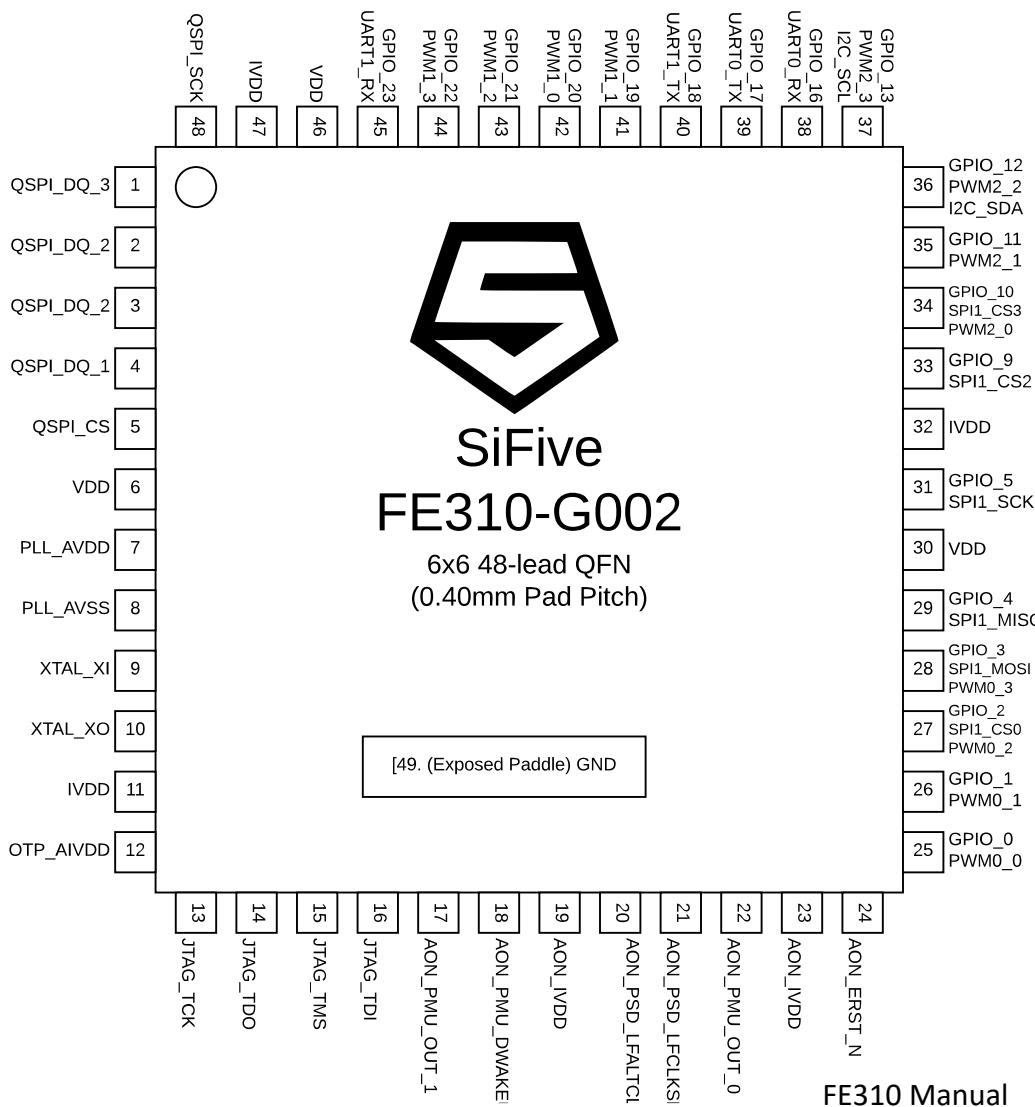
JTAG Joint Test Action Group): A standardized interface used for testing and debugging electronic circuits, often employed for boundary scan testing and in-system programming.

FE310 Pinout

- **48-pin QFN** (quad flat pack, no leads)
 - Hard to hand solder
 - **Pins:**
 - 12 power (GND pad on back)
 - 19 GPIO
 - 4 JTAG programming
 - 6 QSPI Flash code link
 - 2 clock crystals
 - 6 control



designnews.com

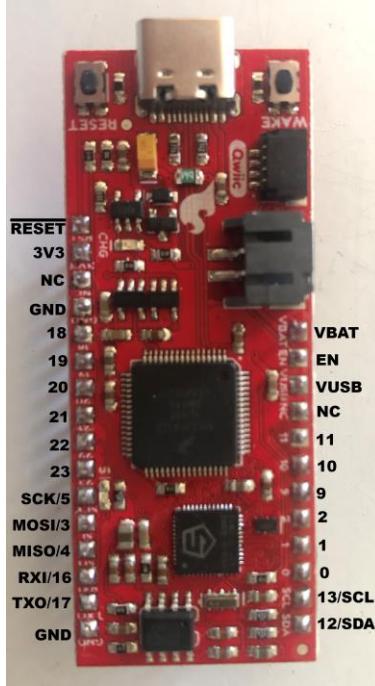


FE310 Boards

SparkFun RED-V Thing Plus

SparkFun RED-V RedBoard

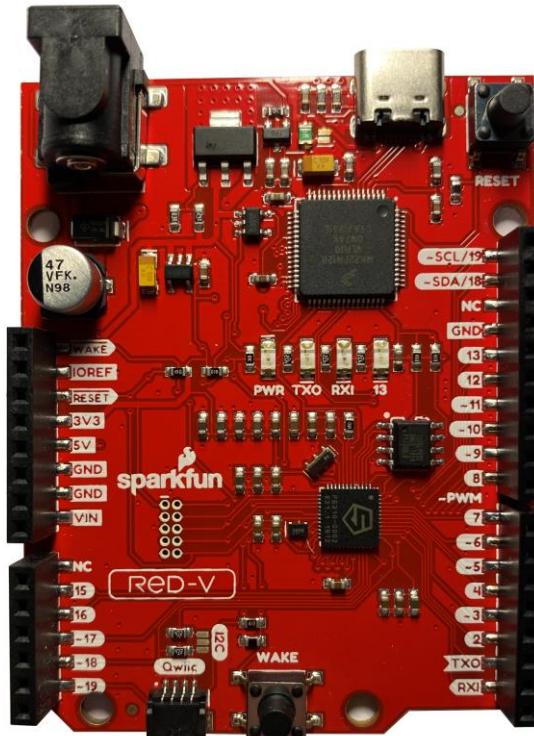
HiFive 1



\$30

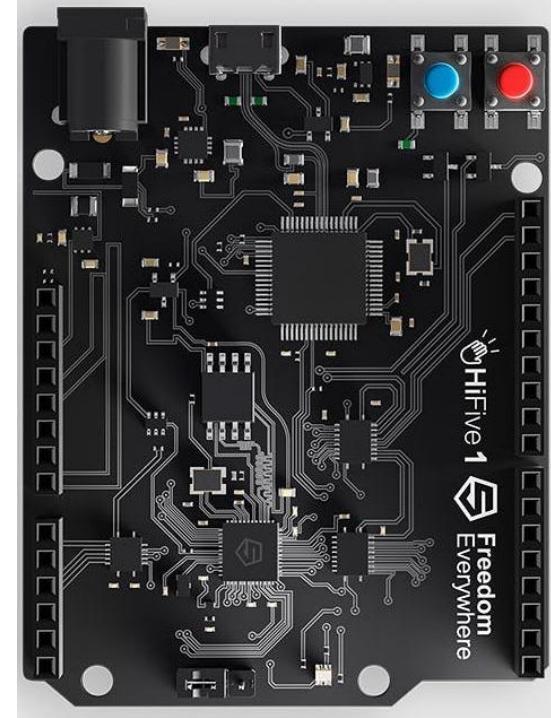
Fits breadboard (2.3x0.9")

Requires soldering
header pins



\$40

Headers soldered
Nonstandard pin labels



sifive.com

\$60

WiFi & Bluetooth

Embedded Systems

Memory-Mapped I/O

Memory-Mapped I/O

- Control peripherals by **reading or writing memory locations** called *registers*
 - Not really the same as a bank of flip-flops
- Just like accessing any other variable, but these registers cause **physical things** to happen.
- A portion of the address space is **reserved for I/O** registers rather than programs or data.
- In C, use **pointers** to specify addresses to read or write.
 - A load (read) from the specified address receives data from the peripheral device.
 - A store (write) to the specified address sends data to the peripheral device.

Example: FE310 Memory Map

- **Boot ROM:**
 - 0x00010000-1FFFF
- **Code Flash:**
 - 0x20000000-3FFFFFF
- **Data SRAM:**
 - 0x80000000-80003FFF
- **Peripherals:**
 - 0x02000000-0x1FFFFFF

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT (Core-Local INTerruptor)	
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

