Digital Design & Computer Arch.

Lecture 11: Microarchitecture Fundamentals

Prof. Onur Mutlu

ETH Zürich

Spring 2022

31 March 2022

Assignment: Lecture Video (April 1)

- Why study computer architecture? Why is it important?
- Future Computing Platforms: Challenges & Opportunities

- **Required Assignment**
  - **Watch one of** Prof. Mutlu's lectures and analyze either (or both)
  - https://www.youtube.com/watch?v=kgiZlSOcGFM (May 2017)
  - https://www.youtube.com/watch?v=mskTeNnf-i0 (Feb 2021)

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page summary** of one of the lectures and email us
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to Moodle by April 1

Extra Assignment: Moore's Law (I)

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page review**
  - Upload PDF file to Moodle – Deadline: April 7

- I strongly recommend that you follow my guidelines for (paper) review (see next slide)

Extra Assignment 2: Moore's Law (II)

- **Guidelines on how to review papers critically**

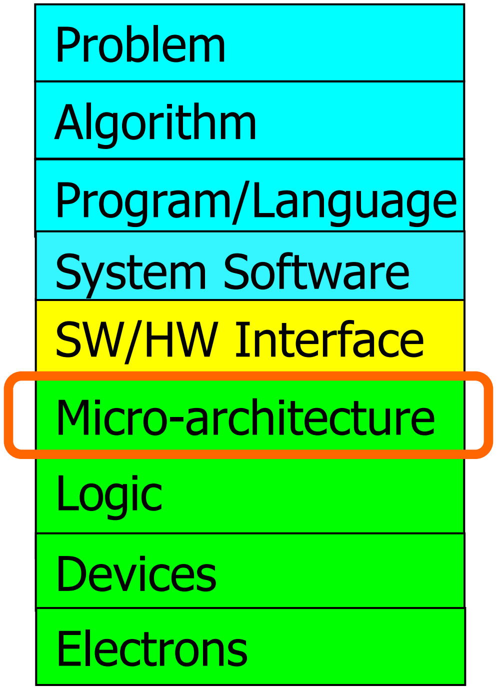
  - Guideline slides: pdf ppt
  - Video: https://www.youtube.com/watch?v=tOL6FANAJ8c

  - Example reviews on "Main Memory Scaling: Challenges and Solution Directions" (link to the paper)
    - Review 1
    - Review 2

  - Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" (link to the paper)
    - Review 1

Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Pipelining
- Issues in Pipelining:
  - Control & Data Dependence Handling
  - State Maintenance and Recovery
- Out-of-Order Execution



Problem

Algorithm

Program/Language

System Software

SW/HW Interface

Micro-architecture

Logic

Devices

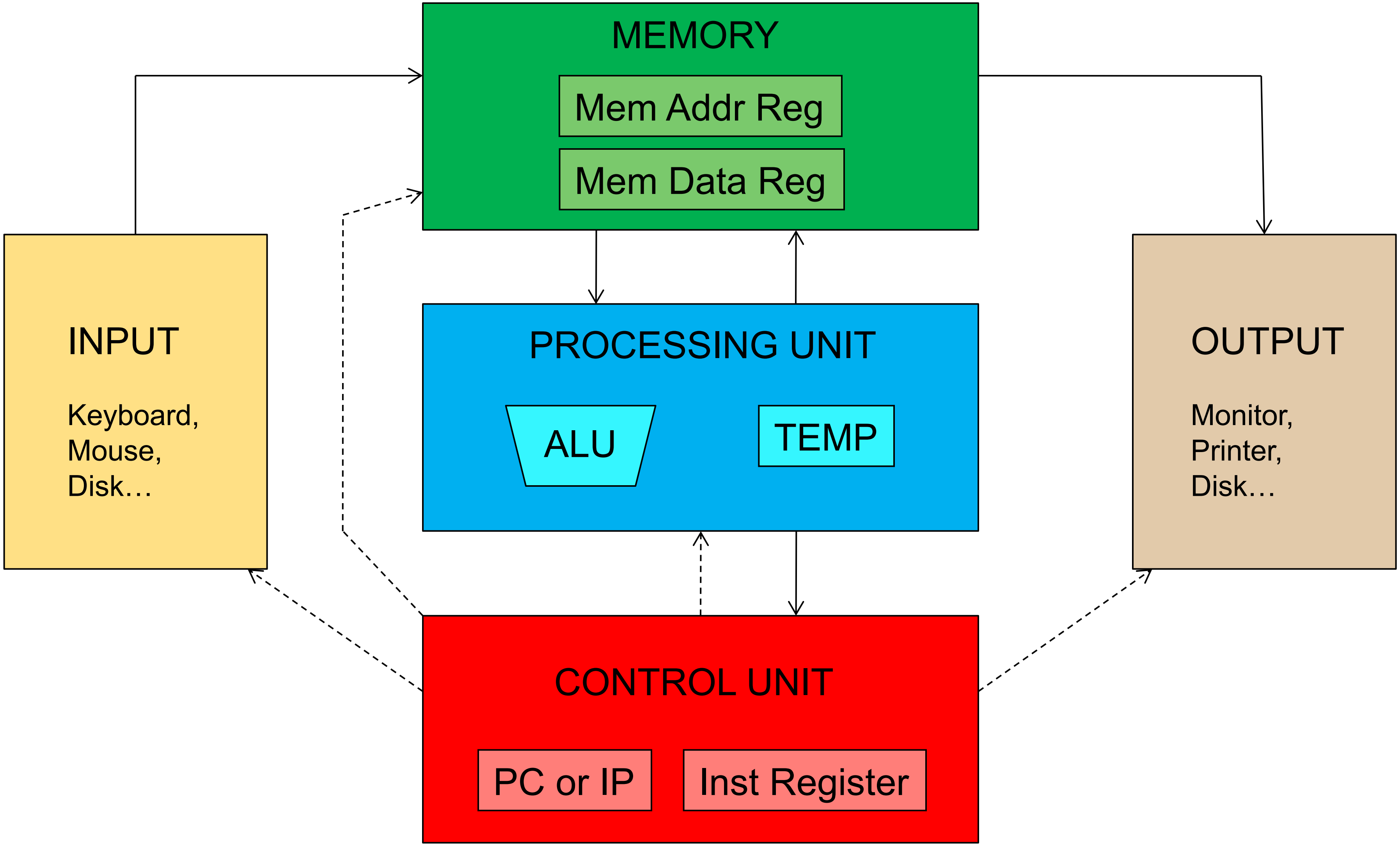
Electrons

Readings

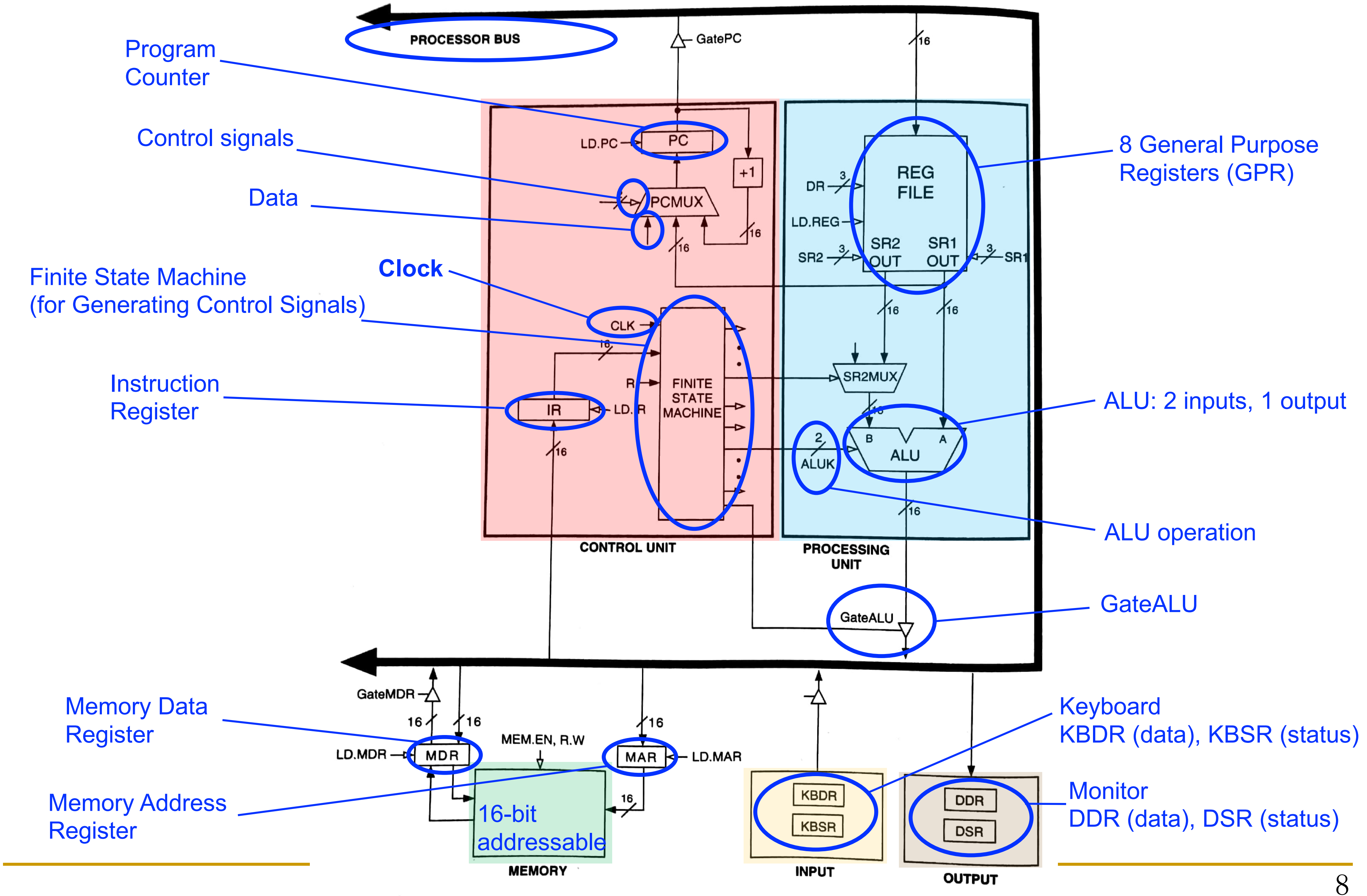
- This week
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C

- Next week
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.7, 7.8.1-7.8.3

Recall: The von Neumann Model



Recall: LC-3: A von Neumann Machine



Program Counter

Control signals

Data

Finite State Machine (for Generating Control Signals)

Clock

Instruction Register

8 General Purpose Registers (GPR)

ALU: 2 inputs, 1 output

ALU operation

GateALU

Memory Data Register

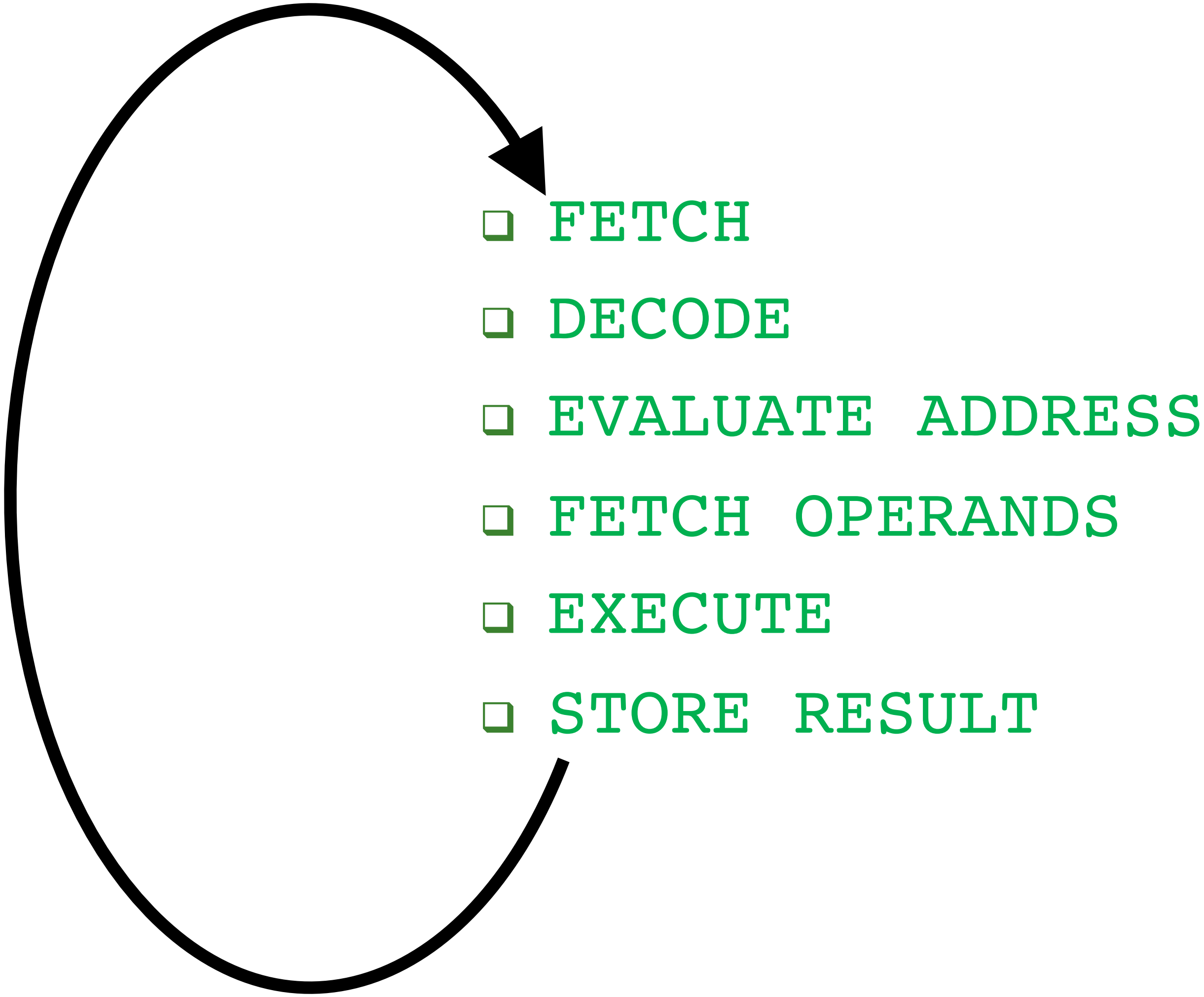
Memory Address Register

Keyboard
KBDR (data), KBSR (status)

Monitor
DDR (data), DSR (status)

Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: The Instruction Cycle

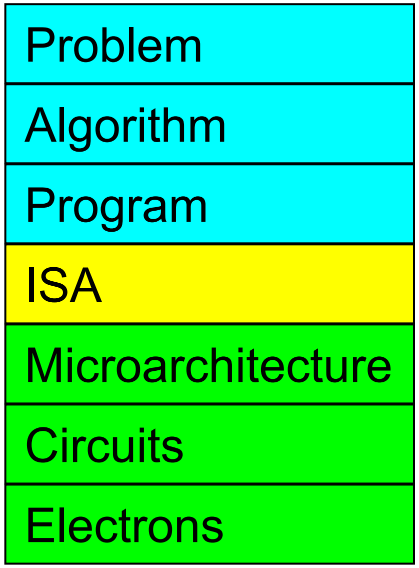


- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

Recall: The Instruction Set Architecture

- The ISA is the interface between what the software commands and what the hardware carries out

- The ISA specifies
  - The memory organization
    - Address space (LC-3: 2^{16}, MIPS: 2^{32})
    - Addressability (LC-3: 16 bits, MIPS: 8 bits)
      - Word- or Byte-addressable
  - The register set
    - 8 registers (R0 to R7) in LC-3
    - 32 registers in MIPS
  - The instruction set
    - Opcodes
    - Data types
    - Addressing modes
    - Length and format of instructions



Problem

Algorithm

Program

ISA

Microarchitecture

Circuits

Electrons

Microarchitecture

- An **implementation** of the ISA

- How do we implement the ISA?
  - We will discuss this for many lectures

- There can be many implementations of the same ISA
  - **MIPS** R2000, R3000, R4000, R6000, R8000, R10000, …
  - **x86**: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cove, Sapphire Rapids, …, AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, …
  - **POWER** 4, 5, 6, 7, 8, 9, 10 (IBM), …, **PowerPC** 604, 605, 620, …
  - **ARM** Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, …
  - **Alpha** 21064, 21164, 21264, 21364, …
  - **RISC-V** …
  - …

(A Bit More on)
ISA Design and Tradeoffs

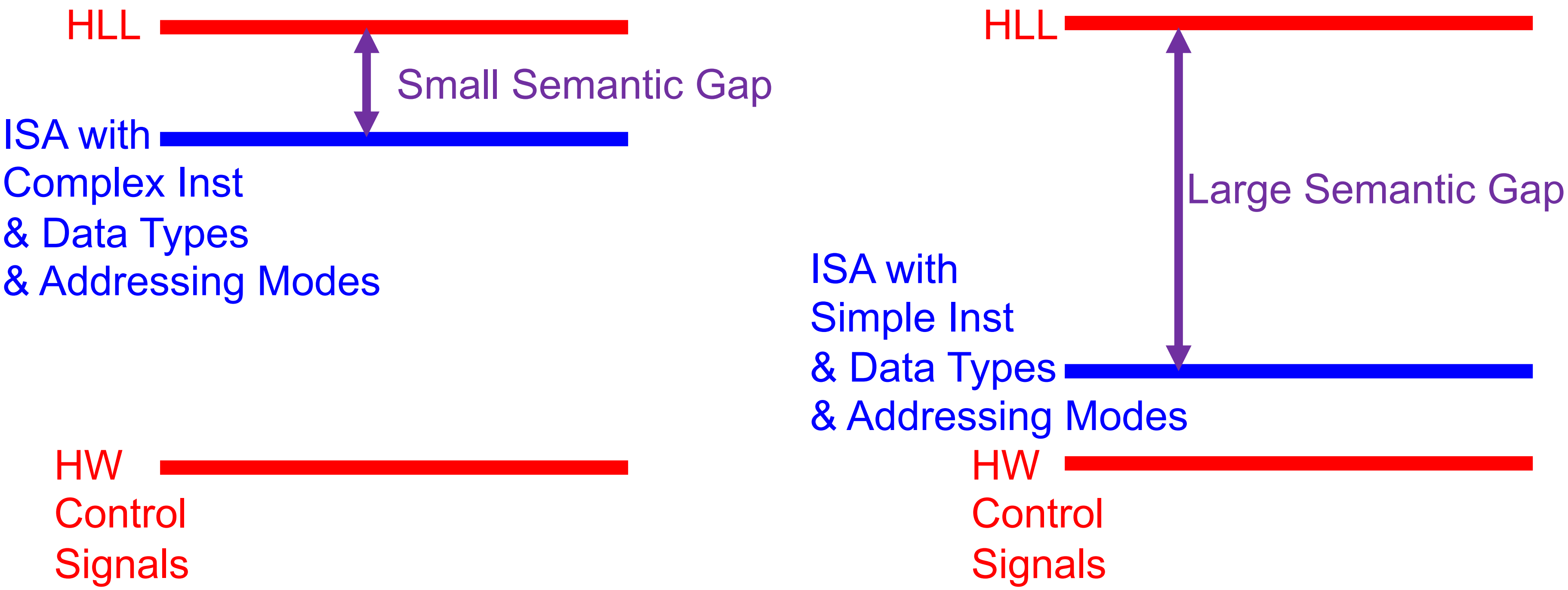
Many Different ISAs Over Decades

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM, RISC-V, …

- What are the fundamental differences?
  - E.g., how instructions are specified and what they do
  - E.g., how complex are instructions, data types, addr. modes

Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)



HLL

Small Semantic Gap

ISA with
Complex Inst
& Data Types
& Addressing Modes

HW
Control
Signals

HLL

Large Semantic Gap

ISA with
Simple Inst
& Data Types
& Addressing Modes

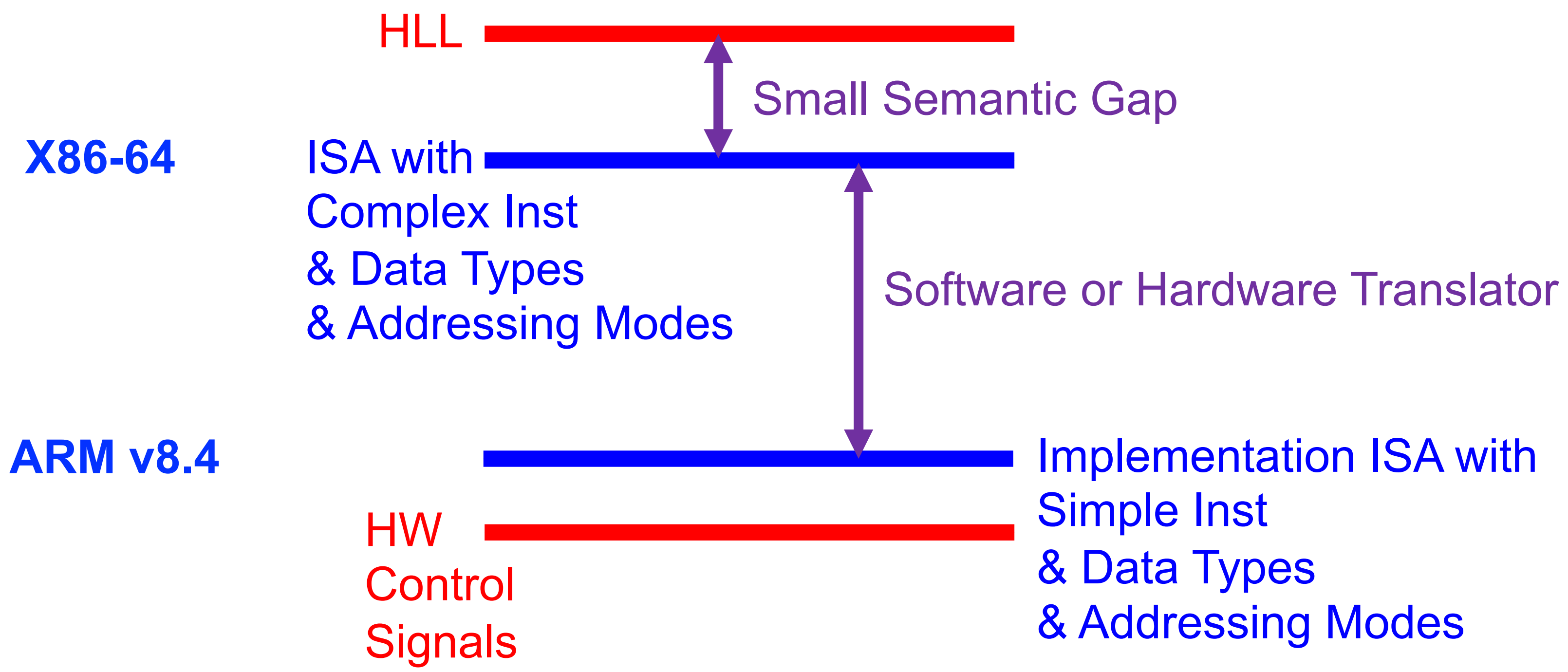
HW
Control
Signals

Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW

Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW

How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different “implementation” ISA



HLL

Small Semantic Gap

X86-64

ISA with
Complex Inst
& Data Types
& Addressing Modes

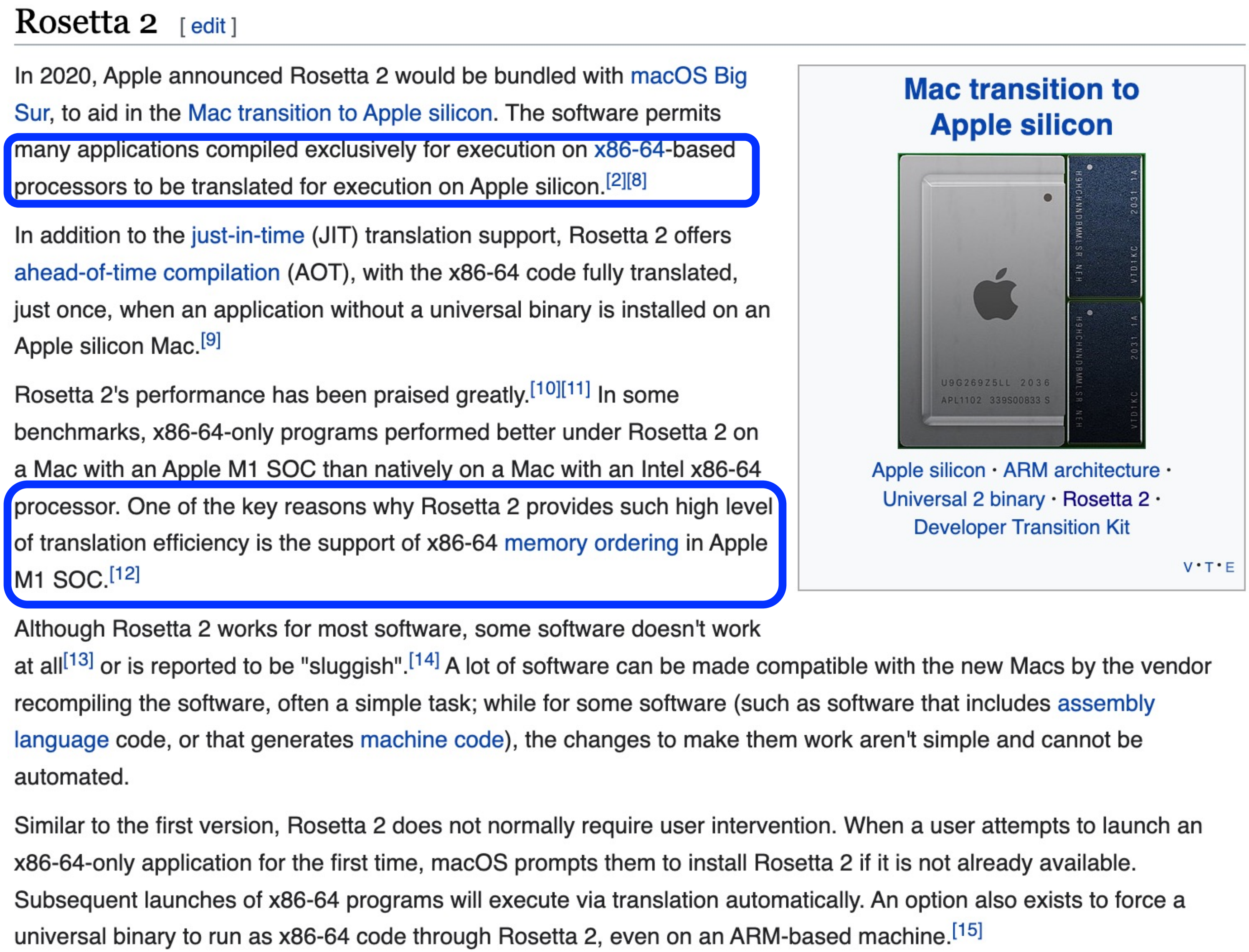
Software or Hardware Translator

ARM v8.4

Implementation ISA with
Simple Inst
& Data Types
& Addressing Modes

HW
Control
Signals

An Example: Rosetta 2 Binary Translator



Rosetta 2 [edit]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

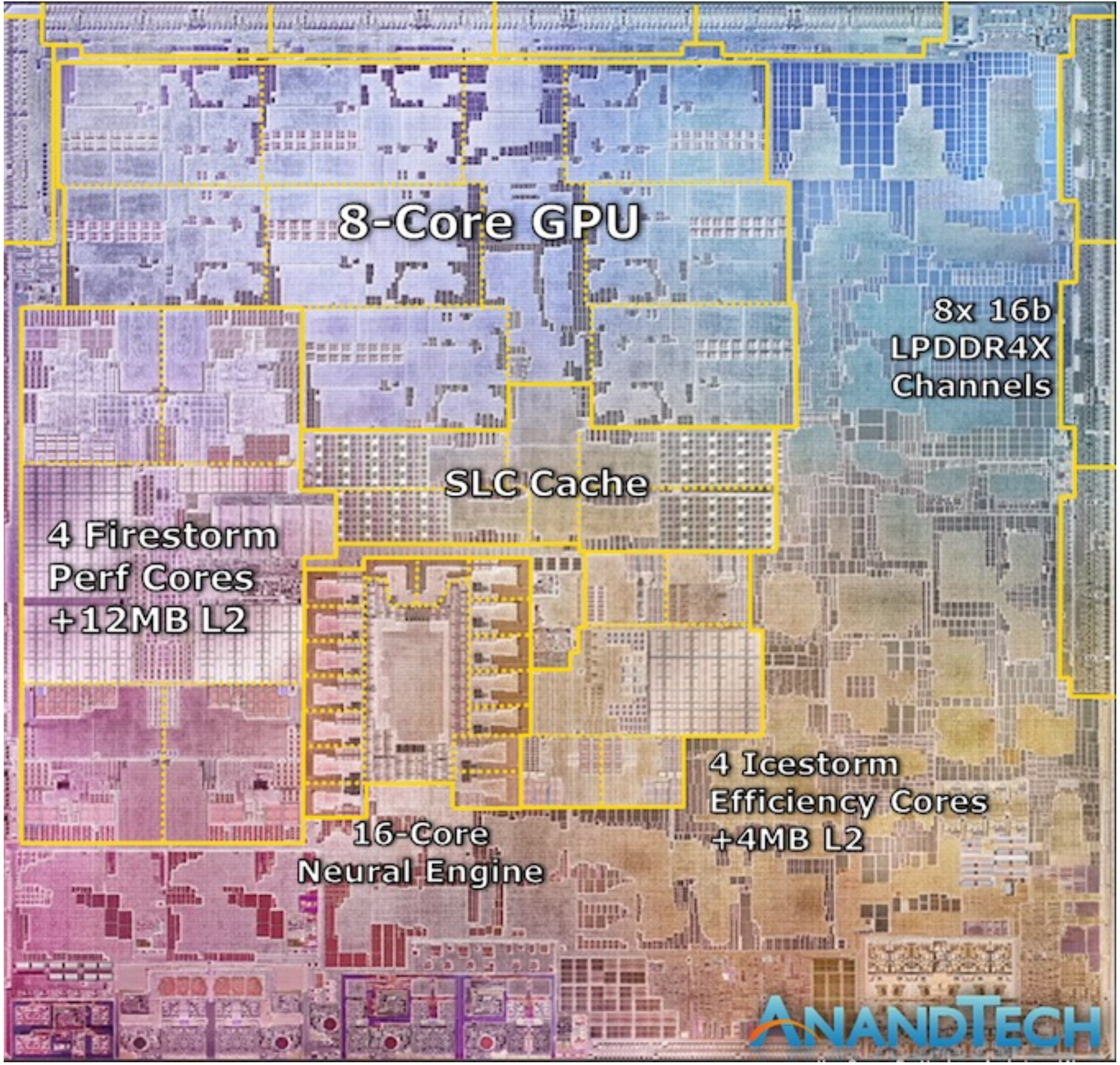
Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]

Mac transition to Apple silicon
U9G269Z5LL 2036
APL1102 339S00833 S
Apple silicon · ARM architecture · Universal 2 binary · Rosetta 2 · Developer Transition Kit
V · T · E