From FE310 Manual

Memory Mapped I/O in C

```
#include <stdint.h>

// Pointers to memory-mapped I/O registers
volatile uint32_t *GPIO_INPUT_VAL = (uint32_t*)0x10012000;
volatile uint32_t *GPIO_OUTPUT_VAL = (uint32_t*)0x1001200C;

// read all the GPIO inputs
uint32_t allInputs = *GPIO_INPUT_VAL;

// read GPIO bit 19
int gpio19 = (*GPIO_INPUT_VAL >> 19) & 0b1; // 0b1 means binary 1

// Wait for bit 19 to be 0
while ((*GPIO_INPUT_VAL >> 19) & 0b1); // 0b1 means binary 1

// Wait for bit 19 to be 1
while (!(*GPIO_INPUT_VAL >> 19) & 0b1)); // 0b1 means binary 1

// Write 1 to GPIO bit 5
*GPIO_OUTPUT_VAL |= (1<<5);

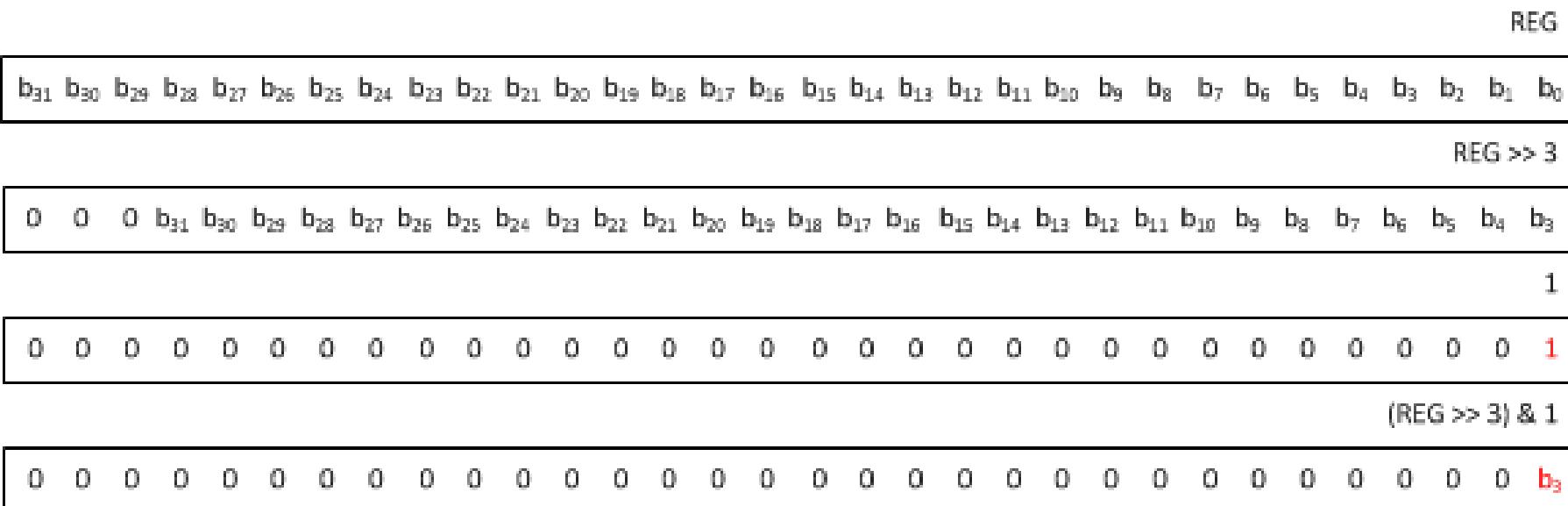
// Write 0 to GPIO bit 5
*GPIO_OUTPUT_VAL &= ~(1<<5);
```

The **prefix 0b** is a standard notation in C, C++, Python, to indicate that the number following it is in binary format.

0b1 means binary number 1.

C Idioms: Read value of bit

```
int bit = (*REG >> 3) & 1; // get value of bit 3
```



C Idioms: Write Bit to 1

```
*REG |= (1 << 3); // turn on bit 3
```

1

1583

REG

b₃₁ **b₃₀** **b₂₉** **b₂₈** **b₂₇** **b₂₆** **b₂₅** **b₂₄** **b₂₃** **b₂₂** **b₂₁** **b₂₀** **b₁₉** **b₁₈** **b₁₇** **b₁₆** **b₁₅** **b₁₄** **b₁₃** **b₁₂** **b₁₁** **b₁₀** **b₉** **b₈** **b₇** **b₆** **b₅** **b₄** **b₃** **b₂** **b₁** **b₀**

REG | 1<<3

b₃₁ **b₃₀** **b₂₉** **b₂₈** **b₂₇** **b₂₆** **b₂₅** **b₂₄** **b₂₃** **b₂₂** **b₂₁** **b₂₀** **b₁₉** **b₁₈** **b₁₇** **b₁₆** **b₁₅** **b₁₄** **b₁₃** **b₁₂** **b₁₁** **b₁₀** **b₉** **b₈** **b₇** **b₆** **b₅** **b₄** **1** **b₂** **b₁** **b₀**

C Idioms: Write Bit to 0

`*REG &= ~(1 << 3); // turn off bit 3`

1

1

1 < 3

n(1 << 3)

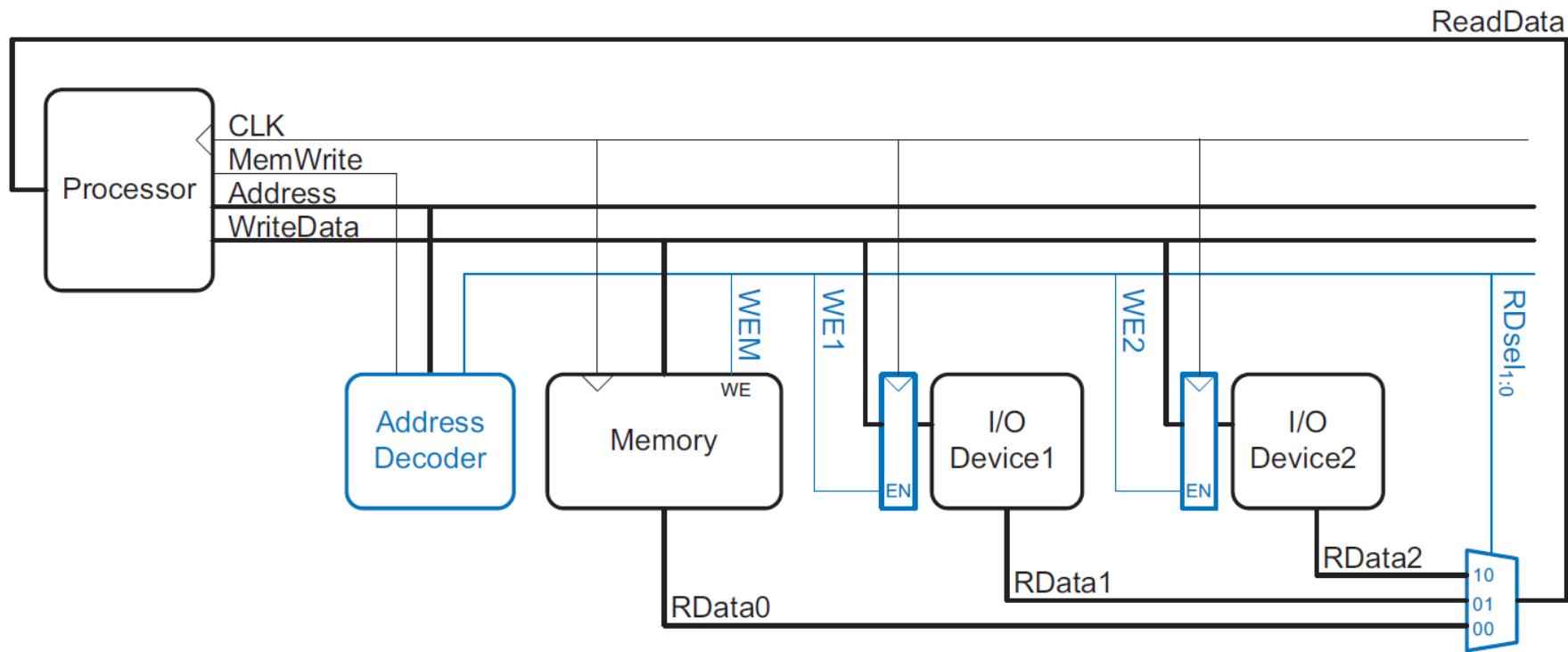
REG

For more information about the study, please contact Dr. John D. Cawley at (609) 258-4626 or via email at jdcawley@princeton.edu.

REG & γ [1 << m]

Example e9.1 Communicating With I/O Devices

Suppose that I/O Device 1 in [Figure e9.1](#) is assigned the memory address 0x20001000. Show the RISC-V assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.



[Figure e9.1](#) Support hardware for memory-mapped I/O

Example e9.1 Communicating With I/O Devices

Solution The following RISC-V assembly code writes the value 7 to I/O Device 1. The .equ assembler directive replaces the named symbol with the given value. So, the li s1, ioadr instruction becomes li s1, 0x20001000.

```
.equ ioadr 0x20001000

    li s0, 7          # Load immediate value 7 to register s0
    li s1, ioadr     # Load immediate value 0x20001000 to register s1
    sw s0, 0(s1)      # Store word in s0 to memory address [s1+0]
```

The address decoder detects address 0x20001000 and *MemWrite* = 1, so it asserts WE1, the write enable for Device 1's register. At the next clock edge, the value on the *WriteData* bus, 7, is written into the register, whose output connects to the input pins of I/O Device 1.

Example e9.1 Communicating With I/O Devices

Read from Device 1

To read from I/O Device 1, the processor executes the following RISC-V assembly code.

```
lw s0, 0(s1) # loads word at memory address [s1+0] to register s0
```

The address decoder detects the address 0x20001000, so it sets $RDsel_{1:0}$ to 01. The multiplexer thus selects $RData1$, the read data from Device 1, and connects it to the *ReadData* bus, the value of which is then loaded into s0 in the processor.

The addresses associated with I/O devices are often called *I/O registers* because they may correspond with physical registers in the I/O device like those shown in [Figure e9.1](#).

Device Driver

- Software that communicates with an I/O device is called a **device driver**.
- Writing a device driver requires detailed knowledge about the I/O device hardware, including the addresses and behaviour of the memory-mapped I/O registers.
- Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.
- You have probably downloaded or installed device drivers for your printer or other I/O device.

Embedded Systems

General Purpose I/O (GPIO)

General-Purpose I/O (GPIO)

- GPIOs can be written (driven to 0 or 1) or read.
- **Examples:** LEDs, switches, connections to other digital logic
- **In general:**
 - Look up how many pins your microcontroller has, how they are named.
 - Each pin needs to be configured as an input, output, or special function.
 - Then read or write the pin.