https://en.wikipedia.org/wiki/Rosetta_(software)#Rosetta_2

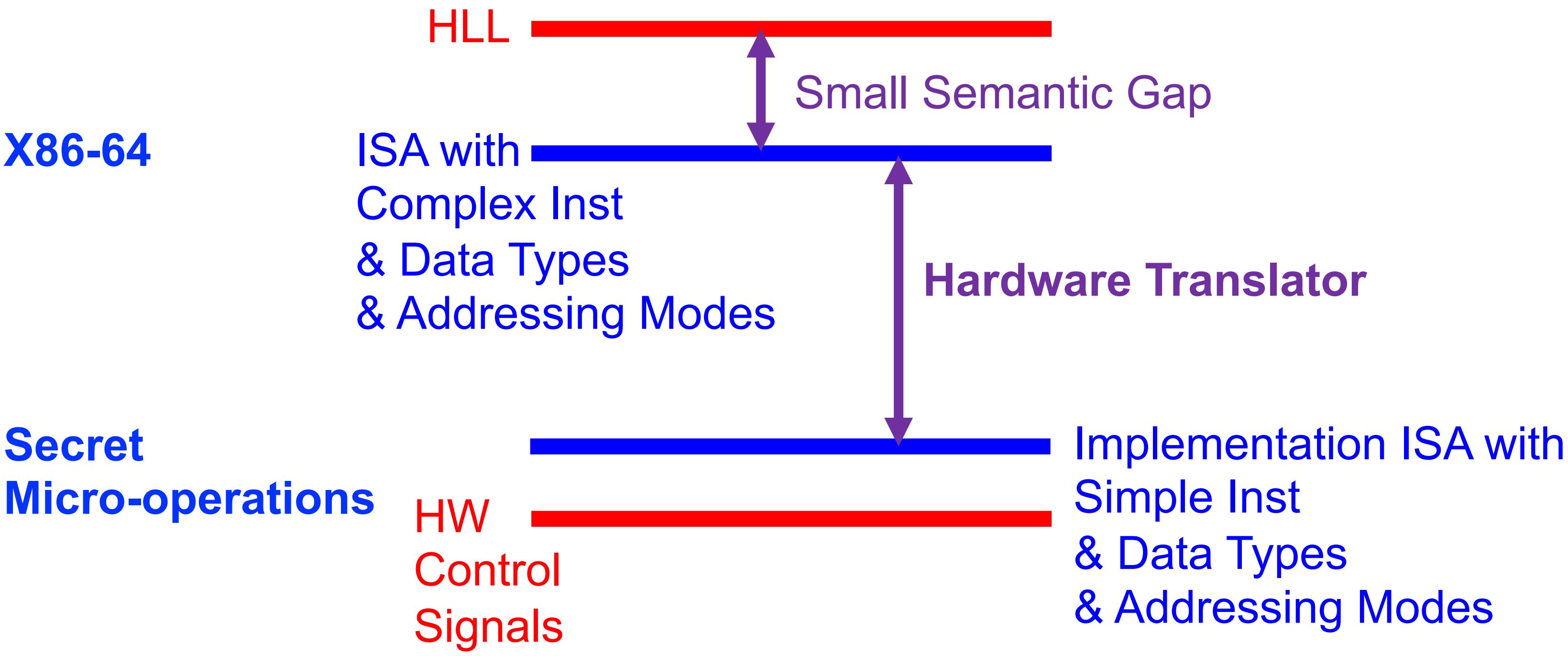
An Example: Rosetta 2 Binary Translator



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

Another Example: Intel and AMD Processors



HLL

Small Semantic Gap

X86-64

ISA with
Complex Inst
& Data Types
& Addressing Modes

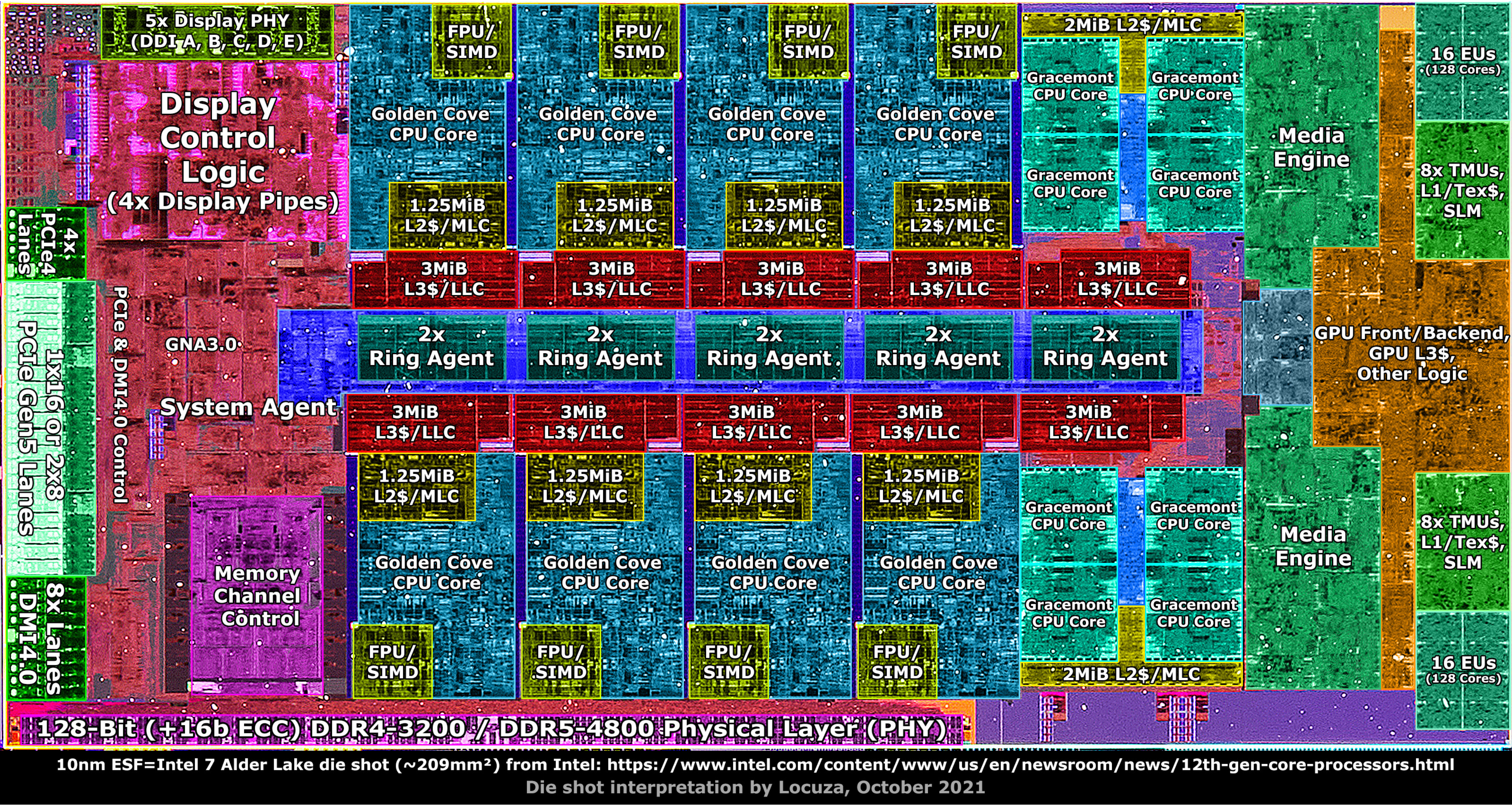
Hardware Translator

Secret
Micro-operations

Implementation ISA with
Simple Inst
& Data Types
& Addressing Modes

HW
Control
Signals

Another Example: Intel and AMD Processors



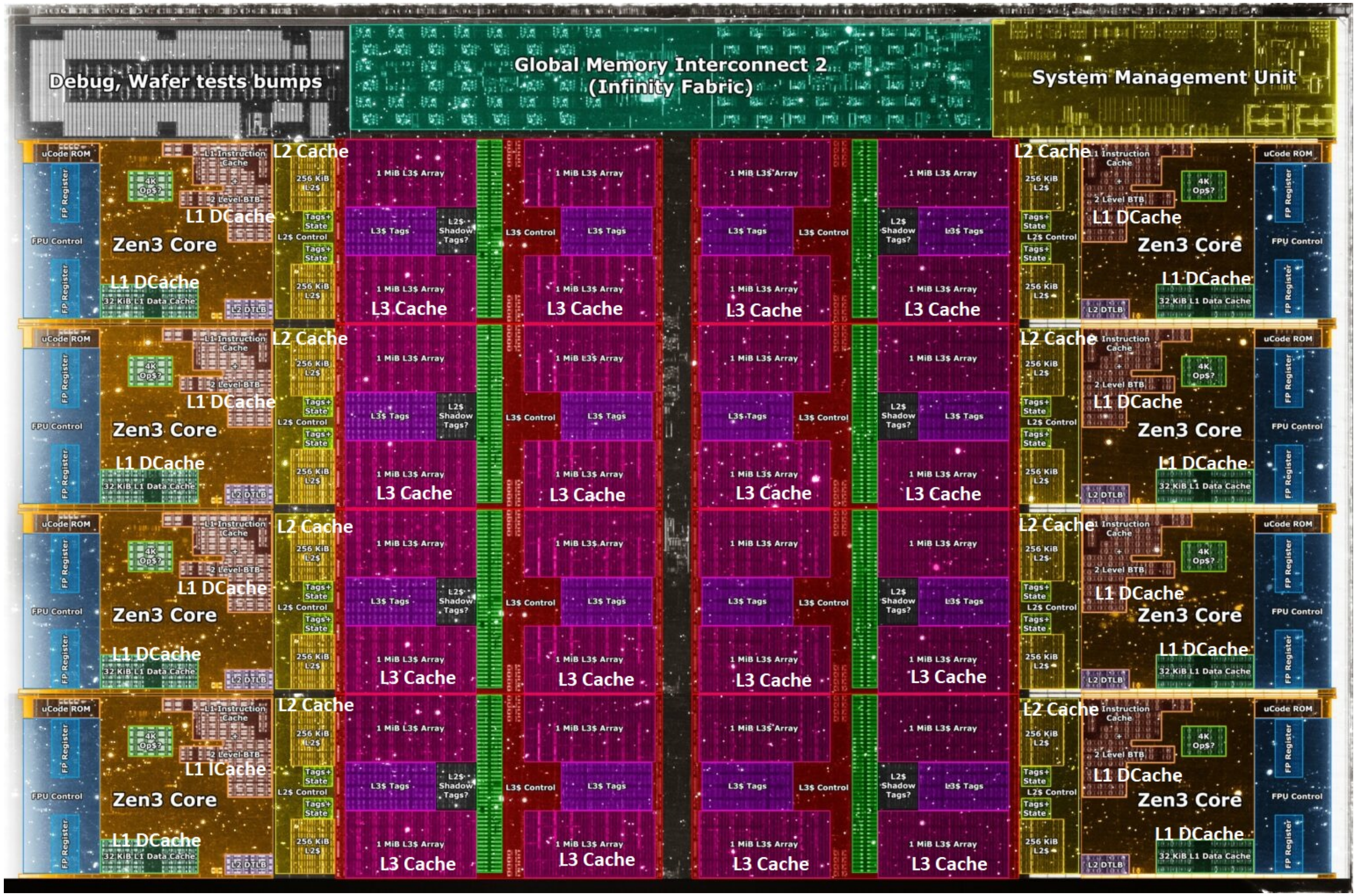
10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html

Die shot interpretation by Locuza, October 2021

Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

Another Example: Intel and AMD Processors



Core Count:

8 cores/16 threads

L1 Caches:

32 KB per core

L2 Caches:

512 KB per core

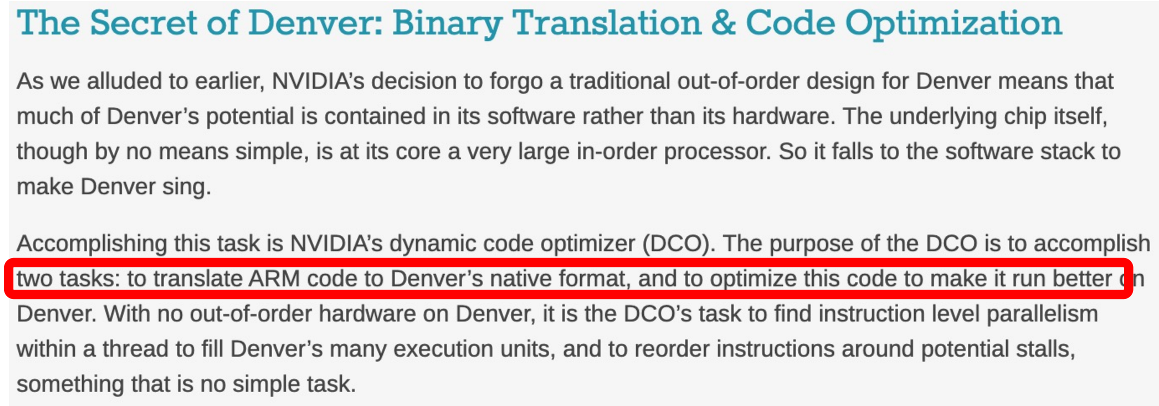
L3 Cache:

32 MB shared

AMD Ryzen 5000, 2020

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

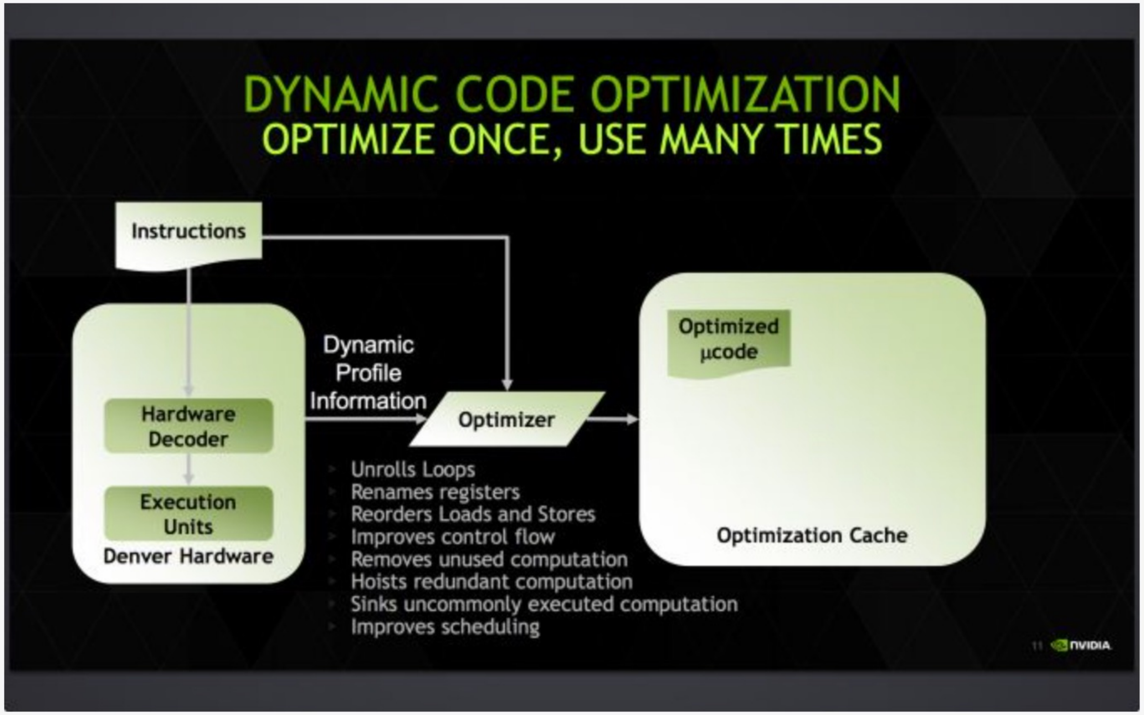
Another Example: NVIDIA Denver



The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

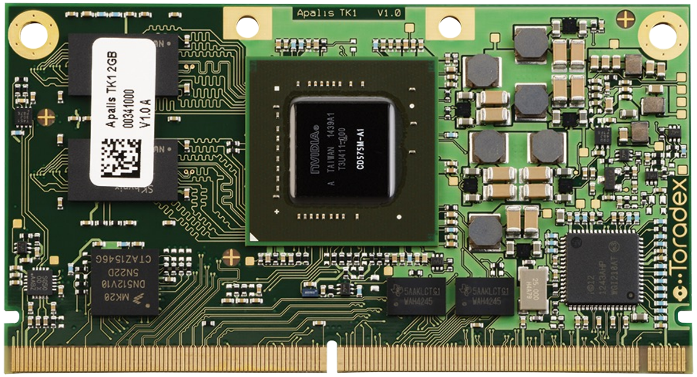
Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.



DYNAMIC CODE OPTIMIZATION

OPTIMIZE ONCE, USE MANY TIMES

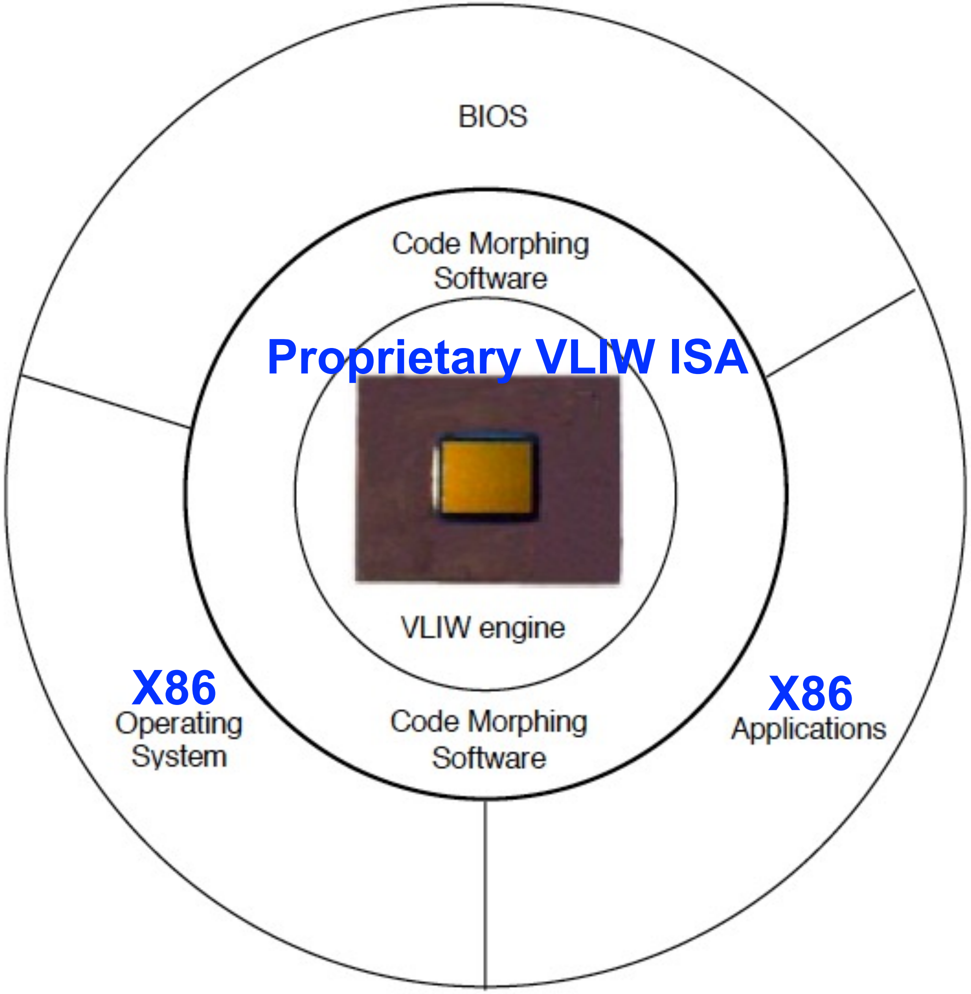
- Unrolls Loops
- Renames registers
- Reorders Loads and Stores
- Improves control flow
- Removes unused computation
- Hoists redundant computation
- Sinks uncommonly executed computation
- Improves scheduling



https://www.anandtech.com/show/8701/the-google-nexus-9-review/4

https://www.toradex.com/computer-on-modules/apalis-arm-family/nvidia-tegra-k1

Transmeta: x86 to VLIW Translation



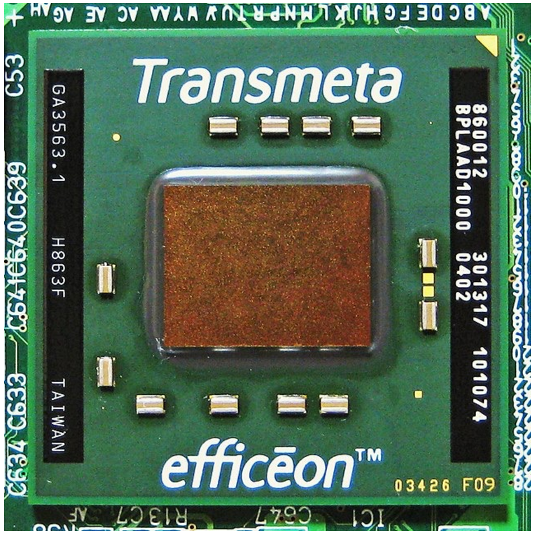
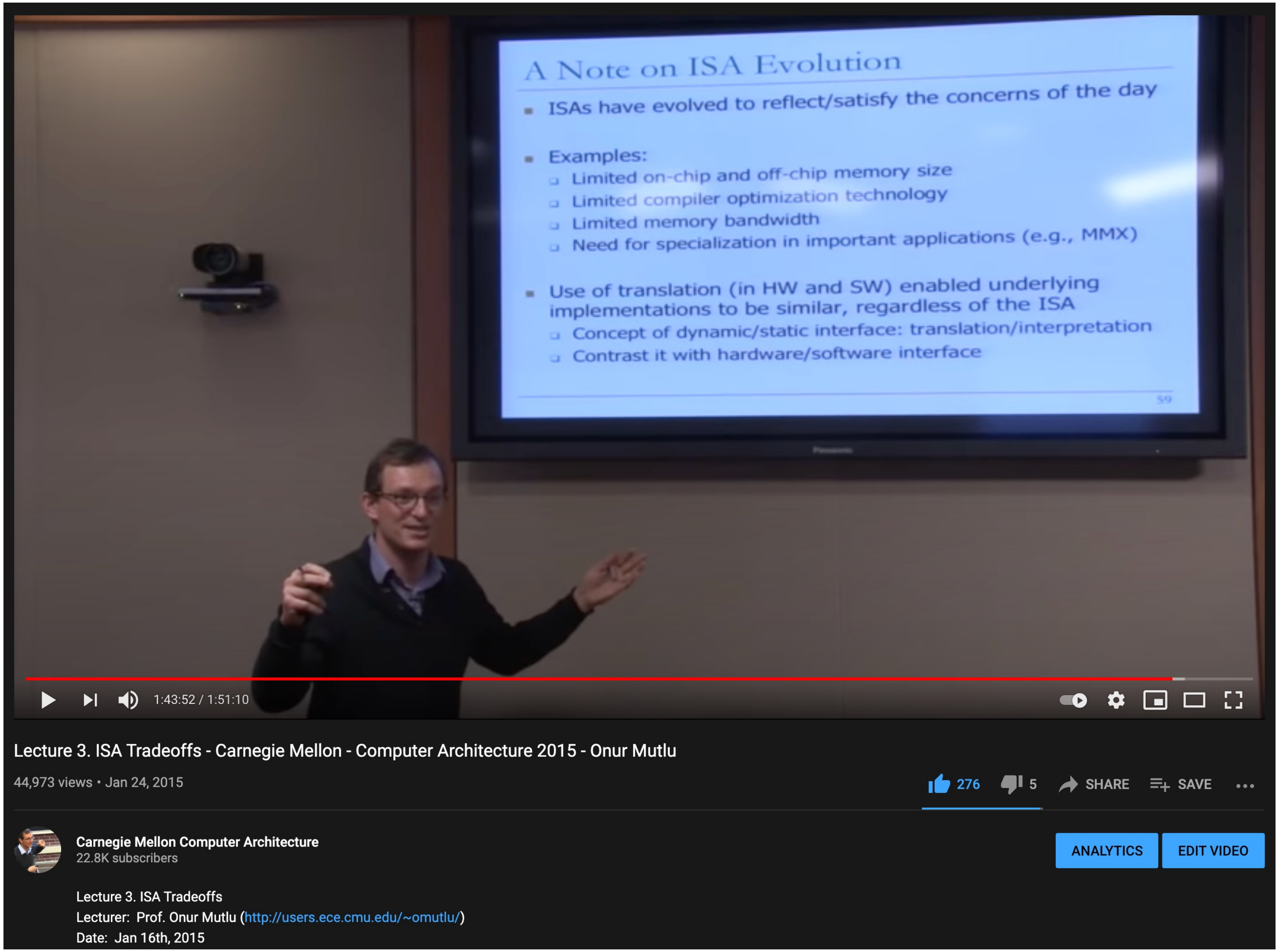


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, "The Technology Behind Crusoe Processors," Transmeta White Paper 2000.

https://www.wikiwand.com/en/Transmeta_Efficeon

There Is A Lot More to Cover on ISAs



A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
  - Limited on-chip and off-chip memory size
  - Limited compiler optimization technology
  - Limited memory bandwidth
  - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface: translation/interpretation
  - Contrast it with hardware/software interface