Embedded Systems

Interfacing

Interfacing

- **Interfacing:** connecting external devices to a microcontroller
 - Sensors
 - Actuators
 - Other Processors
- **Interfacing Methods**
 - Parallel
 - Serial
 - **SPI:** Serial Peripheral Interface
 - 1 clock, 1 data out, 1 data in pin
 - **UART:** Universal Asynchronous Receiver/Transmitter
 - no clock, 1 data out, 1 data in pin, agree in software about intended data rate
 - **I2C:** Inter-Integrated Circuit
 - 1 clock, 1 bidirectional data pin
 - Analog
 - **ADC:** Analog/Digital Converter
 - **DAC:** Digital/Analog Converter
 - **PWM:** Pulse Width Modulation

Parallel Interfacing

- Connect **1 wire / bit** of information
 - **Example:** The PATA (Parallel Advanced Technology Attachment) interface protocol typically transfers 16 bits at a time.
- Also send **clock** or **REQ/ACK** (request/acknowledge) to indicate when data is ready
- Parallel busses are **expensive** and **cumbersome** because of the large number of wires
- Mostly used for **high-performance applications** such as DRAM interfaces

Serial Interfacing

- Serial interface sends **one bit at a time**
 - Use many clock cycles to send a large block of information
 - Also send timing information about when the bits are valid
- **Common Serial Interfaces:**
 - Serial Peripheral Interface (**SPI**)
 - Serial clock, 2 unidirectional data wires (MOSI, MISO)
 - Very common, easy to use
 - Inter-Integrated Circuit (**I²C**) “I squared C”
 - 1 clock, 1 bidirectional data wire
 - Fewer wires, more complex internal hardware
 - Universal Asynchronous Receiver/Transmitter (**UART**) “you-art”
 - 2 unidirectional data wires (Tx, Rx)
 - Asynchronous (no clock); configure (agreed upon) speed at each end

Embedded Systems

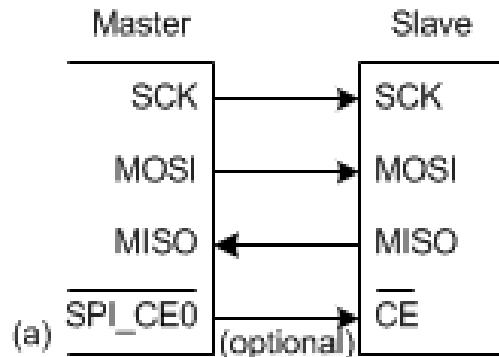
SPI

Serial Peripheral Interface (SPI)

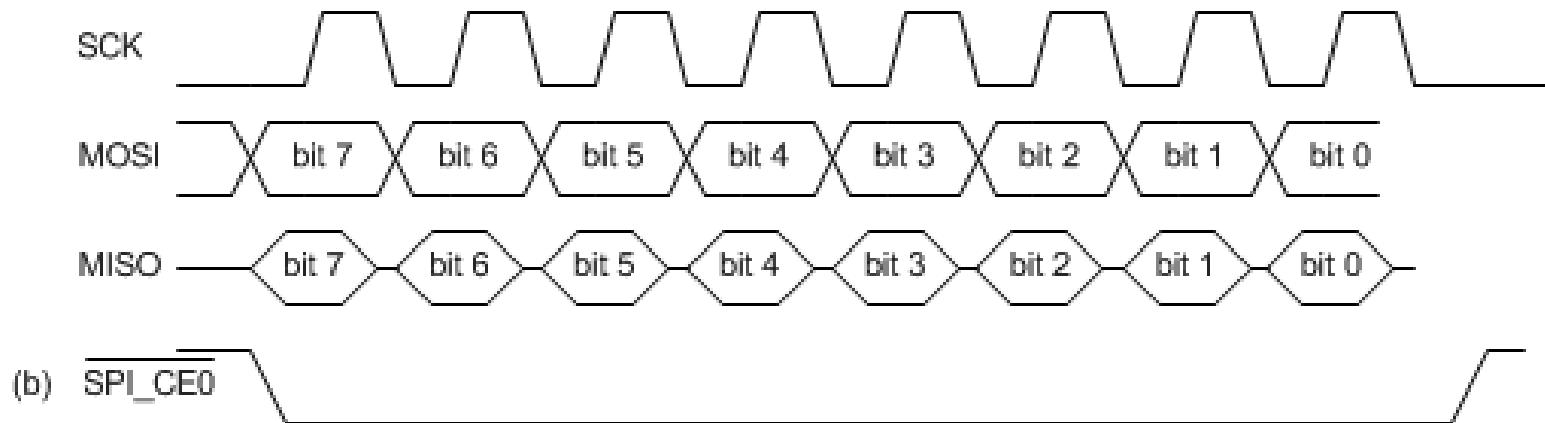
- SPI is an easy way to connect devices
- **Master device** communicates with one or more **slave devices**
 - Slave select signals indicate which slave is chosen
 - Master sends clock and **data out**. Slave sends back **data in**.
- **Signals:**
 - **SCK/SCLK**: (Serial Clock) Generated by master; one pulse per bit
 - **MOSI**: (Master Out Slave In) serial data from master to slave
 - **MISO**: (Master In Slave Out) serial data from slave to master
 - **SS/CE/CS**: (Slave select/Chip Enable/Chip Select) optional, one or more, may be active low
- SPI allows full-duplex communication, meaning MOSI and MISO operate simultaneously:
 - While the master sends data on MOSI, the slave responds with data on MISO at the same time.
 - This synchronised data exchange is controlled by the clock signal (SCLK) provided by the master.

SPI Waveforms

- SPI Slave hardware can be just a **shift register**

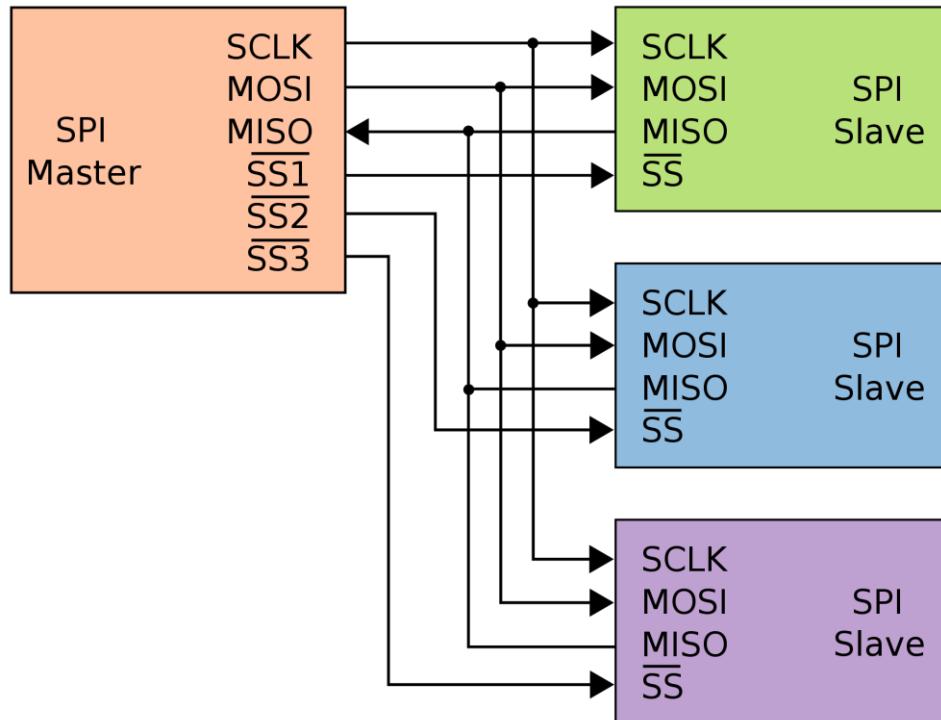


SCK: Generated by master; one pulse per bit
MOSI: (Master Out Slave In) serial data from master to slave
MISO: (Master In Slave Out) serial data from slave to master
CE: (Chip Enable) optional, one or more, may be active low

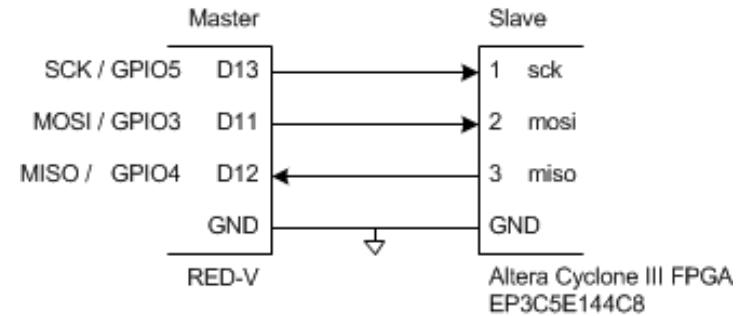


SPI Connection

Generic Master to Several Slaves



RED-V Master to One Slave



en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#/media/File:SPI_three_slaves.svg

PCI (Peripheral Component Interconnect)

- PCI is a parallel interface standard developed in the 1990s for connecting peripherals to a computer's motherboard, such as network cards, sound cards, and storage controllers.
 - **PCI**: A legacy **parallel** interface with shared bandwidth, now largely obsolete, replaced by **PCIe** (PCI express).
- Key Features:
 - Bus Width: 32-bit or 64-bit parallel bus.
 - Speed: Transfer speeds up to 533 MB/s (depending on version and width).
 - Shared Bus Architecture: All devices on the PCI bus share the same communication channel, which can lead to contention and limited scalability.
 - Plug-and-Play: Devices are automatically configured by the system's BIOS or operating system.
- Limitations:
 - Limited bandwidth due to the shared bus and parallel communication.
 - Signal integrity issues over longer distances.

PCIe (PCI Express)

- PCIe is the successor to PCI, designed as a high-speed serial interface to overcome the limitations of the older parallel architecture.
- Key Features:
 - Point-to-Point Architecture: Each device has a dedicated communication channel with the system (no shared bus).
 - Serial Communication: Data is transmitted in serial across lanes, which consist of one pair for transmitting and one for receiving.
 - Scalability: Supports multiple lanes (x1, x4, x8, x16, x32), enabling higher bandwidth.
 - Example: PCIe 4.0 x16 offers up to 32 GB/s of bidirectional bandwidth.
 - Compatibility: Backward-compatible with earlier PCIe versions.

PCIe (PCI Express) Versions

- PCIe 3.0: 8 GT/s per lane (~1 GB/s per lane in each direction).
- PCIe 4.0: 16 GT/s per lane (~2 GB/s per lane in each direction).
- PCIe 5.0: 32 GT/s per lane (~4 GB/s per lane in each direction).
- PCIe 6.0 (emerging): 64 GT/s per lane.

• A **lane** is a bidirectional communication channel consisting of one transmit pair and one receive pair of differential wires.
• "**32 GT/s per lane**" means each lane can perform 32 billion transfers per second, and the total bandwidth scales with the number of lanes (e.g., x4, x16).

PCIe (PCI Express) Bandwidth

PCIe Version	Transfer Rate (GT/s per lane)	Encoding Scheme	Effective Bandwidth (per lane)
PCIe 3.0	8 GT/s	8b/10b encoding	7.88 Gbps
PCIe 4.0	16 GT/s	128b/130b encoding	15.75 Gbps
PCIe 5.0	32 GT/s	128b/130b encoding	31.5 Gbps
PCIe 6.0	64 GT/s	PAM4 signaling (no encoding overhead)	64 Gbps

PAM4 Signaling: PCIe 6.0 uses Pulse Amplitude Modulation with 4 levels (PAM4) instead of the traditional NRZ (Non-Return to Zero) signalling used in earlier versions. This allows PCIe 6.0 to double the data rate while keeping the same base frequency.