Lecture 3. ISA Tradeoffs - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

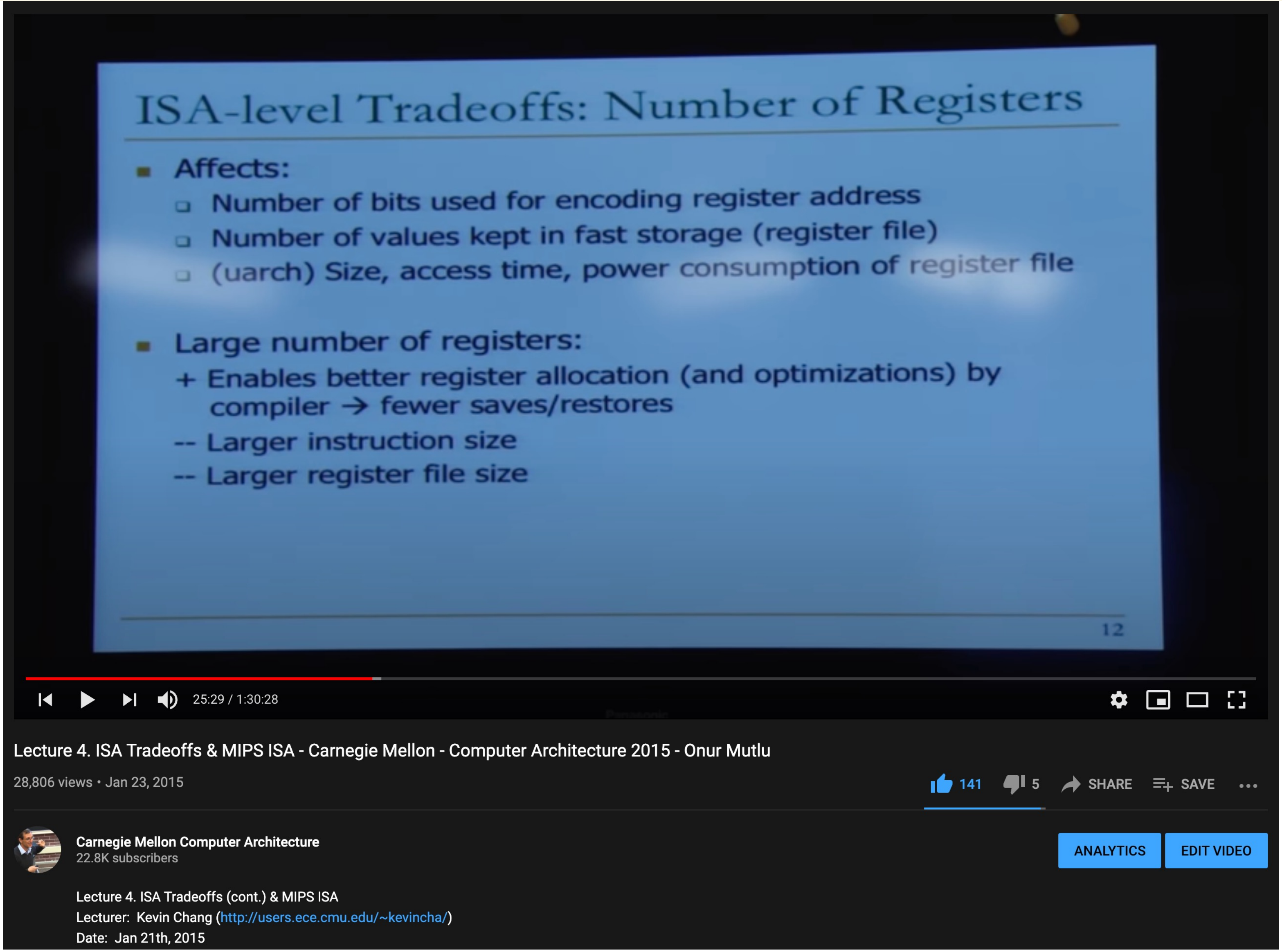
44,973 views • Jan 24, 2015

Carnegie Mellon Computer Architecture
22.8K subscribers

Lecture 3. ISA Tradeoffs
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: Jan 16th, 2015

https://www.youtube.com/onurmutlulectures

There Is A Lot More to Cover on ISAs



ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file
- Large number of registers:
  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  -- Larger instruction size
  -- Larger register file size

12

Lecture 4. ISA Tradeoffs & MIPS ISA - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

28,806 views • Jan 23, 2015

Carnegie Mellon Computer Architecture
22.8K subscribers

Lecture 4. ISA Tradeoffs (cont.) & MIPS ISA
Lecturer: Kevin Chang (http://users.ece.cmu.edu/~kevincha/)
Date: Jan 21th, 2015

https://www.youtube.com/onurmutlulectures

Detailed Lectures on ISAs & ISA Tradeoffs

- Computer Architecture, Spring 2015, Lecture 3
  - ISA Tradeoffs (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=QKdiZSfwg-g&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3

- Computer Architecture, Spring 2015, Lecture 4
  - ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4

- Computer Architecture, Spring 2015, Lecture 2
  - Fundamental Concepts and ISA (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2

ISA Design and Tradeoffs: More Critical Thinking

The Von Neumann Model/Architecture

Stored program

Sequential instruction processing

The von Neumann Model/Architecture

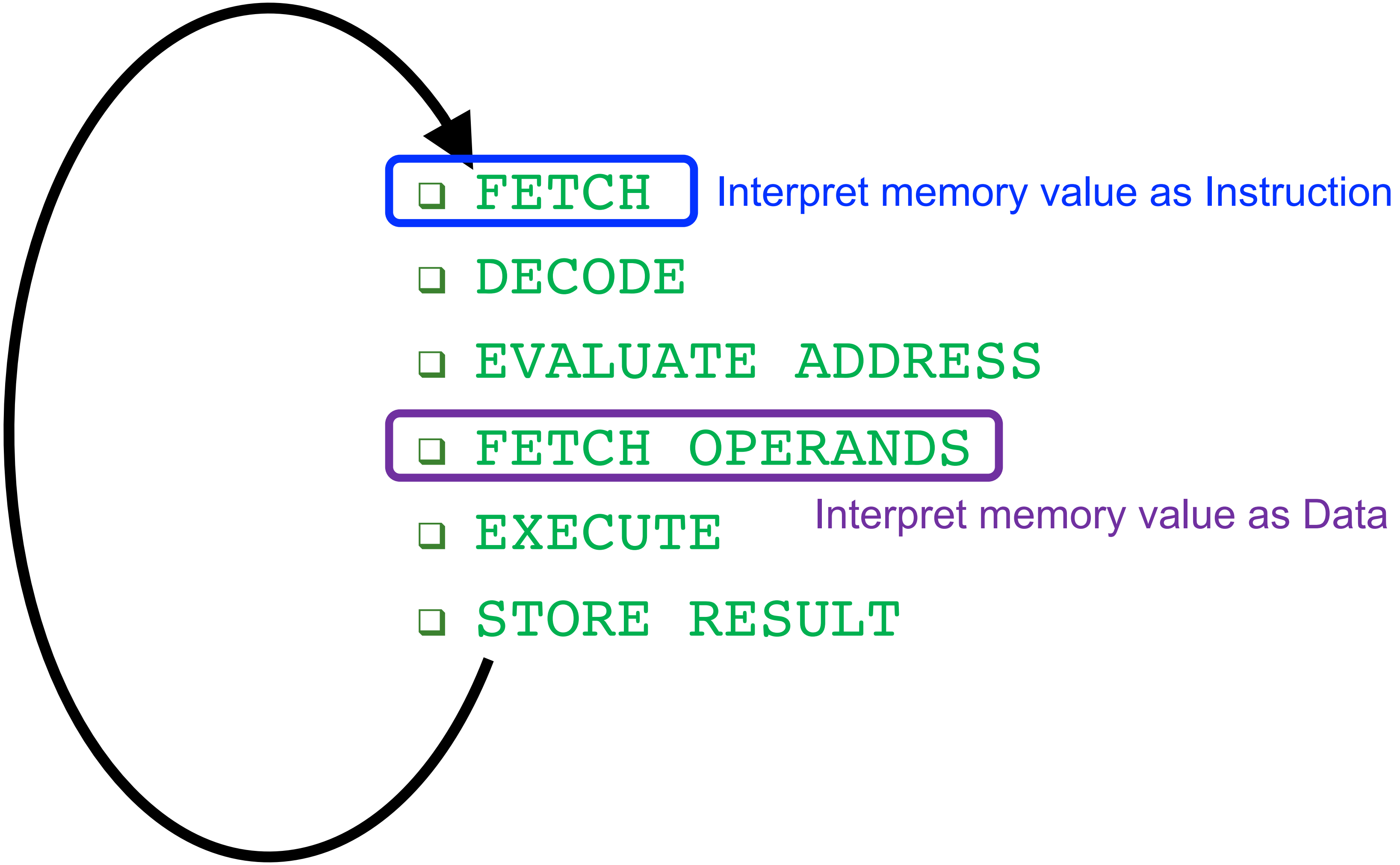
- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals

When is a value interpreted as an instruction?

- Sequential instruction processing

Recall: The Instruction Cycle



- FETCH

Interpret memory value as Instruction

- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS

Interpret memory value as Data

- EXECUTE
- STORE RESULT

Whether a value fetched from memory is interpreted as an instruction depends on **when** that value is **fetched** in the instruction processing cycle.

The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:

- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals

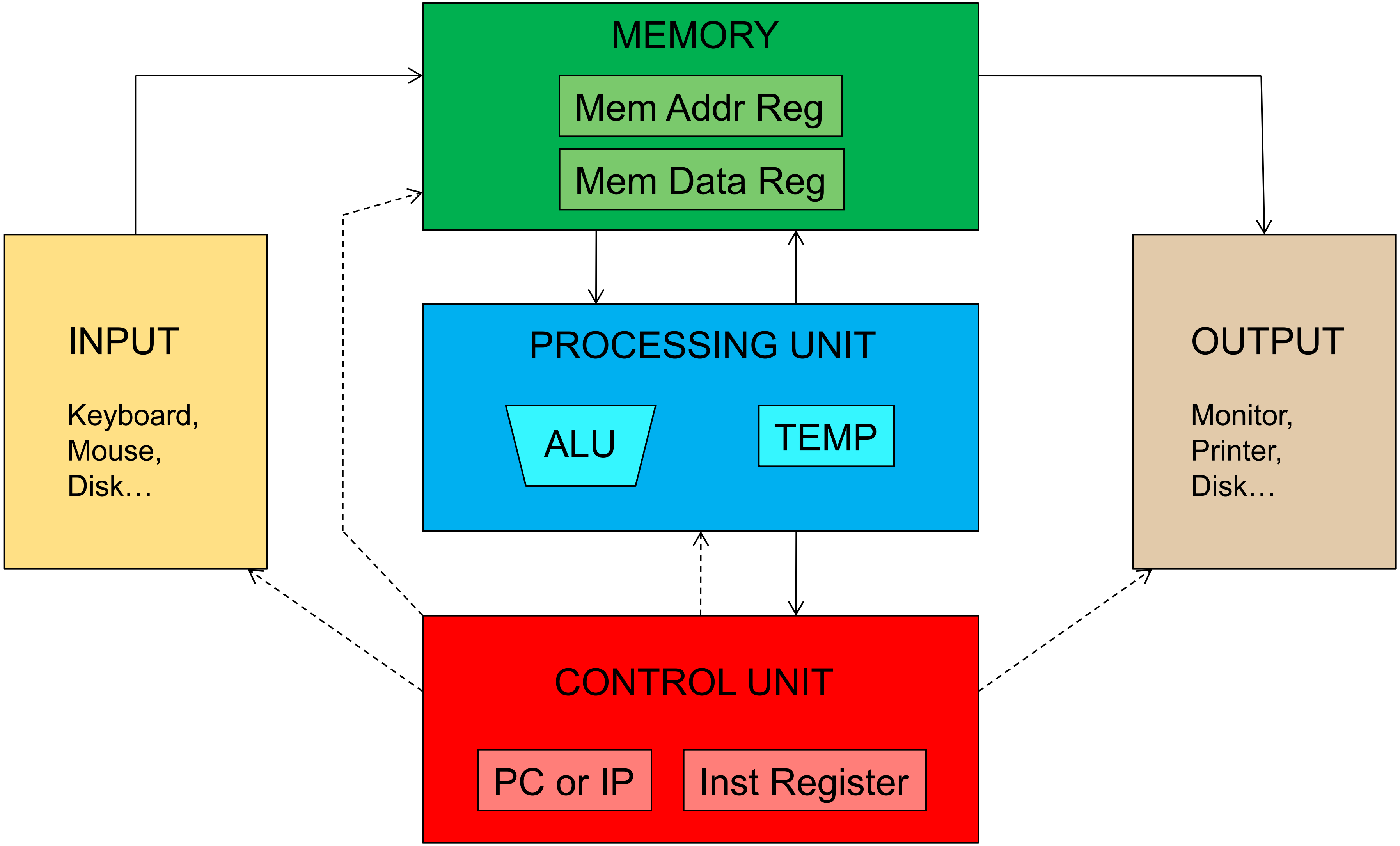
When is a value interpreted as an instruction?

- Sequential instruction processing
  - One instruction processed (fetched, executed, completed) at a time
  - Program counter (instruction pointer) identifies the current instruction
  - Program counter is advanced sequentially except for control transfer instructions

The von Neumann Model/Architecture

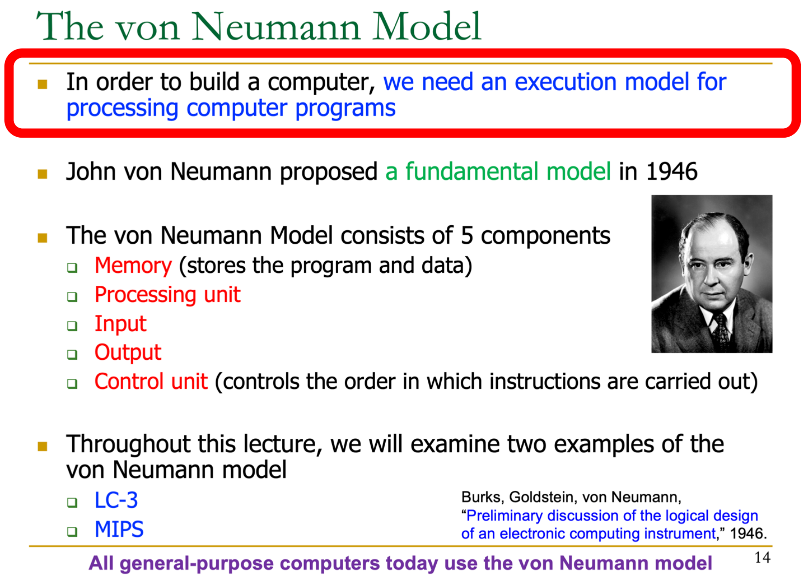
- Recommended reading
  - Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.
- Important reading
  - Patt and Patel book, Chapter 4, "The von Neumann Model"
- **Stored program**
- **Sequential instruction processing**

The Von Neumann Model (of a Computer)



The Von Neumann Model (of a Computer)

- Q: Is this the only way that a computer can process computer programs?



The von Neumann Model

- In order to build a computer, we need an execution model for processing computer programs
- John von Neumann proposed a fundamental model in 1946
- The von Neumann Model consists of 5 components
  - Memory (stores the program and data)
  - Processing unit
  - Input
  - Output
  - Control unit (controls the order in which instructions are carried out)
- Throughout this lecture, we will examine two examples of the von Neumann model
  - LC-3
  - MIPS

Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.

All general-purpose computers today use the von Neumann model

14

- A: No.
- Qualified Answer: No. But, it has been the dominant way
  - i.e., the dominant paradigm for computing
  - for N decades

Let's examine a **completely different model** for processing computer programs

The Dataflow Execution Model of a Computer

The Dataflow Model (of a Computer)

- Von Neumann model: An instruction is fetched and executed in control flow order
  - As specified by the program counter (instruction pointer)
  - Sequential unless explicit control flow instruction

- Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no program counter (instruction pointer)
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction can "fire" whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel

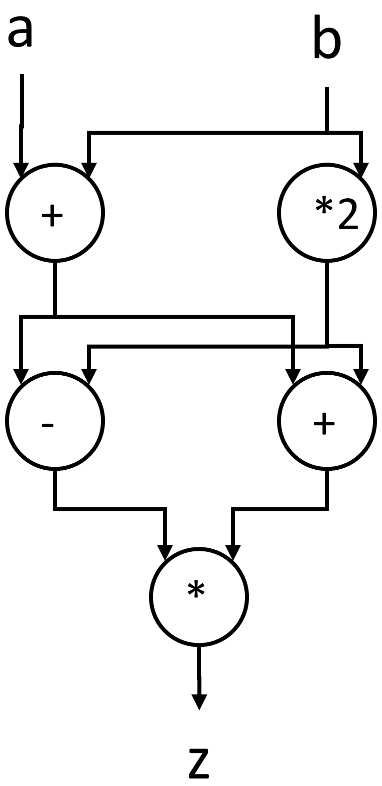
Von Neumann vs. Dataflow

- Consider a Von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v = a + b;
w = b * 2;
x = v - w
y = v + w
z = x * y
```

Sequential

a, b are the only inputs
z is the only output



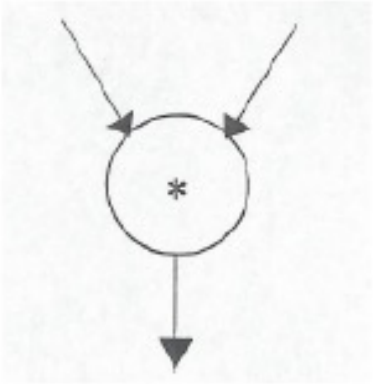
Dataflow

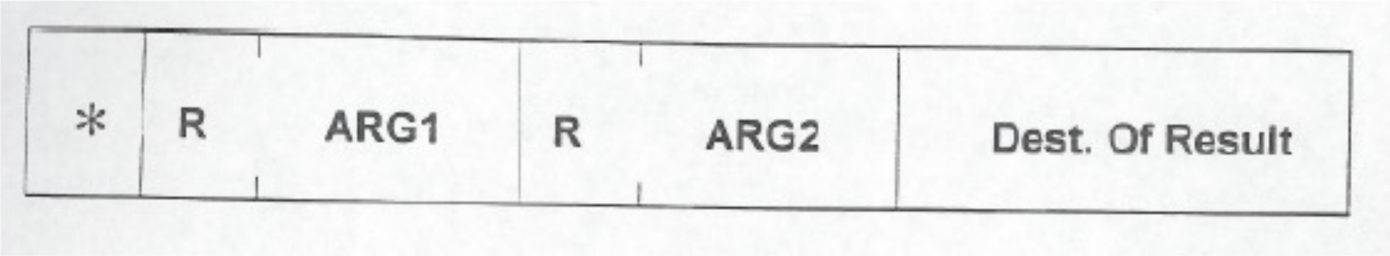
Which model is more natural to you as a programmer?

More on Dataflow

- In a dataflow machine, a program consists of dataflow nodes
  - A dataflow node fires (fetched and executed) when all it inputs are ready
    - i.e. when all inputs have tokens

- Dataflow node and its ISA representation

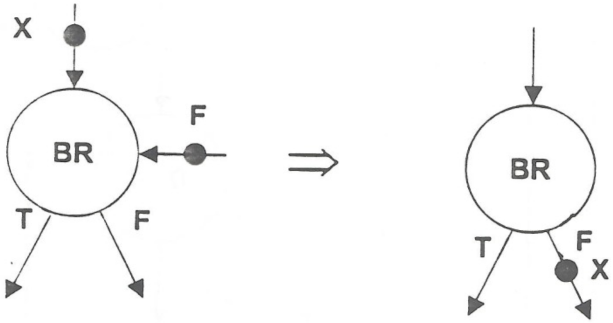




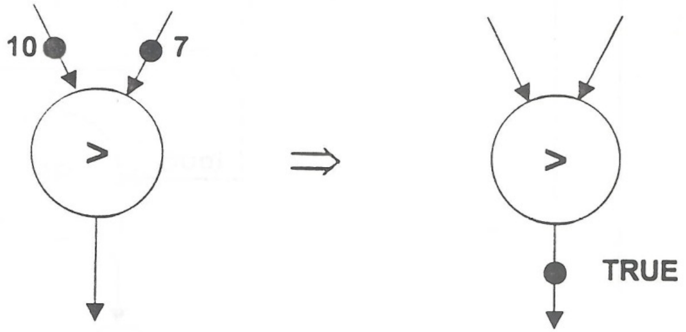
*	R	ARG1	R	ARG2	Dest. Of Result

Example Dataflow Nodes

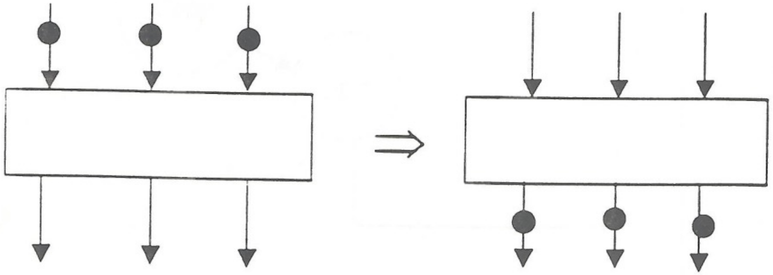
*Conditional



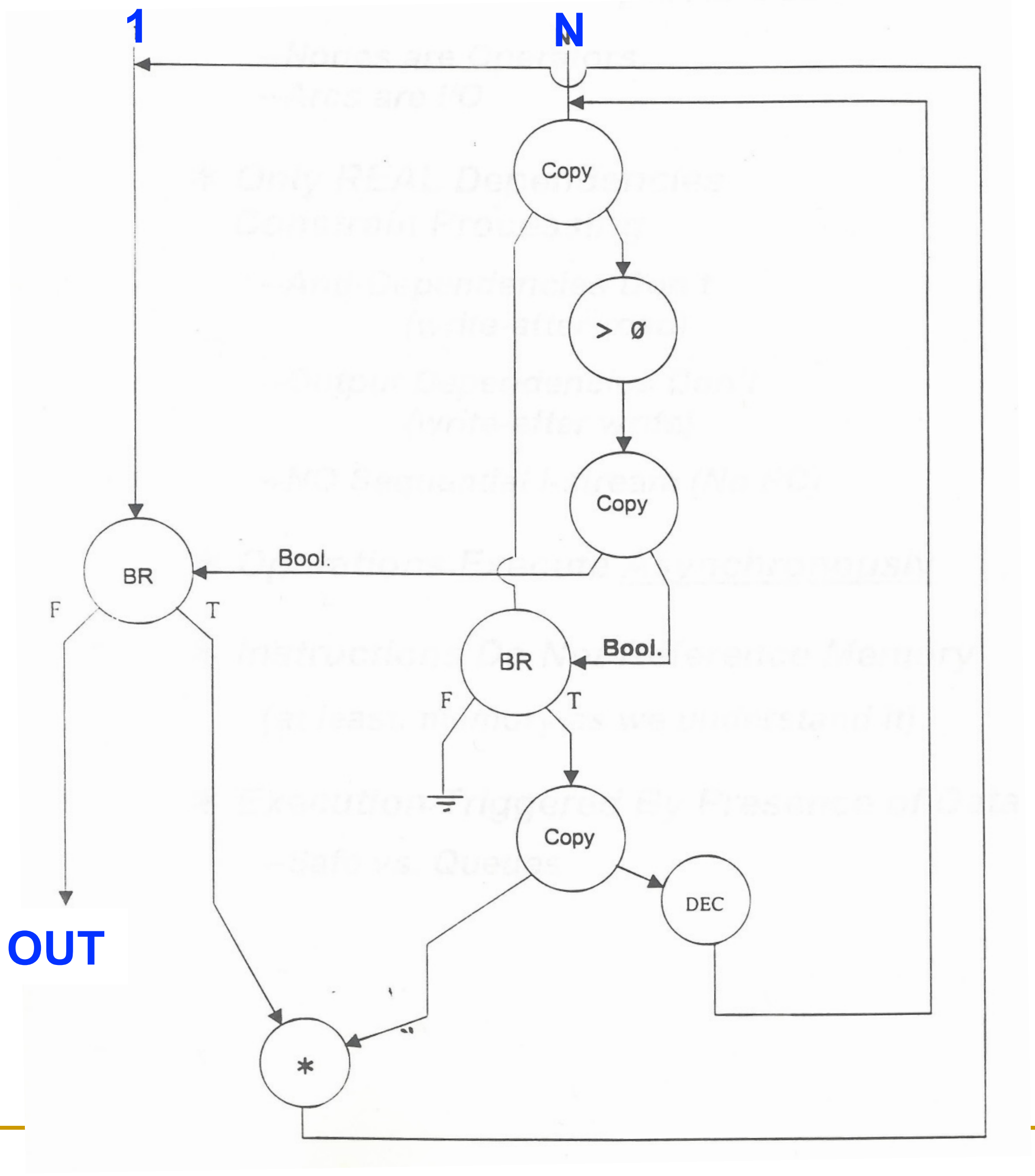
*Relational



*Barrier Synch



A Simple Example Dataflow Program



N is a non-negative integer

What is the value of OUT?

ISA-level Tradeoff: Program Counter

- Do we want a Program Counter (PC or IP) in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the PC points to it
    - PC automatically changes sequentially (except for control flow instructions) → sequential
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available → dataflow

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?

ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- ISA: Specifies how the **programmer sees** the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a dataflow execution order

- Microarchitecture: How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
    - Programmer should see the order specified by the ISA

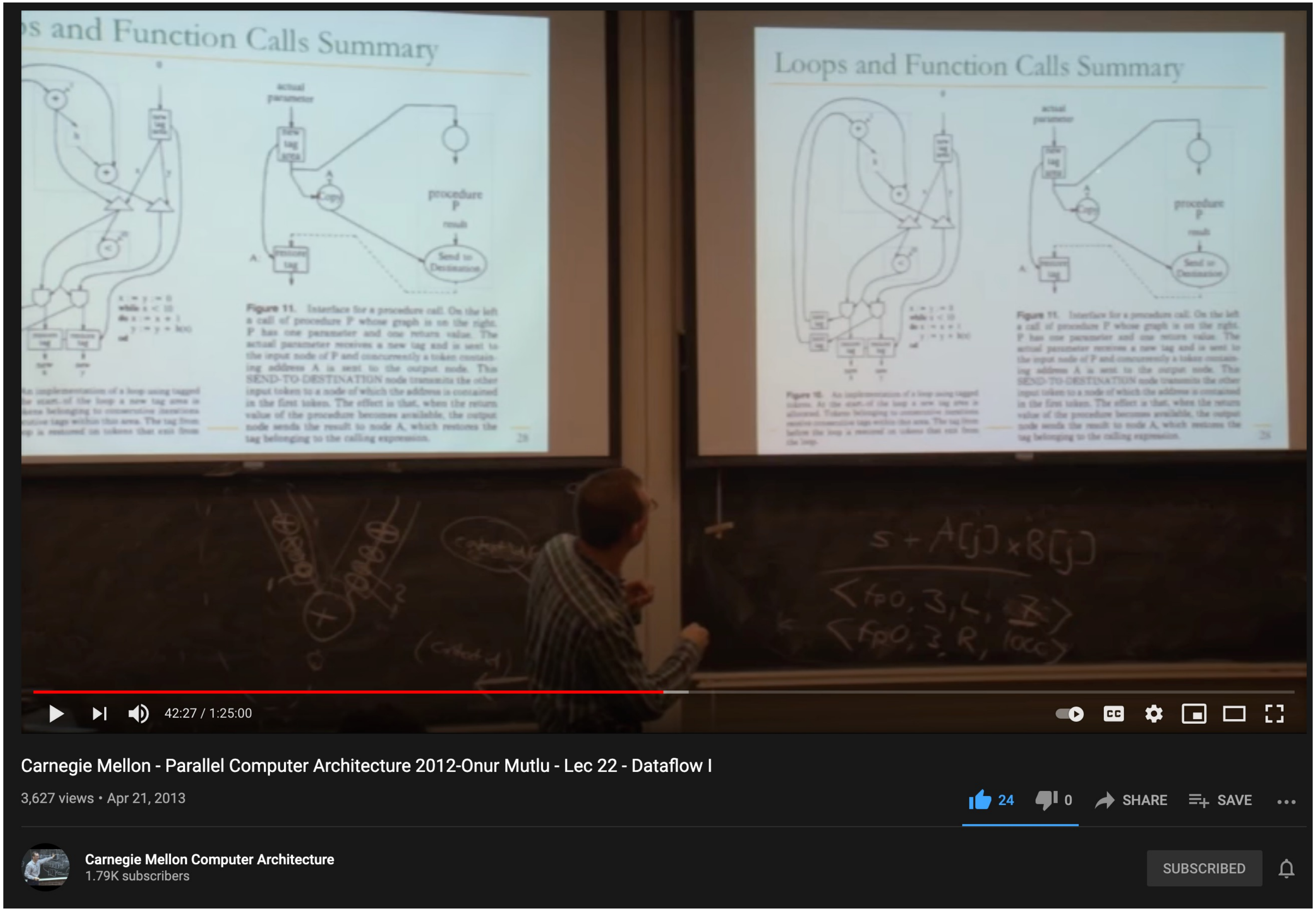
Let's Get Back to the von Neumann Model

- But, if you want to learn more about dataflow…

- Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
- A later lecture

- If you are really impatient:
  - http://www.youtube.com/watch?v=D2uue7izU2c
  - http://www.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.2.1-dataflow-part1.ppt

Lecture Video on Dataflow Architectures



Carnegie Mellon - Parallel Computer Architecture 2012-Onur Mutlu - Lec 22 - Dataflow I

3,627 views • Apr 21, 2013

Carnegie Mellon Computer Architecture
1.79K subscribers

http://www.youtube.com/watch?v=D2uue7izU2c

The von Neumann Model

- All major *instruction set architectures* today use this model
  - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, …

- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution: *Intel 80486 uarch*
  - Multiple instructions at a time: *Intel Pentium uarch*
  - Out-of-order execution: *Intel Pentium Pro uarch*
  - Separate instruction and data caches

- But, what happens underneath that is ***not* consistent** with the von Neumann model is ***not* exposed** to software
  - Difference between ISA and microarchitecture

What is Computer Architecture?

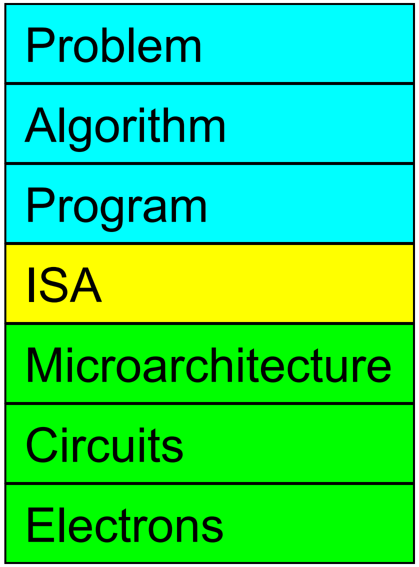
- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional (ISA-only) definition:** "The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior ***as distinct from*** the organization of the dataflow and controls, the logic design, and the physical implementation."

  Gene Amdahl, IBM Journal of R&D, April 1964

ISA vs. Microarchitecture

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs


Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture

ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for "acceleration"
  - Internals of the engine: implement "acceleration"

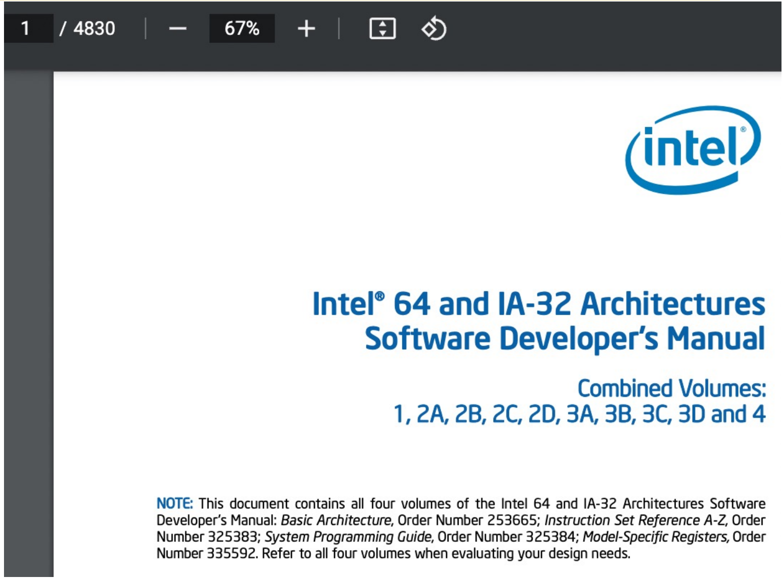
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture **(see H&H Chapter 5.2.1)**
  - x86 ISA has many implementations:
    - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, Ice Lake, Golden Cover, Sapphire Rapids, …, AMD K5, K7, K9, Bulldozer, BobCat, Ryzen X, …

- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) but many uarchs
  - *Why?*

https://www.vox.com/2015/7/1/8877583/two-foot-driving-pedal-error

ISA: What Does It Specify?

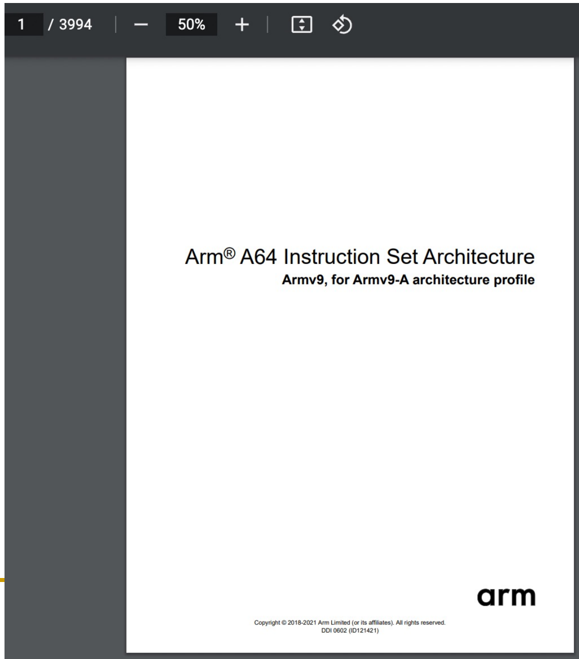
- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O: memory-mapped vs. instructions
- Task/thread Management