OpenAI. (2024). *ChatGPT* [Large language model]. <https://chatgpt.com>

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

**These notes may be used and modified for educational
and/or non-commercial purposes so long as the source is
attributed.**

A Brief Introduction to Xilinx FPGA



© 2011-2014 Brendan Cronin
School of Electronic Engineering
Dublin City University
Dublin 9, Ireland

ASICs vs FPGAs

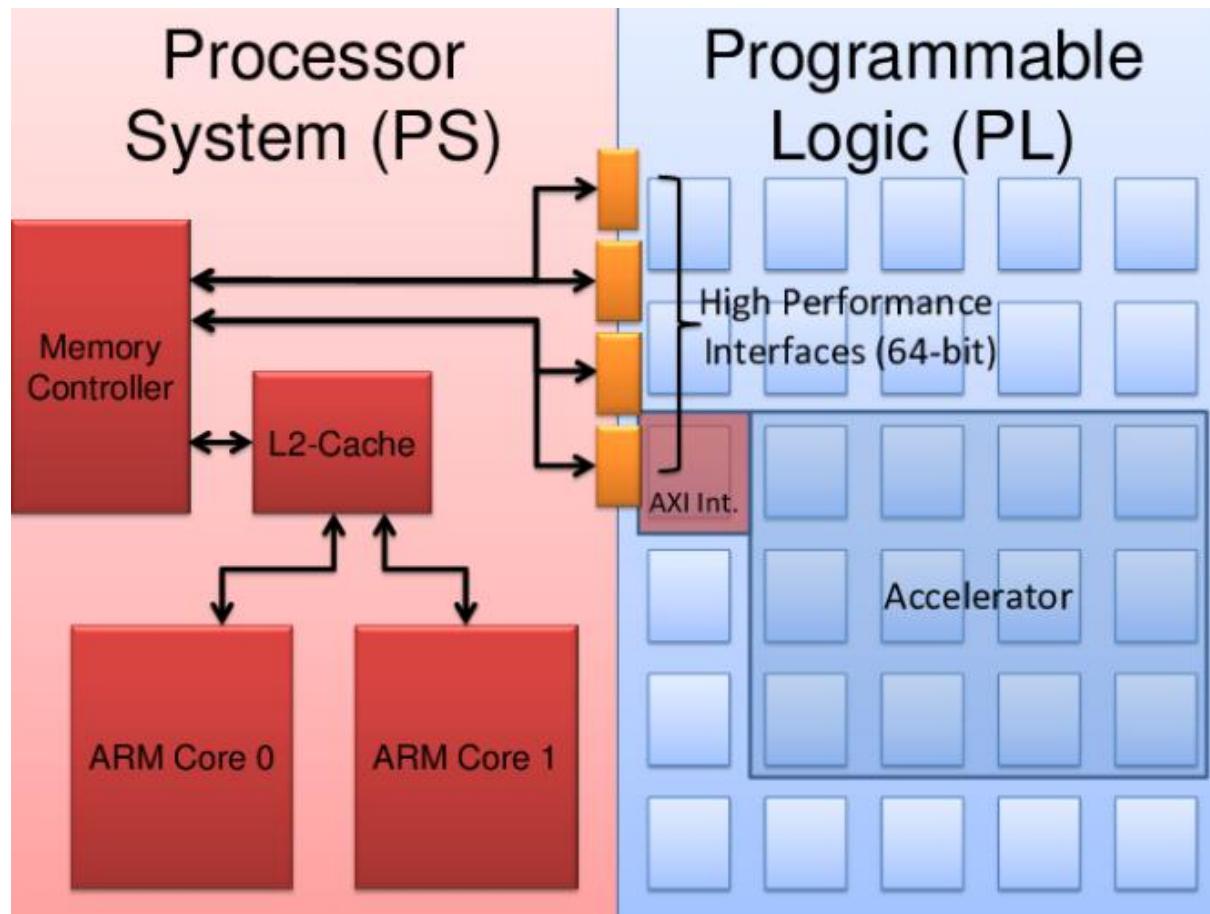
- ❑ ASIC – Application Specific Integrated Circuits
 - ❑ purpose-built and mass produced for a specific function
 - ❑ cannot be reprogrammed, and require a significant NRE (Non-Recurring Engineering) investment
 - ❑ NRE cost – one-time cost to research, design, develop and test a new product or product enhancement
- ❑ FPGA – Field Programmable Gate Arrays
 - ❑ Integrated circuits with a programmable hardware fabric
 - ❑ The circuitry inside an FPGA can be reprogrammed to implement a wide variety of functions
 - ❑ Prefabricated hardware fabric can be reprogrammed by customers in their labs or in the field
 - ❑ Do not require-NRE investment
 - ❑ Help innovators get to the market extremely fast
 - ❑ Great option for differentiation in a quickly changing environment

FPGA (Field Programmable Gate Array)

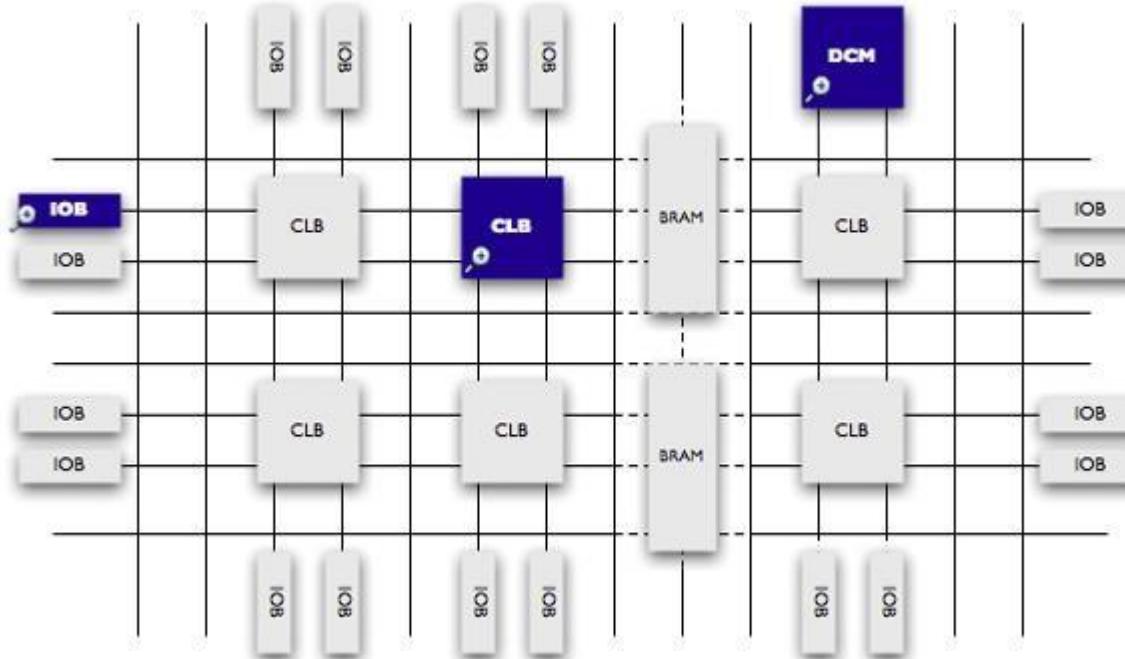
- ❑ An FPGA is a programmable semiconductor device
- ❑ It contains a matrix of Configurable Logic Blocks (CLBs) connected via programmable interconnects
- ❑ There are a number of types
 - **SRAM-based FPGAs** : Configuration data is stored in volatile SRAM. They must be programmed (configured) upon start. Most Xilinx FPGAs are of this type. Two modes of programming:-
 - *Master mode* : FPGA reads configuration data from an external source such as external flash chip.
 - *Slave mode* : FPGA is configured by an external device such as a processor.
 - **SRAM-based FPGAs with internal flash memory** : Similar to previous type, but the bitstream is stored in the internal flash, thereby eliminating the need for external non-volatile memory. e.g. Xilinx Spartan-3AN family.
 - **Flash-based FPGAs** : flash is used as the primary resource for configuration storage (i.e. no SRAM). Consume less power. Produced by Actel.
 - **Antifuse-based FPGAs** : can only be programmed once. Produced by Actel.

Xilinx Zynq-7000 SoC Block Diagram

- ❑ Zynq-7000 family is based on Xilinx SoC (System-on-Chip) architecture
- ❑ Dual-core ARM Cortex-A9 based processing system (PS) and 28nm Programmable logic (PL)
- ❑ Communication between PS and PL through high-performance AXI interface
- ❑ AXI – The Advanced eXtensible Interface between blocks of IP
- ❑ MPSoC – Multi-Processor SoC, contains two or more CPU cores



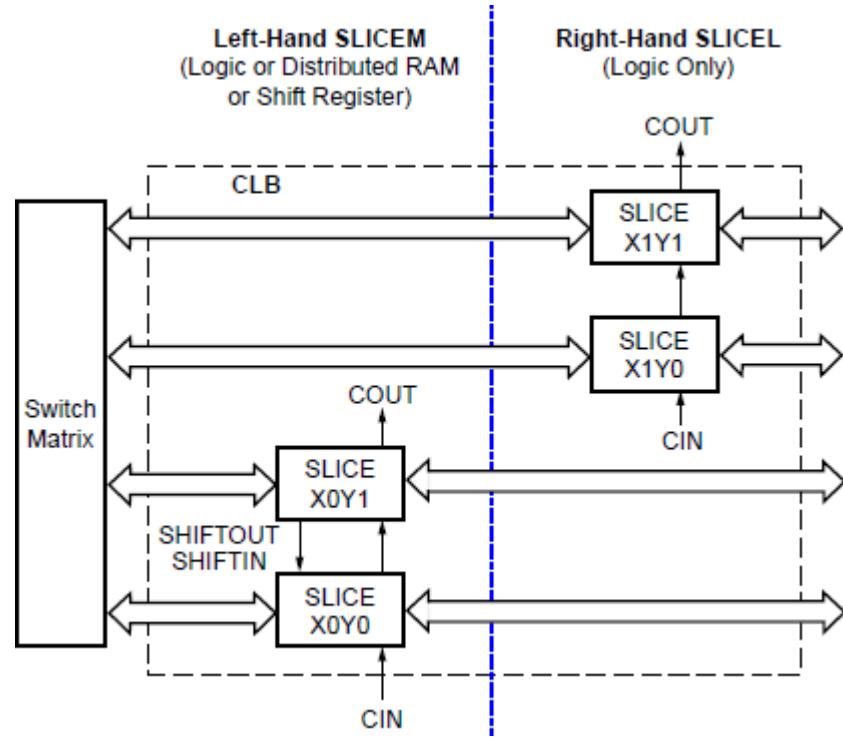
Structure of SRAM-based FPGAs



- ❑ Input/Output Block (IOB): implements the input and output functions of the device.
- ❑ Configurable Logic Block (CLB) : basic logic unit in FPGA
- ❑ Digital Clock Management (DCM) : multiplication/division of clock frequency, phase shifting of clock, clock skew elimination, etc.
- ❑ Block RAM (BRAM): Dedicated on-chip memory

Structure of Xilinx Spartan3 CLB

- ❑ CLBs are organised in an interconnected grid. CLB structure depends on device. This slide shows the Spartan3 CLB.
- ❑ Each CLB contains 4 slices, 2 of type SLICEM (M=Memory) and 2 of type SLICEL (L=Logic)
- ❑ Certain Xilinx FPGA families also contain SLICE_X slices which are equivalent to SLICEL without carry logic.
- ❑ Each Spartan3 SLICEL slice contains two 4-input LUTs (Look Up Tables), two flip-flops, two muxes, carry & arithmetic logic. The LUT implements combinational logic.
- ❑ SLICEM slice is like a SLICEL slice but the LUT may alternatively be used as a 16-bit RAM or a 16-bit shift register.



Block RAM

- ❑ A block RAM is a dedicated two port memory containing several kilobits of RAM. It cannot be used to implement any other function.
- ❑ The size and number of block RAMs depends on the particular FPGA device, e.g. BRAM on the Spartan 3A family of devices has a size of 18Kbits. The more advanced the FPGA, the greater the number of BRAMs.

Xilinx Distributed RAM

- ❑ SLICEM slices can be configured to implement distributed memory instead of combinational logic. Each LUT (Look-Up Table) in the SLICEM can be configured as a 16x1bit RAM, ROM, LUT or 16bit shift register.
- ❑ Distributed RAM is RAM which is “*distributed*” over multiple LUTs configured as 16x1bit RAMs.
- ❑ For example, CLBs on the Spartan3 FPGA consist of four slices, two of which are SLICEM. Each SLICEM contains two LUTs, giving a total of four 16-bit LUTs available for distributed RAM per CLB. Each Spartan3 CLB can therefore support 64x1bit RAMs or 32x2bit RAMs.
- ❑ The synthesis tool implements larger and wider memory by connecting several distributed RAMs in parallel.

Block RAM versus Distributed RAM

- ❑ Besides the implementation differences outlined on the previous two slides (i.e. BRAM is dedicated memory, whereas Distributed RAM is based on LUTs, etc.), the following are some other differences
- ❑ Distributed RAM is ideal for small sized memories. It is not suitable for larger memories because of the wiring delays due to its distributed nature.
- ❑ Block RAMs have fixed sizes (e.g. 18Kbits) and it can be quite wasteful to use them to implement a small RAM.
- ❑ BRAM should be used for large sized memories and distributed RAM for small.
- ❑ Both types have a **synchronous write operation**.
Distributed RAM has an asynchronous read, whereas
BRAM is synchronous read.

-
3. Infer the RAM from your VHDL description. The Xilinx synthesis tool, XST, automatically looks for the best implementation for each inferred RAM, which may be either BRAM or Distributed. You can force it to infer a particular type by setting the *ram_style* attribute to either *distributed* or *block*. This method has the advantage of being portable to non-Xilinx FPGAs.

e.g. to force it to use distributed RAM:

```
architecture behav of ram_entity is
--Declaration of type and signal of a 256 element RAM
--with each element being 8 bit wide.
type ram_t is array (255 downto 0) of std_logic_vector(7 downto 0);
signal ram : ram_t := (others => (others => '0'));
attribute ram_style: string;
attribute ram_style of ram : signal is "distributed";
begin
process(clk)
  ...
end process;
end behav;
```

Block RAM Operating Mode

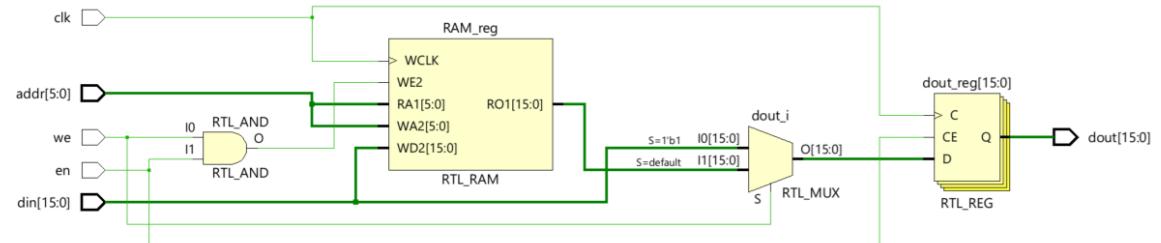
- ❑ Operating Mode determines which data is presented on the output during a write operation. Xilinx FPGA supports three different modes which can be configured per BRAM port.
- ❑ **Write First:** Input data is simultaneously written into memory and presented on the data output.
- ❑ **Read First:** Data previously stored at the write address is presented on the data output.
- ❑ **No Change:** Data output is latched during write operation. i.e. previous read data remains on the output.
- ❑ These modes can be configured using any of 3 previous BRAM creation methods, i.e CORE generator, BRAM primitives, inference from VHDL/Verilog.

Inferring Write-First single port BRAM from VHDL

```
module bram_wr_first (
    input logic clk,
    input logic we, // Write Enable
    input logic en, // Enable
    input logic [5:0] addr, // 6-bit Read/Write address
    input logic [15:0] di, // 16-bit data input
    output logic [15:0] do // 16-bit data output);

    // 64x16 bit memory
    logic [15:0] RAM [63:0];

    always_ff @ (posedge clk) begin
        if (en) begin
            if (we) begin
                RAM[addr] <= di; // Write data to address addr
                do <= di; // Read data is the newly written data
            end else begin
                do <= RAM[addr]; // Read data from address addr
            end
        end
    end
endmodule
```



Xilinx DCM functionality and how to add one to a VHDL Design

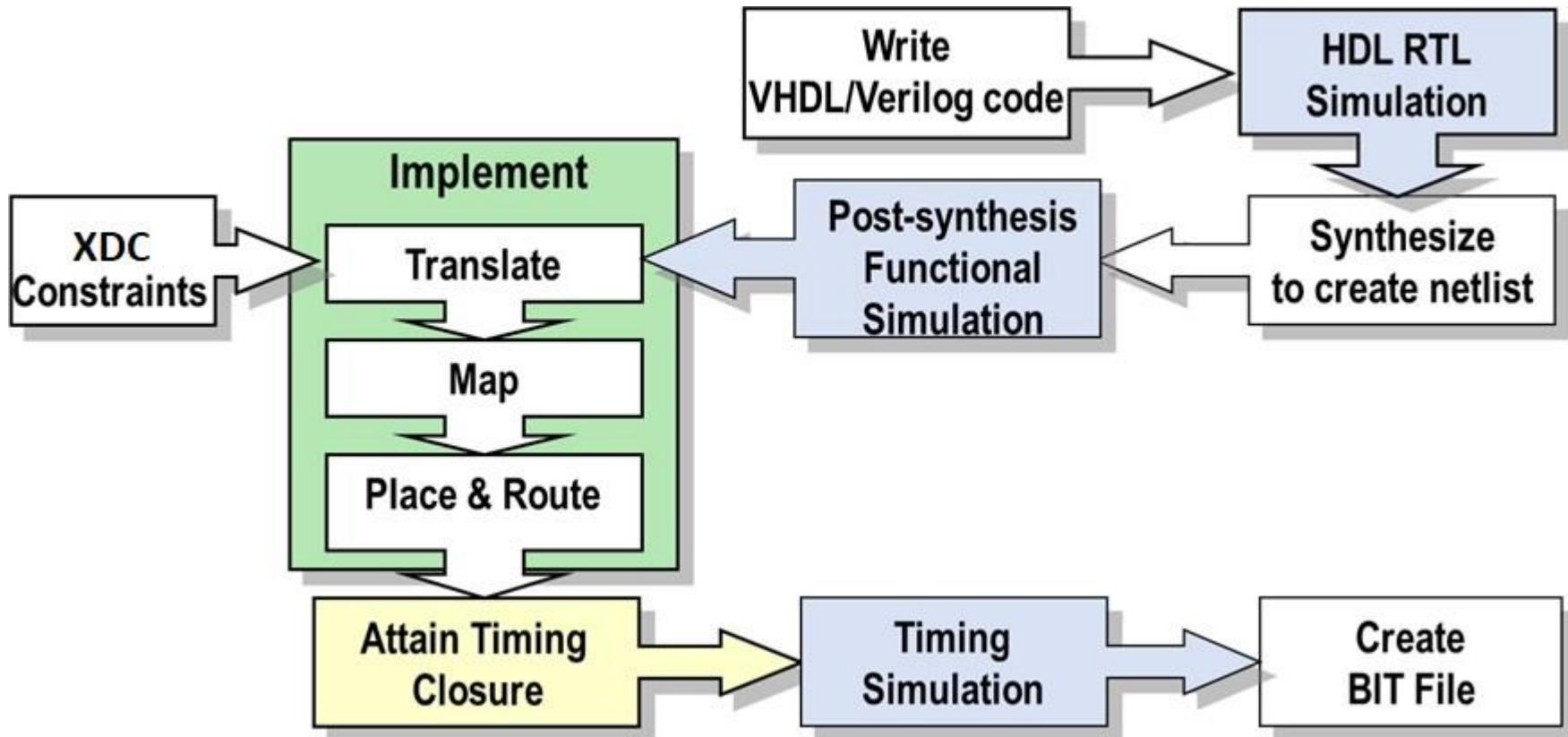
❑ Each DCM contains

- Delay-Locked Loop (DLL):
 - eliminates clock skew by compensating for delays in clock routing
 - http://en.wikipedia.org/wiki/Clock_skew
 - can be used to condition (or clean) an input clock to give an output clock with a 50% duty cycle
- Digital Frequency Synthesizer (DFS): generates clocks whose frequency is determined by multiplication/division of the input clock frequency
- Phase Shift (PS) unit: generates clocks whose phase is shifted in relation to the input clock

❑ Unlike internal memory, a DCM cannot be inferred

- Use Clocking Wizard within the Xilinx CORE Generator tool to create a custom clocking component that you can instantiate in your own SystemVerilog code
- Instantiate a suitable Xilinx DCM primitive (*DCM* or *DCM_SP*).