- Power & Thermal Management
- Multithreading & Multiprocessor support
- …



Intel® 64 and IA-32 Architectures
Software Developer's Manual

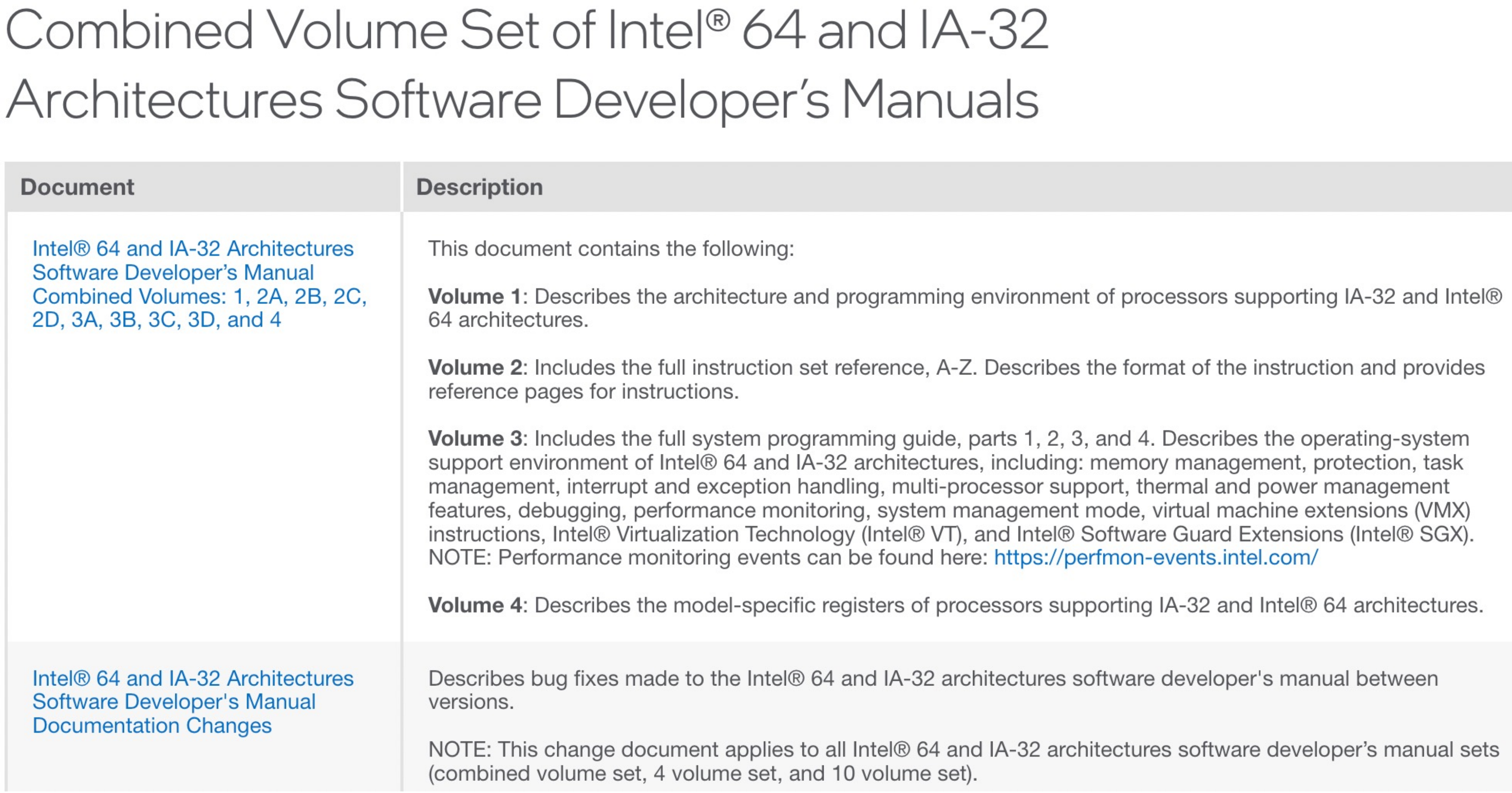
Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.



Arm® A64 Instruction Set Architecture
Armv9, for Armv9-A architecture profile

ISA Manuals: Some Good Bedtime Reading

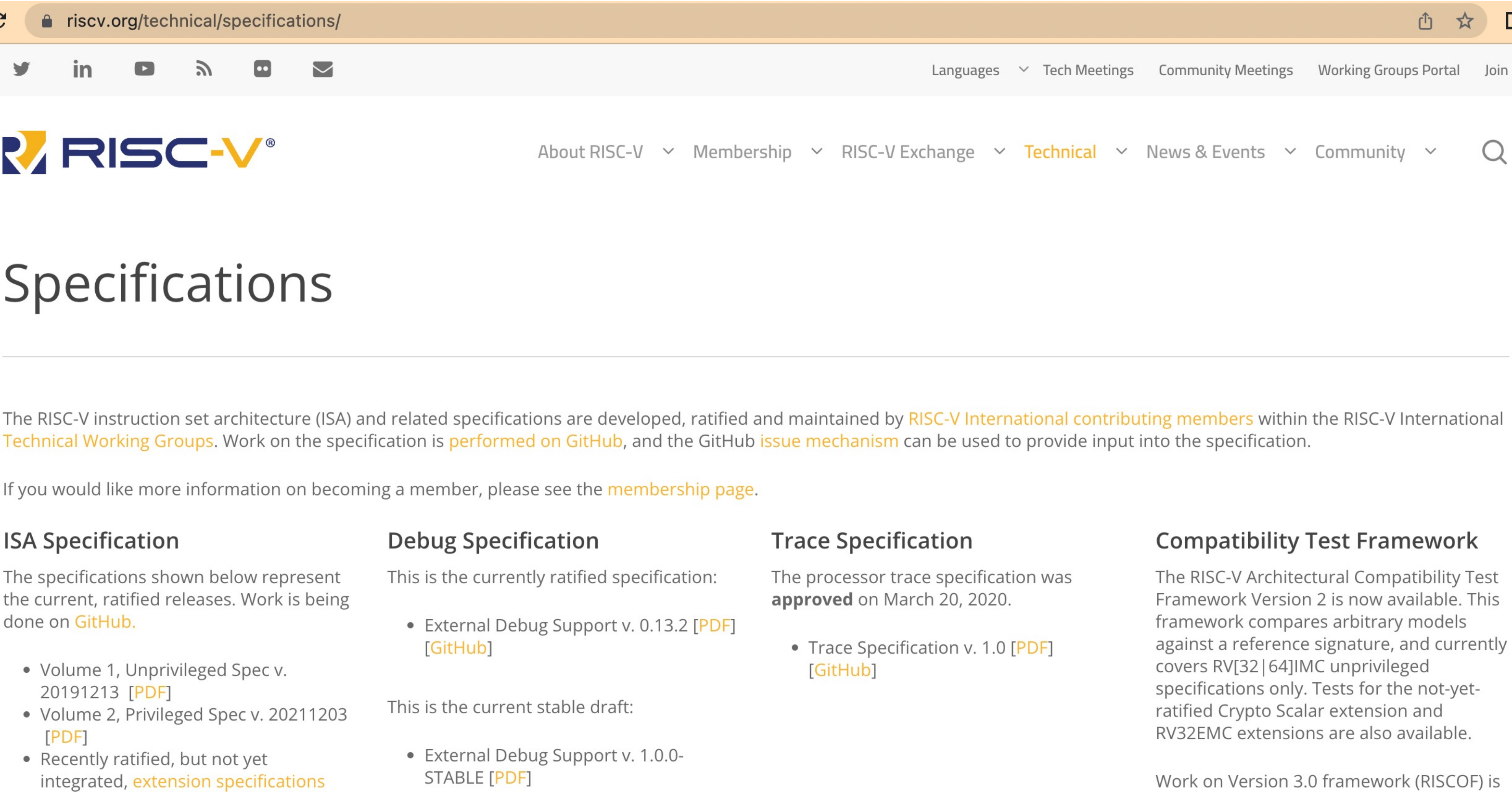


Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals

Document	Description
Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4	This document contains the following: **Volume 1**: Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures. **Volume 2**: Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions. **Volume 3**: Includes the full system programming guide, parts 1, 2, 3, and 4. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including: memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX). NOTE: Performance monitoring events can be found here: https://perfmon-events.intel.com/ **Volume 4**: Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures.
Intel® 64 and IA-32 Architectures Software Developer's Manual Documentation Changes	Describes bug fixes made to the Intel® 64 and IA-32 architectures software developer's manual between versions. NOTE: This change document applies to all Intel® 64 and IA-32 architectures software developer's manual sets (combined volume set, 4 volume set, and 10 volume set).

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

ISA Manuals: Some Good Bedtime Reading



Specifications

The RISC-V instruction set architecture (ISA) and related specifications are developed, ratified and maintained by RISC-V International contributing members within the RISC-V International Technical Working Groups. Work on the specification is performed on GitHub, and the GitHub issue mechanism can be used to provide input into the specification.

If you would like more information on becoming a member, please see the membership page.

ISA Specification

The specifications shown below represent the current, ratified releases. Work is being done on GitHub.

- Volume 1, Unprivileged Spec v. 20191213 [PDF]
- Volume 2, Privileged Spec v. 20211203 [PDF]
- Recently ratified, but not yet integrated, extension specifications

Debug Specification

This is the currently ratified specification:

- External Debug Support v. 0.13.2 [PDF] [GitHub]

This is the current stable draft:

- External Debug Support v. 1.0.0-STABLE [PDF]

Trace Specification

The processor trace specification was **approved** on March 20, 2020.

- Trace Specification v. 1.0 [PDF] [GitHub]

Compatibility Test Framework

The RISC-V Architectural Compatibility Test Framework Version 2 is now available. This framework compares arbitrary models against a reference signature, and currently covers RV[32|64]IMC unprivileged specifications only. Tests for the not-yet-ratified Crypto Scalar extension and RV32EMC extensions are also available.

Work on Version 3.0 framework (RISCOF) is

https://riscv.org/technical/specifications/

Microarchitecture

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?

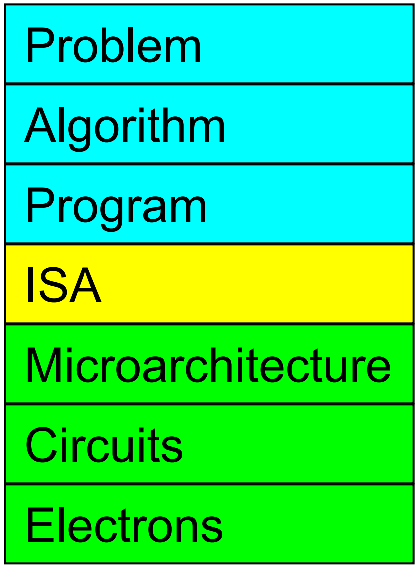
Property of ISA vs. Uarch?

- ADD instruction's opcode
- Type of adder used in the ALU (Bit-serial vs. Ripple-carry)
- Number of general purpose registers
- Number of cycles to execute the MUL instruction
- Number of ports to the register file
- Whether or not the machine employs pipelined instruction execution
- Program counter

- Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals

Design Point

- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Example considerations:
  - Cost
  - Performance
  - Maximum power consumption, thermal
  - Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market
  - Security, safety, predictability, …

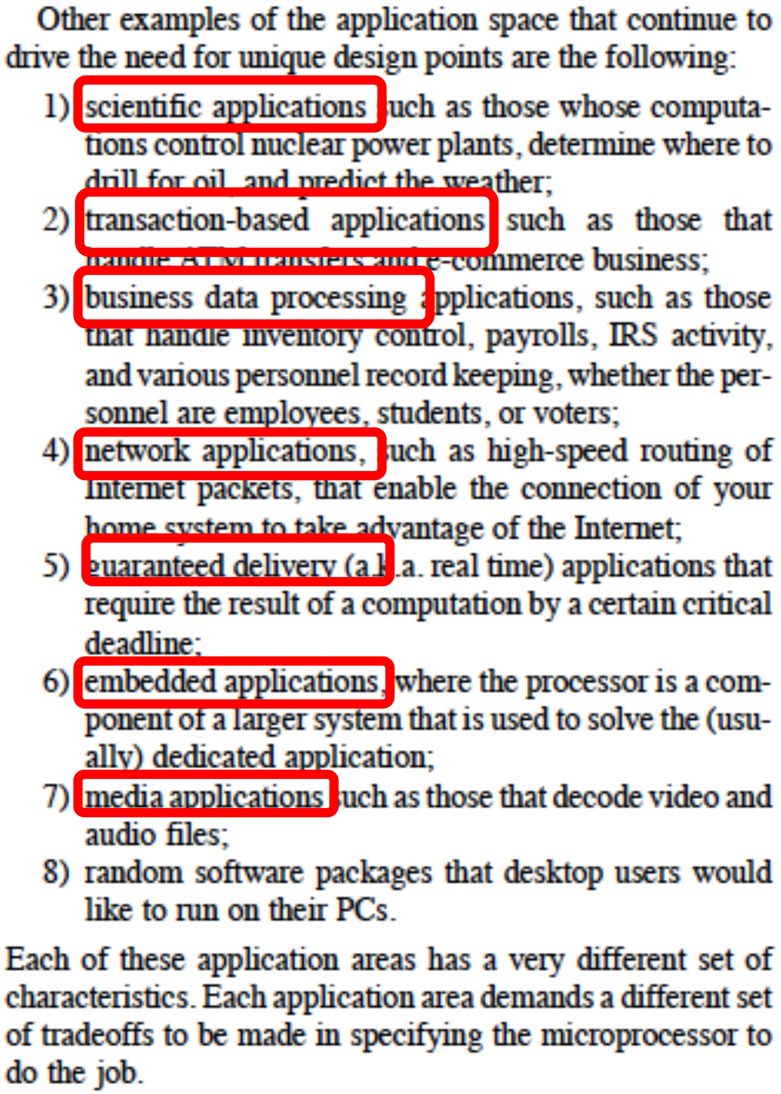


Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

- Design point is determined by the "Problem" space (application space), the intended users/*market*

Application Space

Dream, and they will appear…



Other examples of the application space that continue to drive the need for unique design points are the following:

1) scientific applications such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
2) transaction-based applications such as those that handle ATM transfers and e-commerce business;
3) business data processing applications, such as those that handle inventory control, payrolls, IRS activity, and various personnel record keeping, whether the personnel are employees, students, or voters;
4) network applications, such as high-speed routing of Internet packets, that enable the connection of your home system to take advantage of the Internet;
5) guaranteed delivery (a.k.a. real time) applications that require the result of a computation by a certain critical deadline;
6) embedded applications, where the processor is a component of a larger system that is used to solve the (usually) dedicated application;
7) media applications such as those that decode video and audio files;
8) random software packages that desktop users would like to run on their PCs.

Each of these application areas has a very different set of characteristics. Each application area demands a different set of tradeoffs to be made in specifying the microprocessor to do the job.

Patt, "Requirements, bottlenecks, and good fortune: agents for microprocessor evolution," Proc. of the IEEE 2001.

Many other workloads:

Genome analysis

Machine learning

Robotics

Web search

Graph analytics

…

Increasingly Demanding Applications

Dream

and, they will come

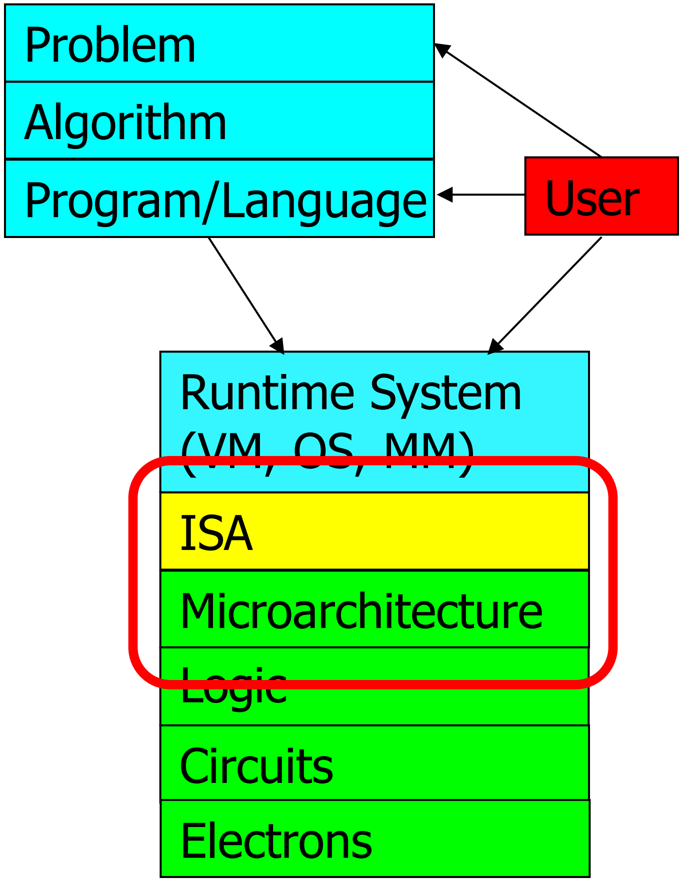
As applications push boundaries, computing platforms will become increasingly strained.

Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why **art**?*

Why Is It (Somewhat) Art?

New demands from the top (Look Up)

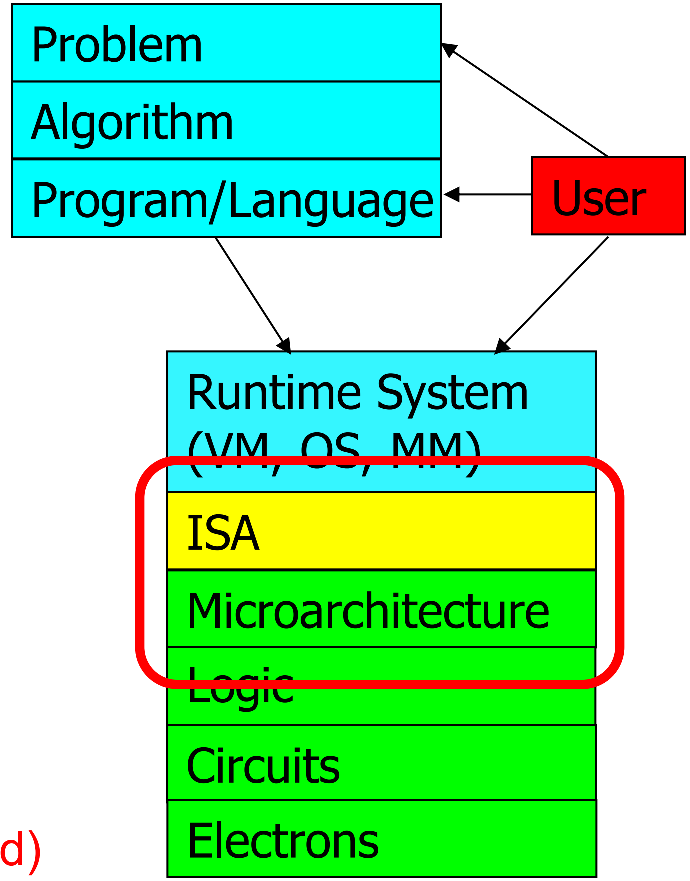


New demands and personalities of users (Look Up)

New issues and capabilities at the bottom (Look Down)

- We do not (fully) know the future (applications, users, market)

Why Is It (Somewhat) Art?



Changing demands at the top (Look Up and Forward)

Changing demands and personalities of users (Look Up and Forward)

Changing issues and capabilities at the bottom (Look Down and Forward)

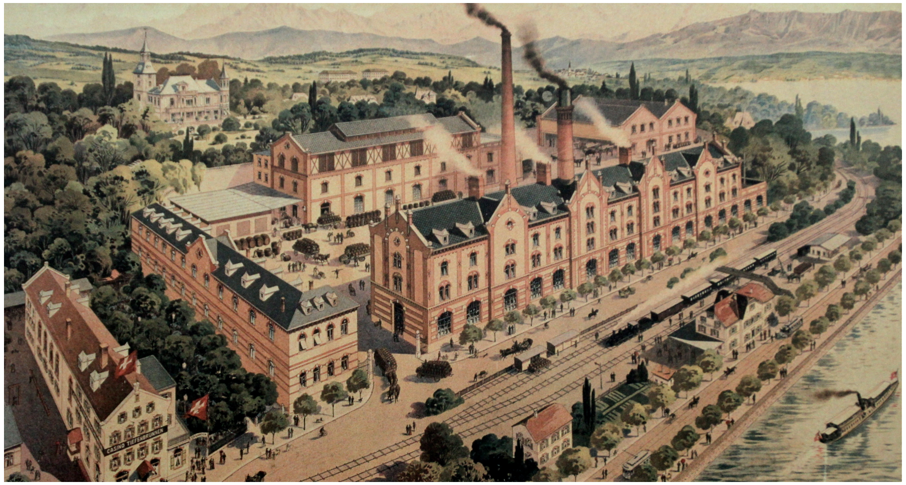
- And, the future is not constant (it changes)!

Analogue from Macro-Architecture

- Future is not constant in macro-architecture, either
- Example: Can a mill be later used as a theater + restaurant + conference room?

Mühle Tiefenbrunnen in Zurich

- Originally built as a brewery in 1889
  - part of it was converted into a mill in 1913
  - and the other part into a cold store
- Today is a center for a variety of activities: theater, conferences, restaurants, shops, museum…



Brewery in 1900

http://www.muehle-tiefenbrunnen.ch/

Another Example in Zurich (I)

Photo credit: Prof. Can Alkan

Another Example in Zurich (II)

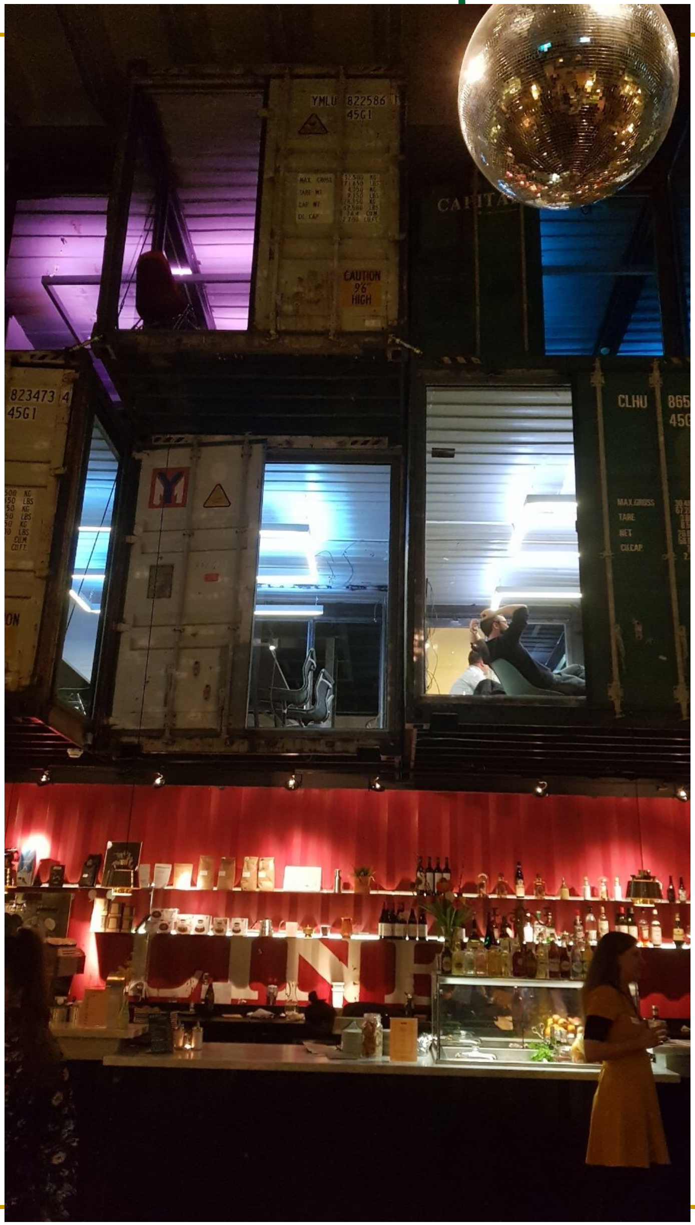
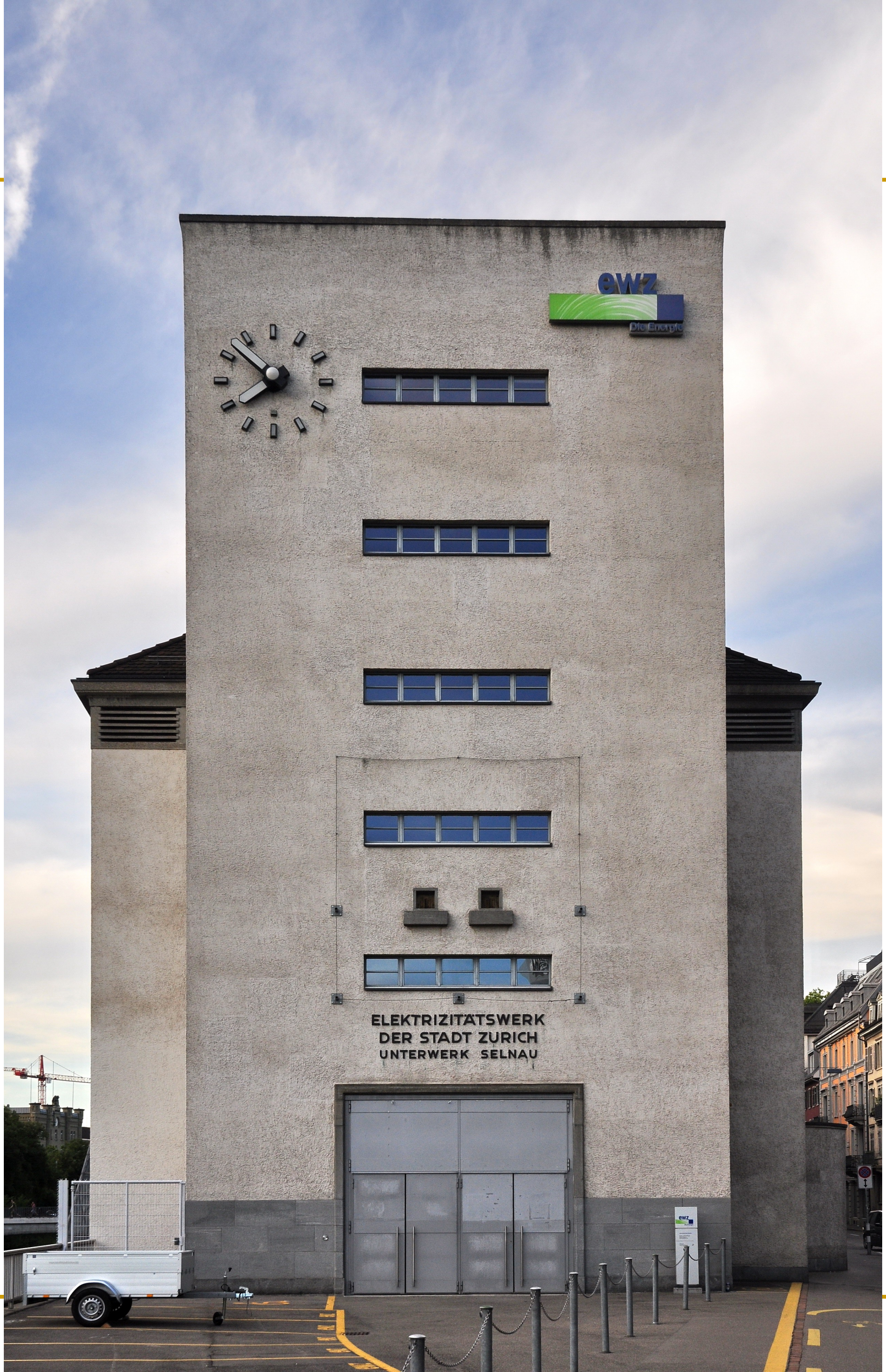
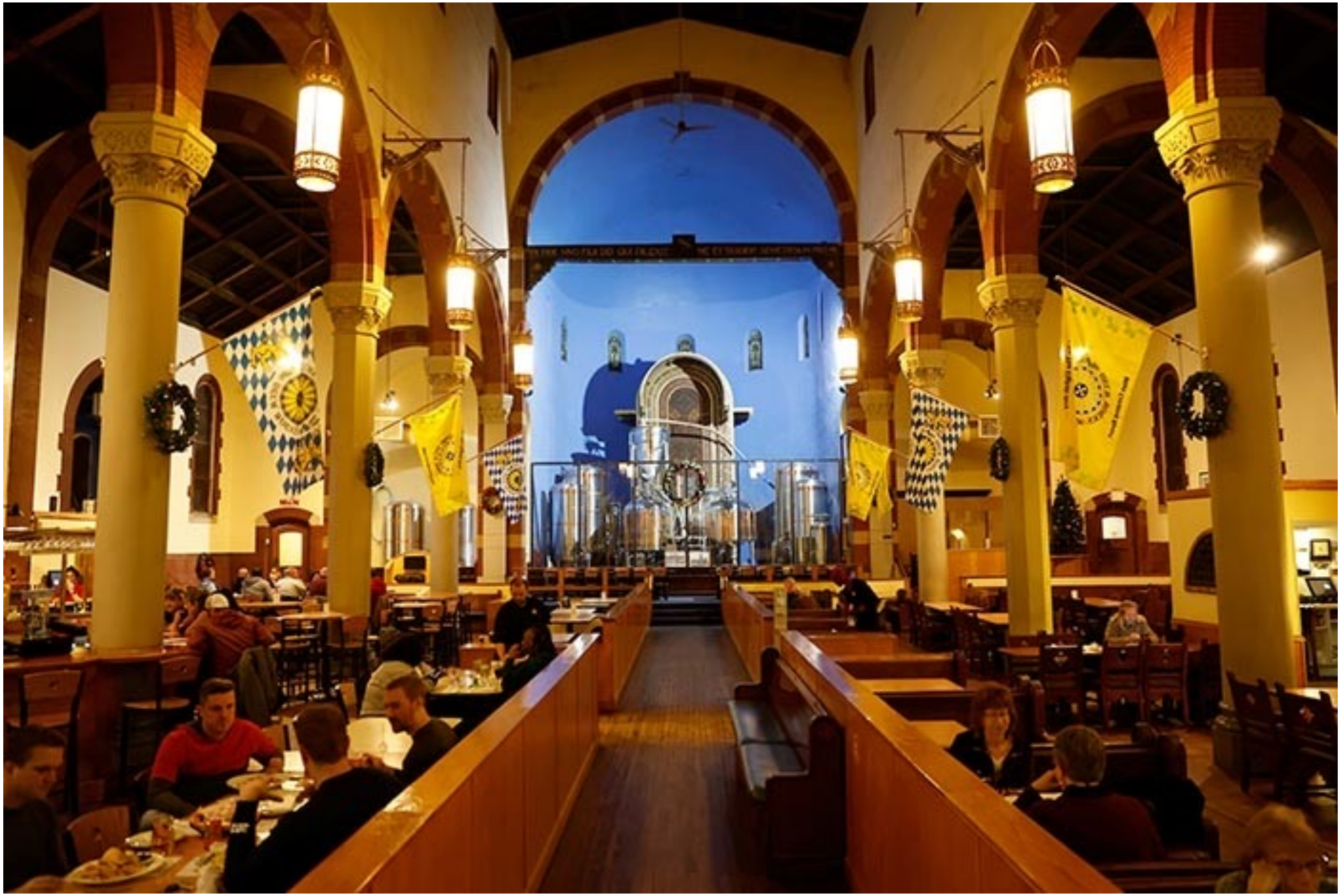


Photo credit: Prof. Can Alkan



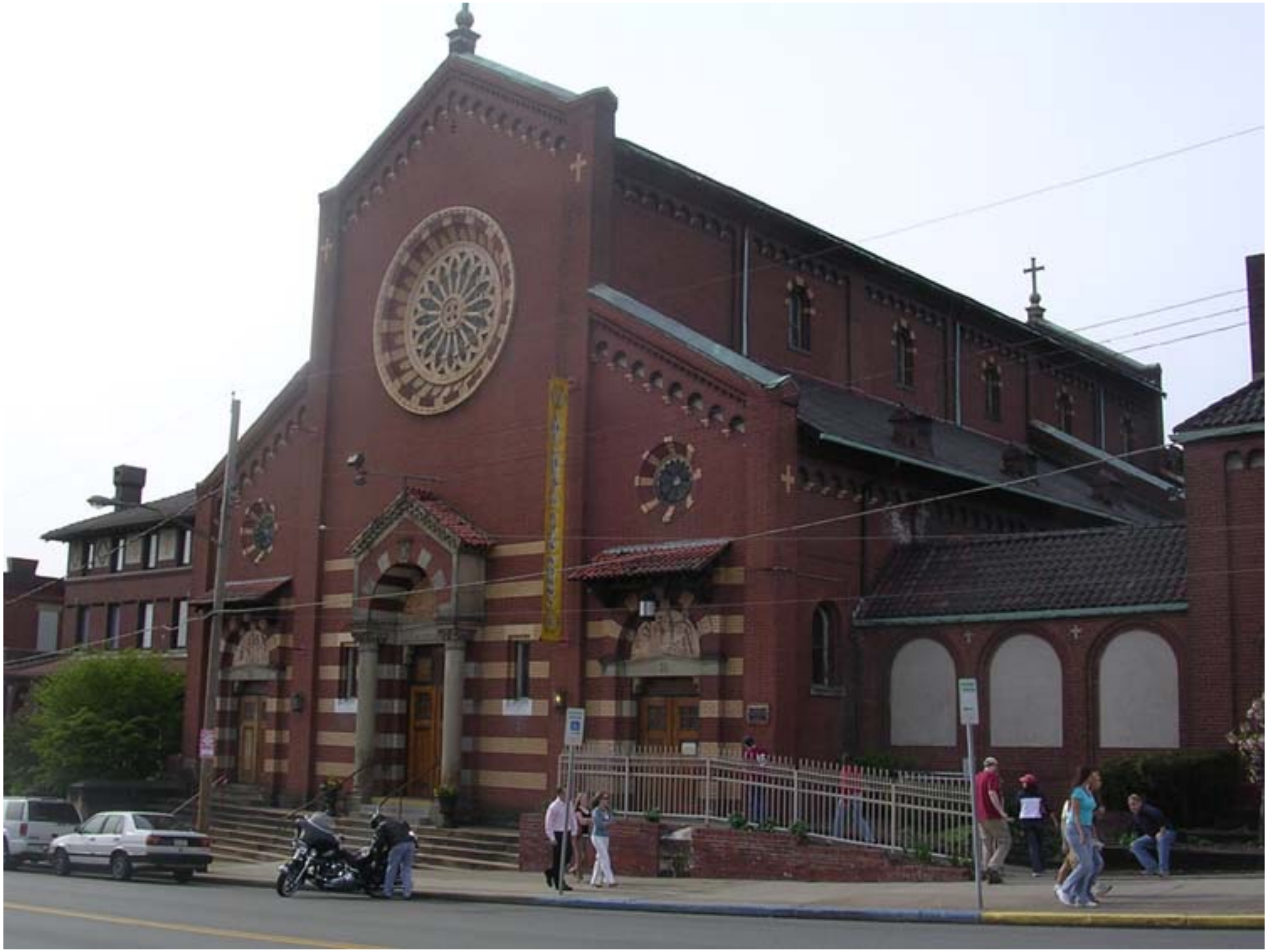
By Roland zh (Own work) [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Yet Another Example from Pittsburgh (I)



https://www.pghcitypaper.com/pittsburgh/a-list-of-pittsburgh-area-churches-born-again-with-new-purposes/Content?oid=20743835

Yet Another Example from Pittsburgh (II)



https://en.wikipedia.org/wiki/The_Church_Brew_Works#/media/File:The_Church_Brew_Works.jpg

Implementing the ISA: Microarchitecture Basics

Now That We Know What an ISA Is…

- How do we implement it?

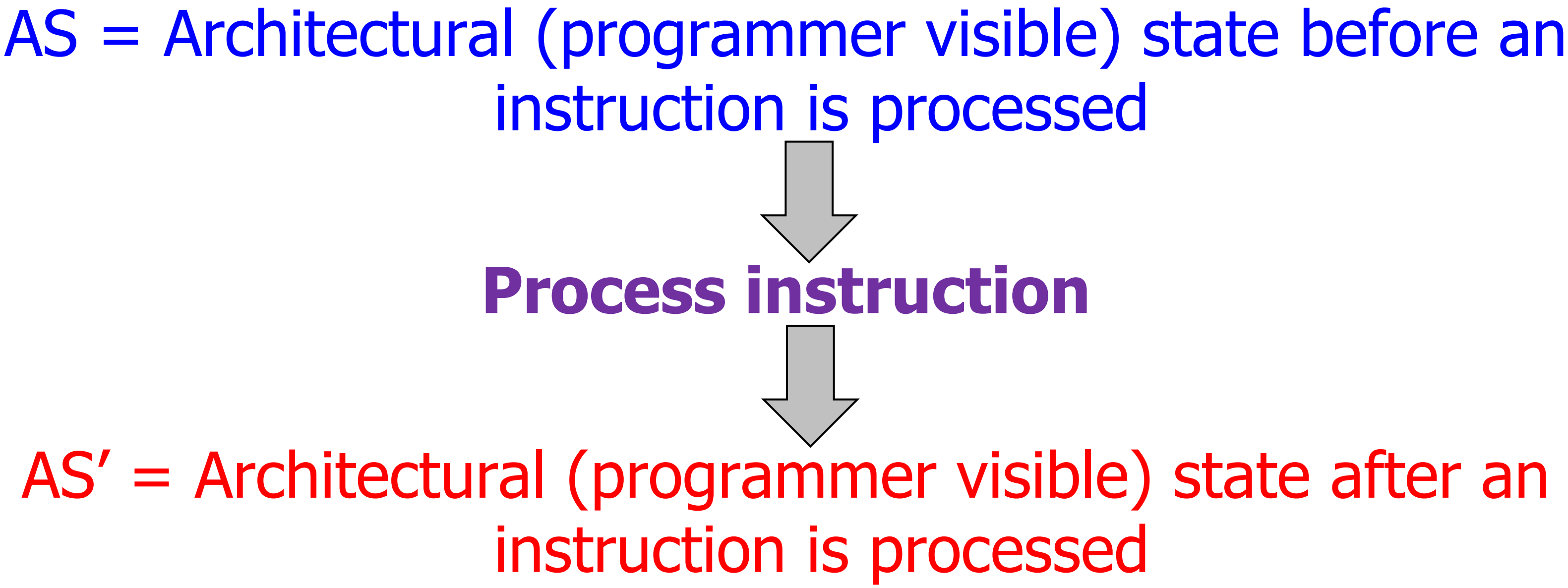
- i.e., **how do we design a system that obeys the hardware/software interface?**

- Aside: “System” can be solely hardware or a combination of hardware and software
  - Recall the “Translation of ISAs”
  - An **ISA** can be converted (by software or hardware) into an **implementation ISA**

- We will assume “completely hardware” implementation for most lectures

How Does a Machine Process Instructions?

- What does processing an instruction mean?
- We will assume the von Neumann model (for now)



AS = Architectural (programmer visible) state before an instruction is processed

Process instruction

AS' = Architectural (programmer visible) state after an instruction is processed