Xilinx Design Flow



Xilinx Simulation Libraries

- ❑ **UNISIM**: a collection of simulation primitives (e.g. DCM, BRAM primitives) for RTL and post-synthesis *functional simulation*.
- ❑ **XilinxCoreLib** : required for RTL & post-synthesis functional simulation if Xilinx CORE Generator tool was used in the design
- ❑ **SIMPRIM**: collection of simulation primitives (e.g. DCM, BRAM primitives) for *Timing Simulations*
- ❑ Xilinx Isim simulator automatically uses these libraries. If you use a third-party simulator, then you must compile the libraries and configure the simulator to use them

Xilinx Synthesis Stage

- ❑ Xilinx Synthesis (XST) converts the VHDL/SystemVerilog code, plus any cores generated by the Core Generator tool, into a gate-level netlist, which is represented in terms of the UNISIM primitives.
 - ❑ Note that most of these primitives are inferred from the VHDL/SystemVerilog code rather than explicitly instantiated.
- ❑ Netlist is in a Xilinx-specific format known as an NGC (Native Generic Circuit) file instead of the industry-standard EDIF (Electronic Design Interchange Format) format.
- ❑ If you wish to use the netlist as an input to a third-party tool, then you must use the *ngc2edif* command line tool to convert the format. This will be required if your design is a non-target-specific IP (Intellectual Property) core.
- ❑ XST also generates a .ngr file containing the RTL schematic.

Xilinx Implementation Stage

- ❑ **Translate:** merges all input netlists (including any third-party IP cores) and design constraint information (e.g. XDC or UCF) and outputs a Xilinx NGD (Native Generic Database) **netlist file** which is represented in terms of technology independent primitives (AND gates, flip-flops, RAMs, etc.). These primitives are similar to the SIMPRIM timing simulation primitives.
- ❑ **Map:** maps the NGD netlist into FPGA elements, such as CLBs and IOBs. Output is an NCD (Native Circuit Description) file that is specific to the targetted FPGA device.
- ❑ **Place and Route:** Takes the NCD file as input and performs placement of logic resources and routing of connections. Output is also in NCD format. A device Programming File (**bitstream**) can then be generated from this NCD file for downloading to the actual FPGA device.

Design Implementation Constraints

- ❑ A VHDL/SystemVerilog model cannot fully describe all the characteristics of a complete design for a Xilinx FPGA.
- ❑ Details such as I/O pin mappings and timing constraints are typically specified in a separate constraints file, the Xilinx Design Constraints (XDC) being the most common

Placement constraint – I/O pin mapping

- ❑ I/O pin mapping is one example of a placement constraint.
- ❑ Mapping is usually configured using the Xilinx PlanAhead tool which saves the data to the XDC file.
- ❑ Alternatively, you can create and edit the editing the XDC file directly in the Xilinx Vivado
- ❑ Example

```
NET "reset" LOC = B6;
```

This assigns the signal *reset* of the design's top-level module to pin *B6* of the FPGA device

Xilinx Timing Closure & Constraints

- ❑ Timing closure is the process by which a design for an FPGA is modified to meet its timing requirements (e.g. a particular clock frequency). Most of these modifications are handled automatically by the synthesis/mapping/place-and-route tools, although manual intervention may also be necessary.
- ❑ Timing requirements are specified as timing constraints which ensure that the design will perform the tasks it was designed to do in a specific time frame.
- ❑ Timing constraints are applied to the path or paths (known as the net) taken from one FPGA element to the input of subsequent elements in the FPGA or on the PCB.
- ❑ The most important timing constraints are PERIOD, OFFSET IN, OFFSET OUT and FROM:TO
- ❑ Timing constraints can be specified using the Xilinx Constraints Editor or by directly editing the XDC file.

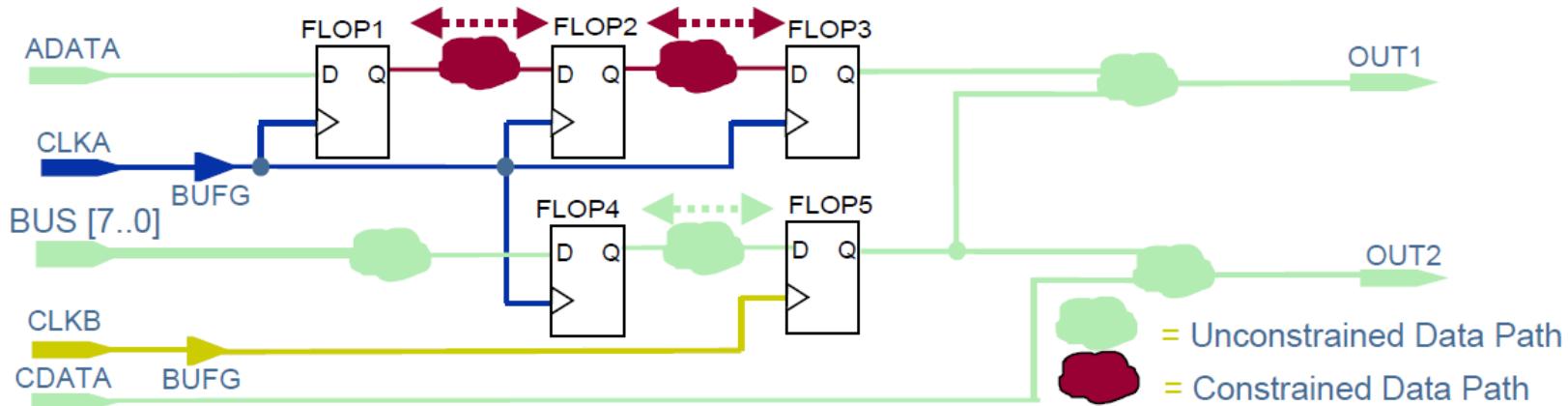
Why are Timing Constraints necessary?

- ❑ You may wonder why we can't just run the tools (synthesis, mapping and, place and route) and see what they tell us is the maximum clock frequency our design can operate at
- ❑ The reason is that the timing constraints determine how much effort (processing time) the tools put into optimising various logic blocks, positioning these blocks and choosing the routing between them
- ❑ The timing characteristics of the external signals into and out of the FPGA also affect how the placement and routing should be carried out

PERIOD Constraint

- ❑ Each PERIOD constraint covers paths between synchronous elements clocked by the reference clock net

```
NET "CLKA" TNM_NET = "clka_tnm";  
TIMESPEC TS_clk = PERIOD "clka_tnm" 10 ns HIGH 50 %;
```



PERIOD Constraint – Example

```
NET "CLKA" TNM_NET = "clka_tnm";  
TIMESPEC TS_clk = PERIOD "clka_tnm" 10 ns HIGH 50 %;
```

- ❑ The first line creates a timing group called *clka_tnm* which contains all of the downstream synchronous elements (e.g. flip-flops) that the net (i.e. signal) *CLKA* drives.
- ❑ The second line constrains all of the paths between these synchronous elements with the timing specification of 10 ns, the period of the clock, with a duty cycle of 50%.
- ❑ PERIOD constraint analysis includes the setup and hold analysis on the synchronous elements.
- ❑ The Map and PAR (Place & Route) tools will strive to meet this constraint and will report a timing violation if they fail.

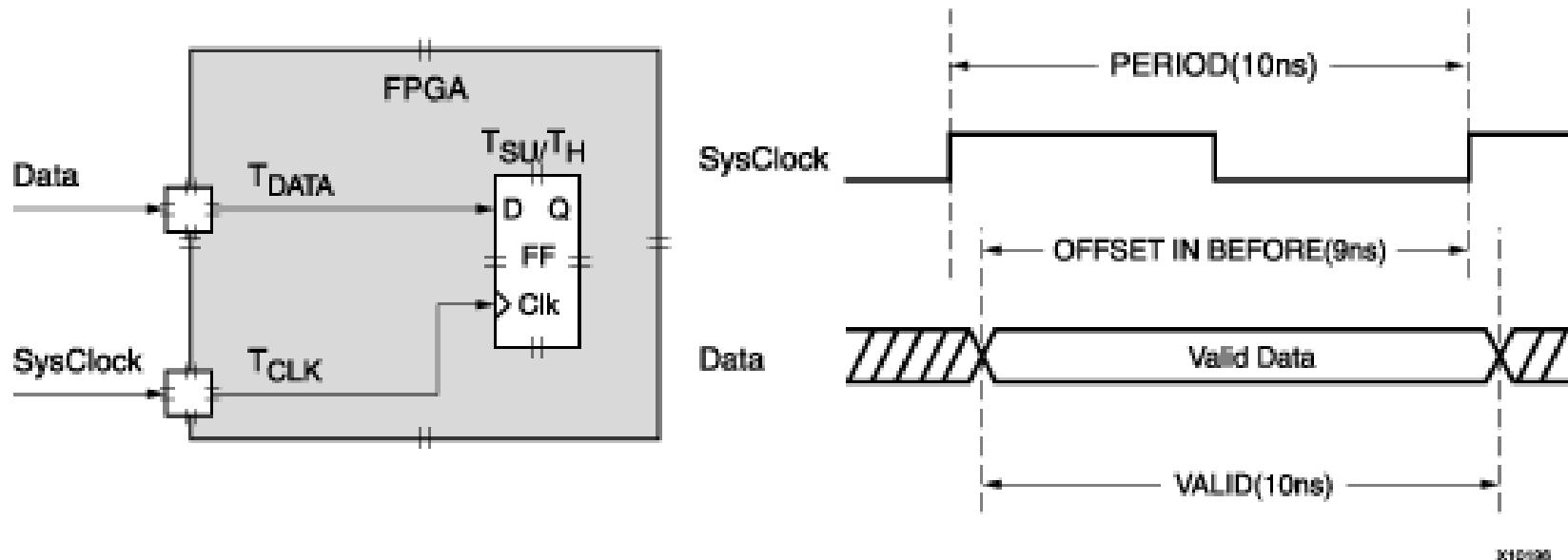
PERIOD Constraint

- ❑ $\text{Setup slack} = \text{constraint_requirement} + T_{\text{skew}} - T_{\text{data}} - T_{\text{su}}$
- ❑ $\text{Hold slack} = T_{\text{data}} - T_{\text{skew}} - T_h$
 - $\text{constraint_requirement}$: clock period specified in constraint
 - T_{skew} : Clock path skew is how much later the clock edge arrives at the second synchronous element with respect to the first
 - T_{data} : Data path delay between flip-flops
 - T_{su} : Flip-flop setup time
 - T_h : Flip-flop hold time
- ❑ Negative slack in the timing report indicates a timing violation
- ❑ Setup violation on a PERIOD constraint often indicates too much combinational logic between flip-flops or high fan-out from flip-flop (Solutions discussed on upcoming slides).
- ❑ Hold violation on a PERIOD constraint is less common and indicates what is known as a “fast path”. This can be resolved by introducing a delay on the data path and/or ensuring that the FPGA’s low-skew global clock network is being used for the clock signal.

OFFSET IN BEFORE Constraint

- ❑ OFFSET IN timing constraints are used to define the timing relationship between an external clock pad and its associated data-in pad.
 - ❑ This relationship is also known as constraining the *Pad-to-Setup* path on the device.
 - ❑ These constraints are important for specifying timing interfaces with external components.
- ❑ OFFSET IN BEFORE constraint defines the time available for data to propagate from the pad and setup at the synchronous element and how long the data will be valid for.

OFFSET IN BEFORE Constraint Timing Diagram



- ❑ **Setup slack** = $T_{offset_IN_BEFORE} - T_{data} + T_{clk} - T_{su}$
 - $T_{offset_IN_BEFORE}$ = Overall Setup Requirement
- ❑ **Hold slack** = $T_{offset_IN_BEFORE_VALID} + T_{data} - T_{clk} - T_h$
 - $T_{offset_IN_BEFORE_VALID}$ = Overall Hold Requirement

OFFSET IN BEFORE Example

```
OFFSET = IN 9 ns VALID 10 ns BEFORE "clk";
```

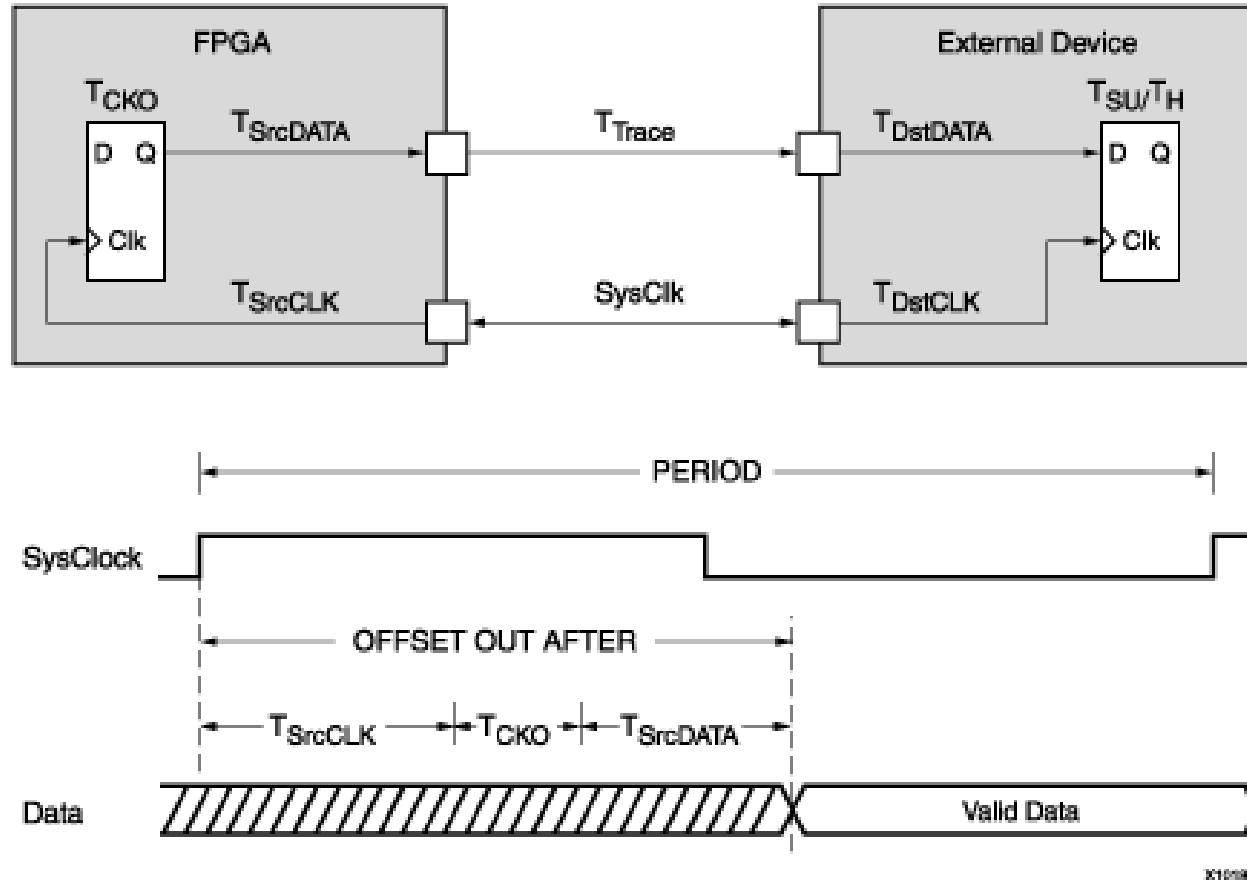
- ❑ This timing constraint tells the tools that data will arrive at the input pads 9 ns before the *c/k* rising edge and that it will remain valid for 10 ns.
- ❑ The tools will check the setup and hold timing requirements of all synchronous elements associated with the constraint (i.e. those whose data & clock inputs are derived from the external data and clock) and will report a timing violation in case of a failure to meet the requirements.
- ❑ $\text{Setup slack} = T_{\text{offset_IN_BEFORE}} - T_{\text{data}} + T_{\text{clk}} - T_{\text{su}}$
 - $T_{\text{offset_IN_BEFORE}}$ = Overall Setup Requirement
- ❑ $\text{Hold slack} = T_{\text{offset_IN_BEFORE_VALID}} + T_{\text{data}} - T_{\text{clk}} - T_{\text{h}}$
 - $T_{\text{offset_IN_BEFORE_VALID}}$ = Overall Hold Requirement
- ❑ A negative slack in the timing report indicates a timing violation

OFFSET OUT AFTER Constraint

- ❑ OFFSET OUT timing constraints are used to define the timing relationship between an external clock pad and its associated data-out pad.
 - ❑ This relationship is also known as constraining the *Clock-to-Out* path on the device.
 - ❑ These constraints are important for specifying timing interfaces with external components.
- ❑ The output timing requirements of the FPGA device are based on the timing requirements of the system and downstream device.
- ❑ The OFFSET OUT AFTER constraint defines *the maximum amount of time the data signal has to propagate from the internal register to the I/O pin of the device*.
 - ❑ The total internal propagation delay includes the clock path delay, the clock-to-output time of the register, and the data path delay from the register to the I/O pin.

OFFSET OUT AFTER Constraint

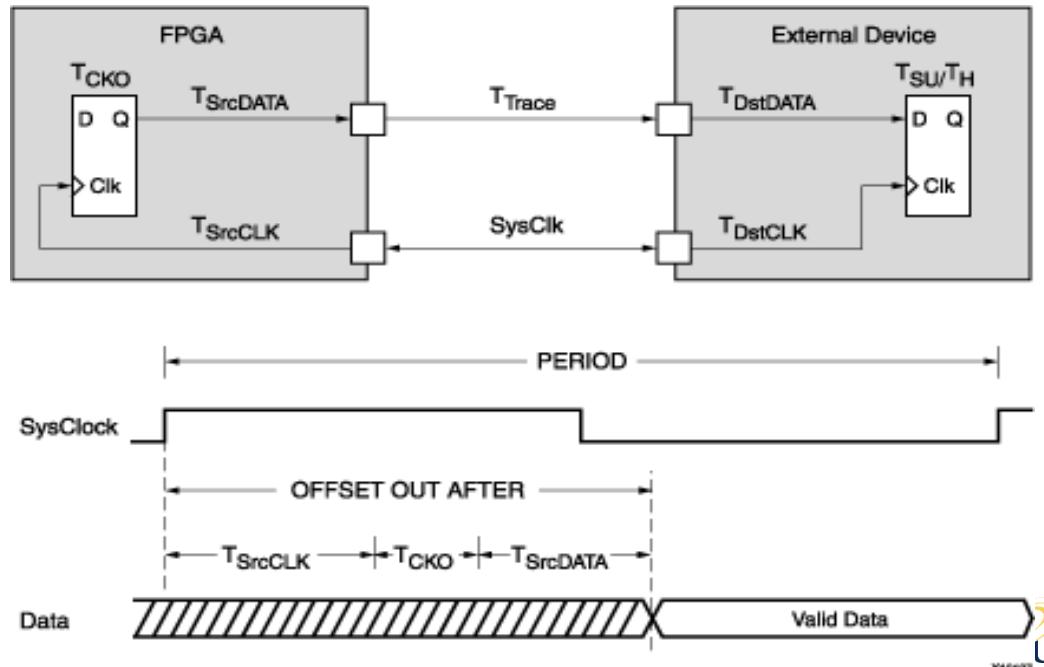
Timing Diagram



OFFSET OUT AFTER Example

```
OFFSET = OUT 3 ns AFTER "clk";
```

- ❑ This constraint tells the tools that you need to ensure data is at the output pin of the FPGA 3 ns after a clock transition at the input to the FPGA.
- ❑ There also needs to be a PERIOD constraint on *clk* so that the tools understand the clocking structure
- ❑ $\text{Slack} = T_{\text{offset_OUT_AFTER}} - (T_{\text{SrcData}} + T_{\text{CKO}} + T_{\text{SrcCLK}})$
- ❑ A negative slack in the timing report indicates a timing violation



Xilinx Static Timing Analysis

- ❑ Timing can be checked using the Timing Analyzer tool after mapping and after place & route.
- ❑ A timing report is generated which shows each of the defined timing constraints, along with the most critical path for the constraint and the corresponding “*path slack*”. The *path slack* is the difference between the required time and the actual path delay.
- ❑ Positive slack means the design works with the specified clock frequency, whereas a negative slack indicated a failure known as a “*timing violation*”.

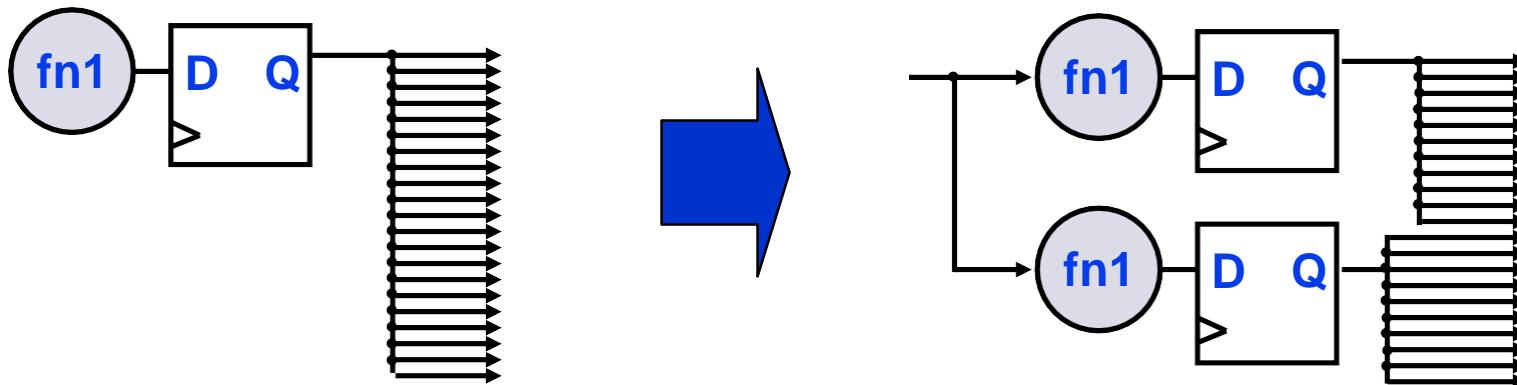
Achieving Timing Closure

❑ What do you do if there is a timing violation?

- if your Place & Route tool allows you to configure its effort level, set it to *high*.
- duplicate flip-flops in your VHDL/SystemVerilog design that have a high fanout.
- add more pipelining in your VHDL/SystemVerilog design, i.e. you may be able to split up combinational logic between flip-flops.
- if an external data signal changes before the clock edge, you need to *delay the clock edge using a DCM* (or PLL). Alternatively, you can use the IDELAY element in the IOB to move the data to where the clock is valid.
- make use of flip-flops within IOBs. Specific registers in your code can be placed in IOBs using the Constraints Editor, or you can configure the Map tool to pack I/O registers into IOBs.
- Manually modify the placing and routing using the Xilinx FPGA Editor tool. You should only do this as a last resort.

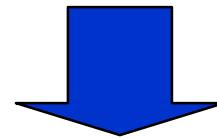
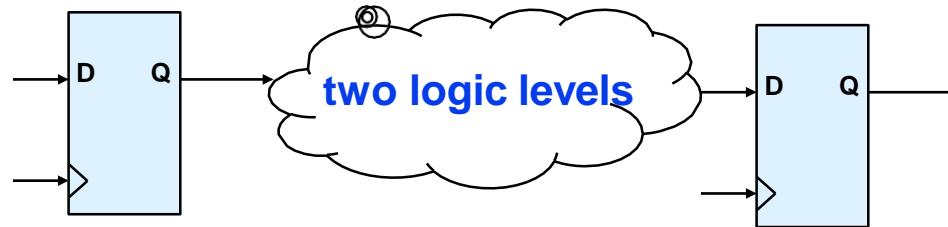
Duplicating flip-flops

- ❑ Duplicating flip-flops reduces the fanout from the flip-flop. This
 - shortens net delay
 - reduces routing congestion

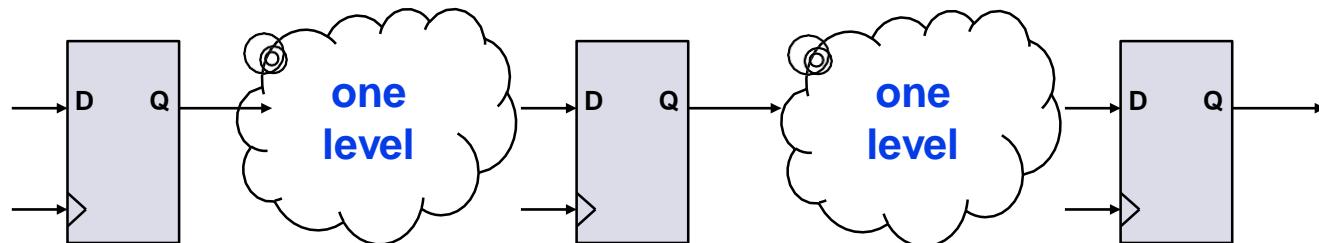


Pipelining

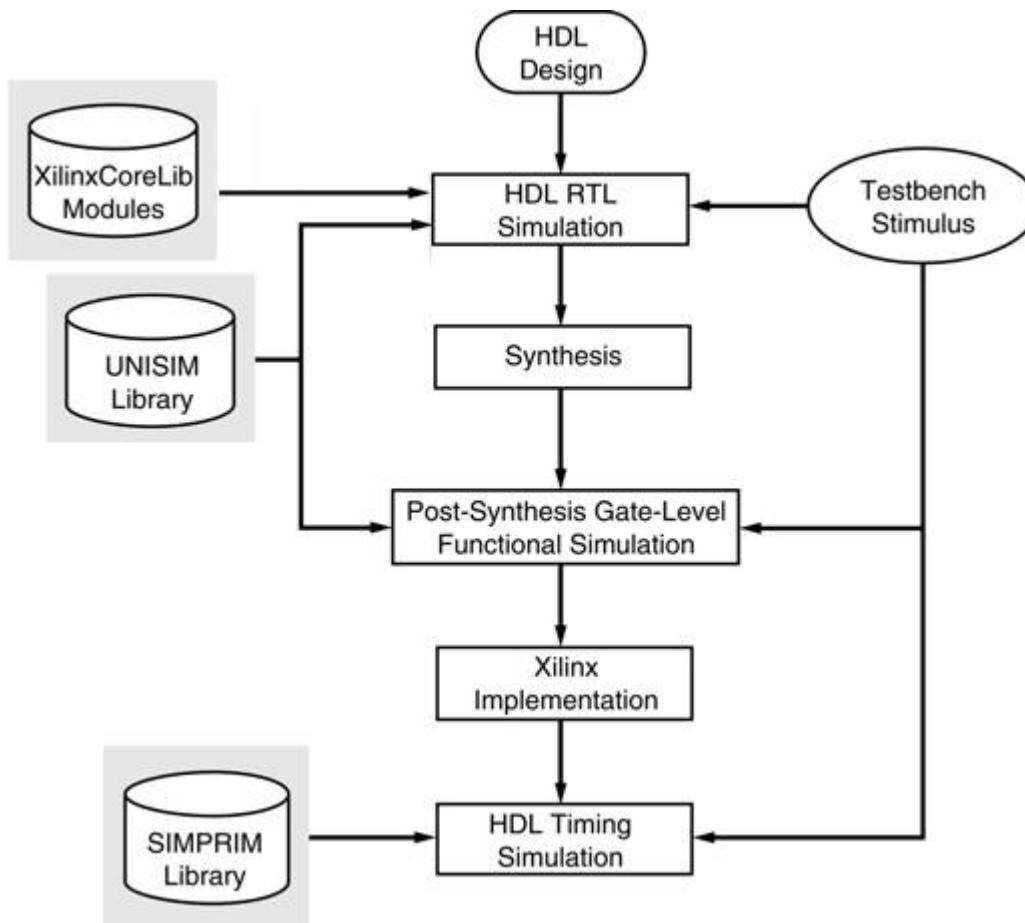
$$f_{\text{MAX}} = n \text{ MHz}$$



$$f_{\text{MAX}} \approx 2n \text{ MHz}$$



Xilinx Simulation Points



Xilinx HDL RTL Simulation

- ❑ This functional simulation corresponds to behavioral simulation in Xilinx Vivado.
- ❑ Allows you to verify a VHDL/SystemVerilog description at the system level and confirm that the code functions as expected.
- ❑ Make use of the test bench you code in VHDL/SystemVerilog.
- ❑ UNISIM library is available to support explicit instantiations of primitives in the VHDL/SystemVerilog.
- ❑ XilinxCoreLib library is available to support instances of CORE Generator components.
- ❑ No timing delay information included (unless modelled in your VHDL/SystemVerilog code)

Other (optional) Xilinx functional simulation points

- ❑ **Post-Synthesis Gate-Level Functional Simulation** : This may be used to perform a functional simulation if you use a third-party synthesis tool instead of XST and the tool produces a UNISIM-compatible netlist. The NetGen program takes the third-party netlist and produces a UNISIM-based netlist in VHDL/Verilog, which is then simulated.
- ❑ **Post-Translate Simulation**: This may be used to perform a functional simulation if your third-party synthesis tool cannot produce a UNISIM-compatible netlist. The NGD (Native Generic Database) file from the translation step is taken, and the NetGen program produces a SIMPRIM-based netlist in VHDL/Verilog which is then simulated.

Xilinx Timing Simulation

Post-Map Timing Simulation:

- optional
- similar to post place and route timing simulation but does not include routing delays.

Post-Place and Route Timing Simulation:

- Performed after the Place and Route process
- Accounts for both block delays and routing delays
- You see how the actual circuit is likely to behave
- you can reuse your functional testbench
- NetGen program converts NCD netlist into a structural simulation netlist in VHDL/SystemVerilog, based on the SIMPRIM models for the device primitives in the design, and an associated **Standard Delay Format (SDF)** file that contains all of **the annotated delays** for the design for use during simulation.
- The simulator uses the structural simulation netlist, the SDF file and the testbench to perform the timing simulation.
- This is the final “sign-off” timing simulation before generating the configuration bitstream file for downloading to Xilinx devices.

-
- ❑ NetGen is an application which takes implementation files from different stages of the design and creates netlists for verification.
 - ❑ It is generally used for creating
 - simulation netlists
 - netlists for Equivalency Checking
 - netlists for Static Timing Analysis.