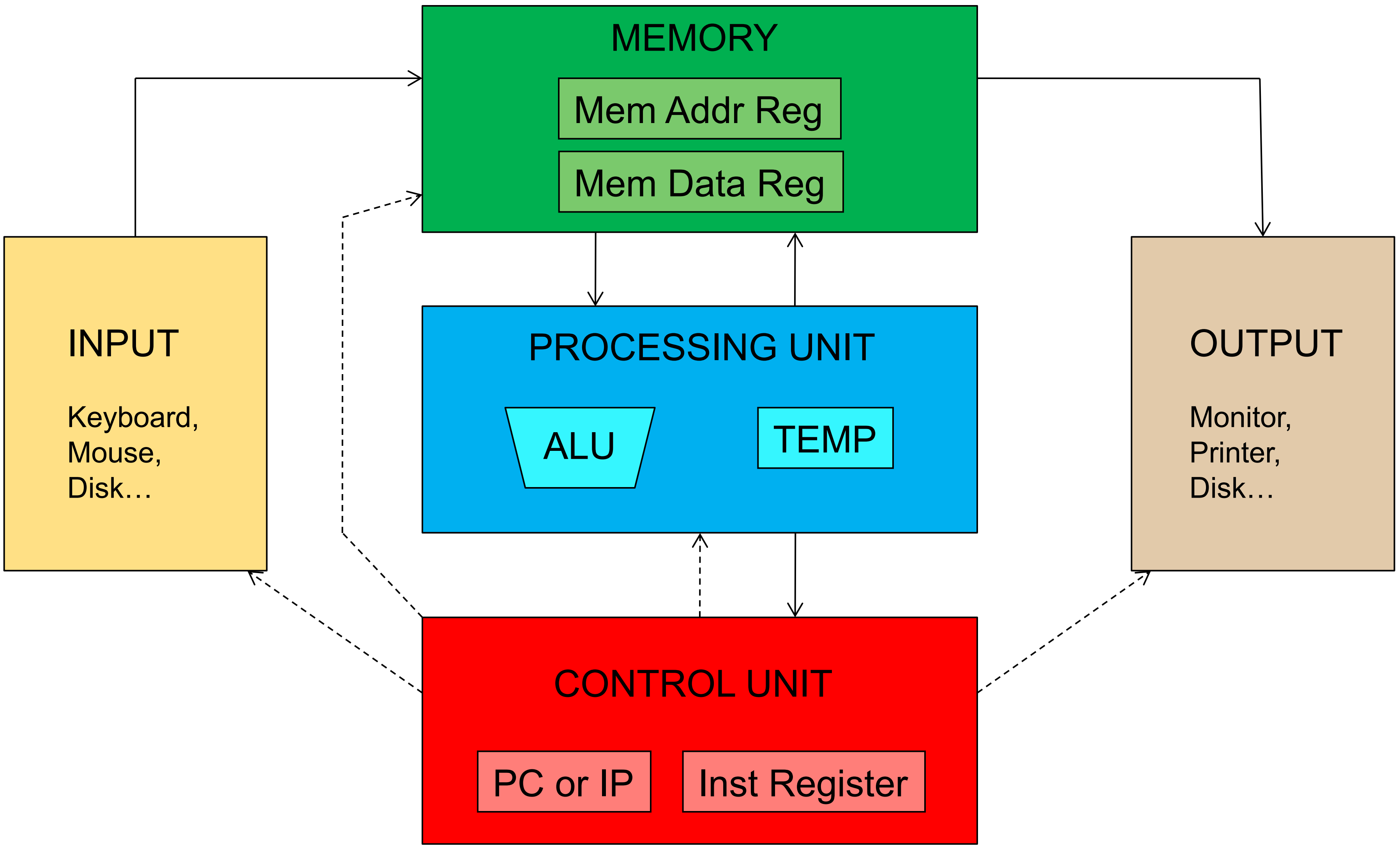
- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The Von Neumann Model/Architecture

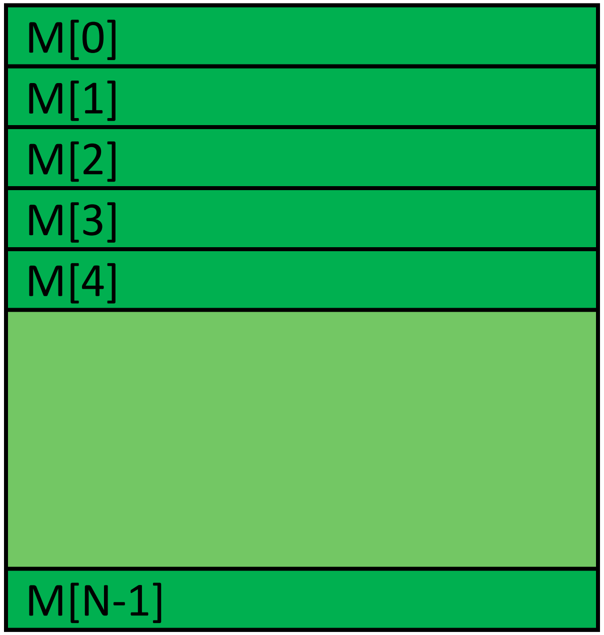
Stored program

Sequential instruction processing

Recall: The Von Neumann Model



Recall: Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address

Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Program Counter
memory address
of the current (or next) instruction

Instructions (and programs) specify how to transform the values of programmer visible state

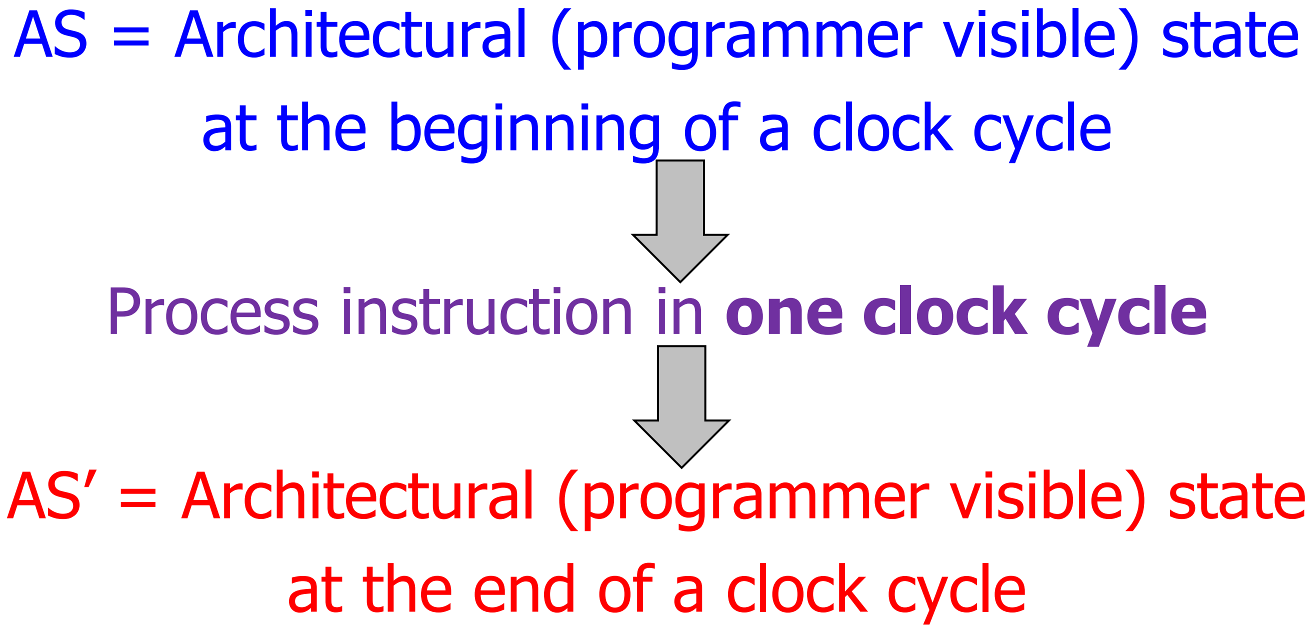
The "Process Instruction" Step

- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an **abstract finite state machine** where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
    - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS'
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Choice 1: AS → AS' (transform AS to AS' in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*



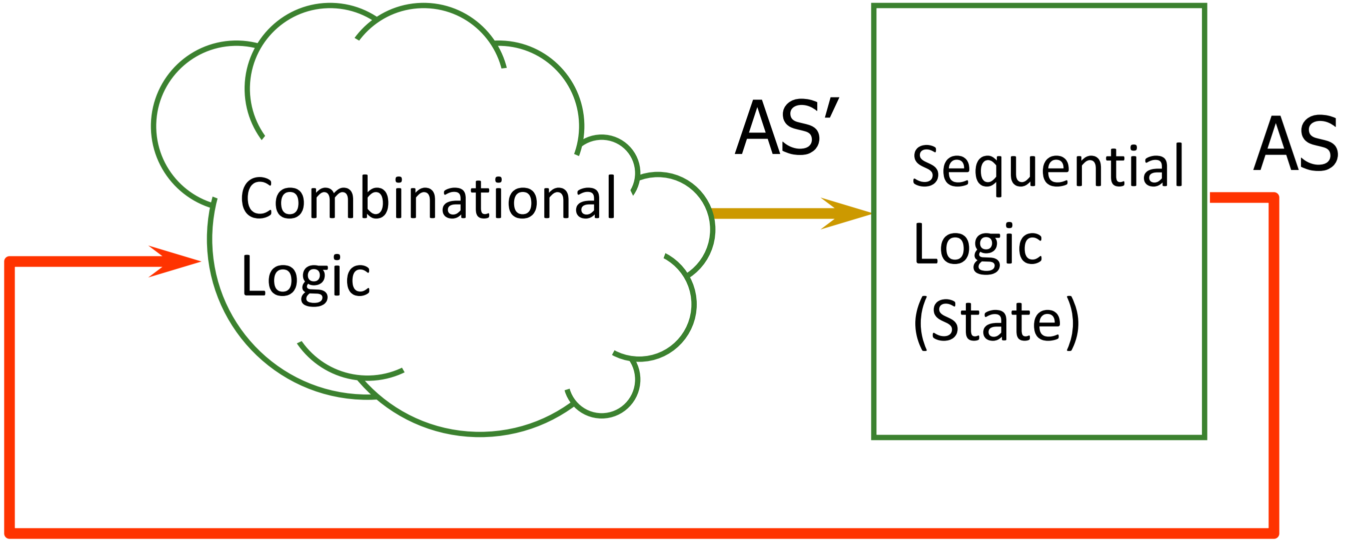
AS = Architectural (programmer visible) state
at the beginning of a clock cycle

Process instruction in **one clock cycle**

AS' = Architectural (programmer visible) state
at the end of a clock cycle

A Very Basic Instruction Processing Engine

- Single-cycle machine



- What is the *clock cycle time* determined by?
- What is the *critical path* (i.e., longest delay path) of the combinational logic determined by?

AS: Architectural State

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

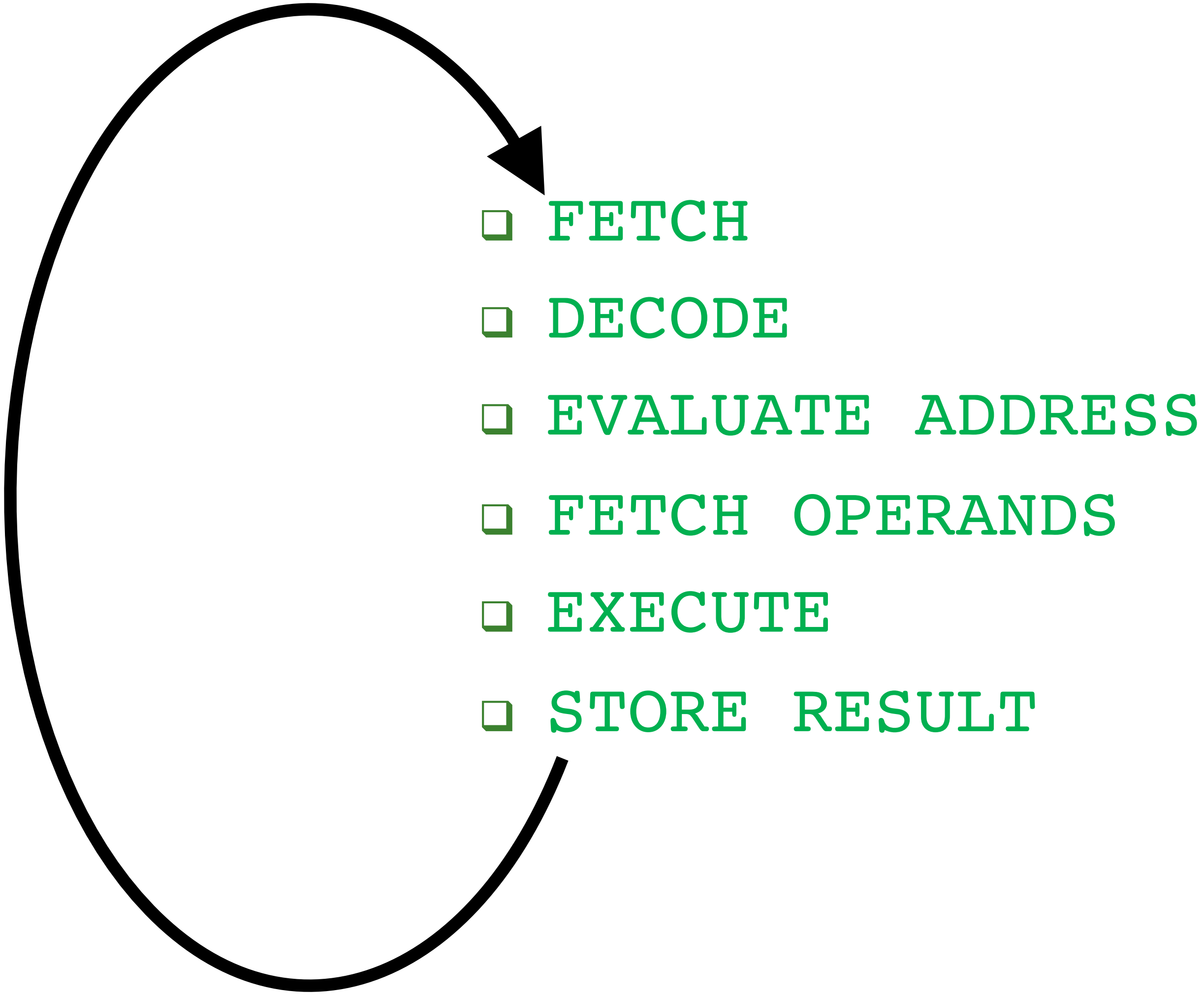
Instruction Processing “Cycle”

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six steps:

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

- Not all instructions require all six steps (see P&P Ch. 4)

Recall: The Instruction Processing “Cycle”



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

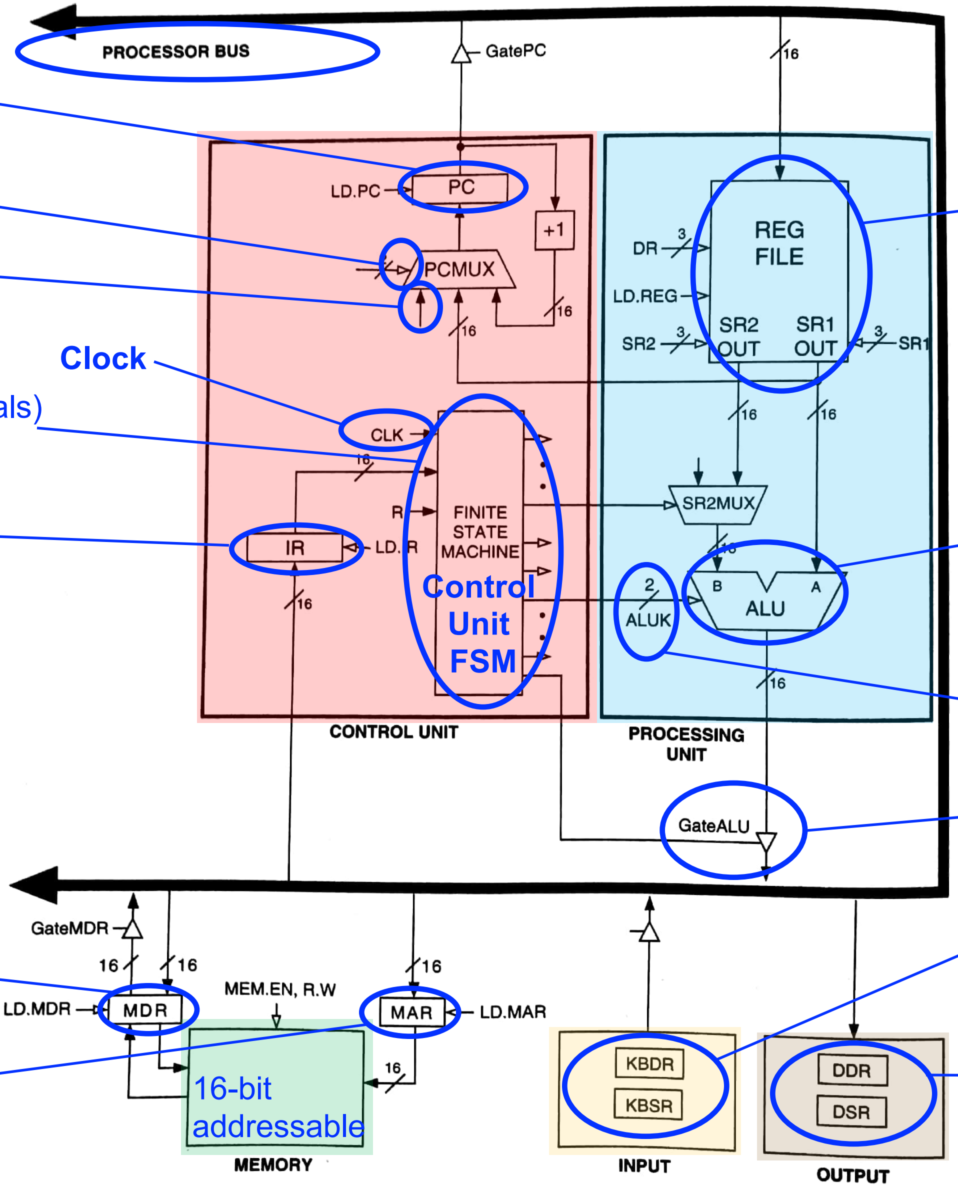
- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock cycles to complete

Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data’ (AS’)
- This transformation is done by functional units
  - Units that “operate” on data
- These units need to be told what to do to the data

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - **functional units** that operate on data
    - **hardware structures** (e.g., wires, muxes, decoders, tri-state bufs) that enable the flow of data into the functional units and registers
    - **storage units** that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

Recall: LC-3: A von Neumann Machine



Program Counter

Control signals

Data

Finite State Machine (for Generating Control Signals)

Clock

Instruction Register

Control Unit FSM

8 General Purpose Registers (GPR)

ALU: 2 inputs, 1 output

ALU operation

GateALU

Memory Data Register

Memory Address Register

16-bit addressable

Keyboard
KBDR (data), KBSR (status)

Monitor
DDR (data), DSR (status)

Figure 4.3 The LC-3 as an example of the von Neumann model

Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on
  - Everything related to an instruction happens in one clock cycle (serialized processing)

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the current cycle
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism)

- See P&P Appendix C for more (microprogrammed multi-cycle microarchitecture)

Many Ways of Datapath and Control Design

- There are many ways of designing the datapath and control logic

- Example ways
  - Single-cycle, multi-cycle, pipelined datapath and control
  - Single-bus vs. multi-bus datapaths
  - Hardwired/combinational vs. microcoded/microprogrammed control
    - Control signals generated by combinational logic versus
    - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**

CPI: Cycles Per Instruction

- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

- Single-cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

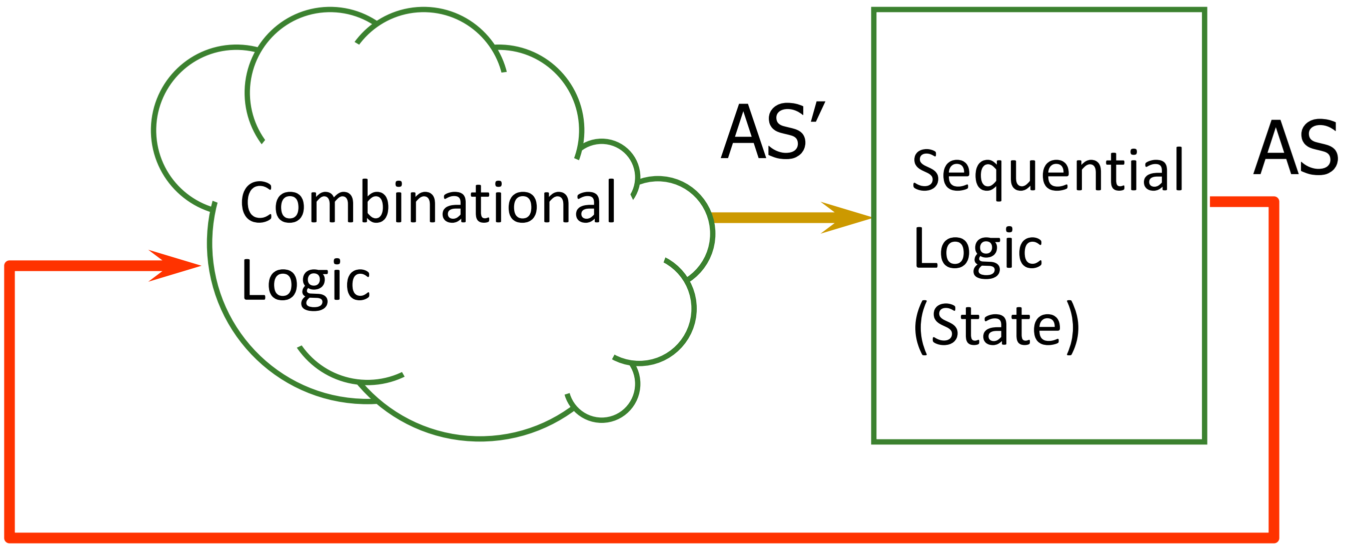
In multi-cycle, we have two degrees of freedom to optimize independently

A Single-Cycle Microarchitecture

From the Ground Up

Remember…

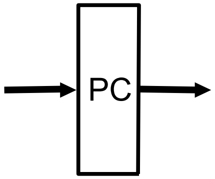
- Single-cycle machine

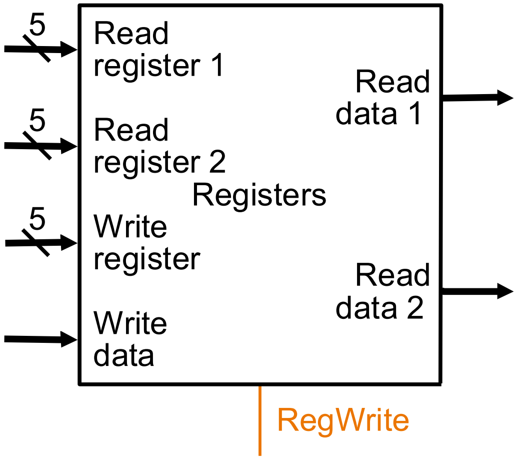


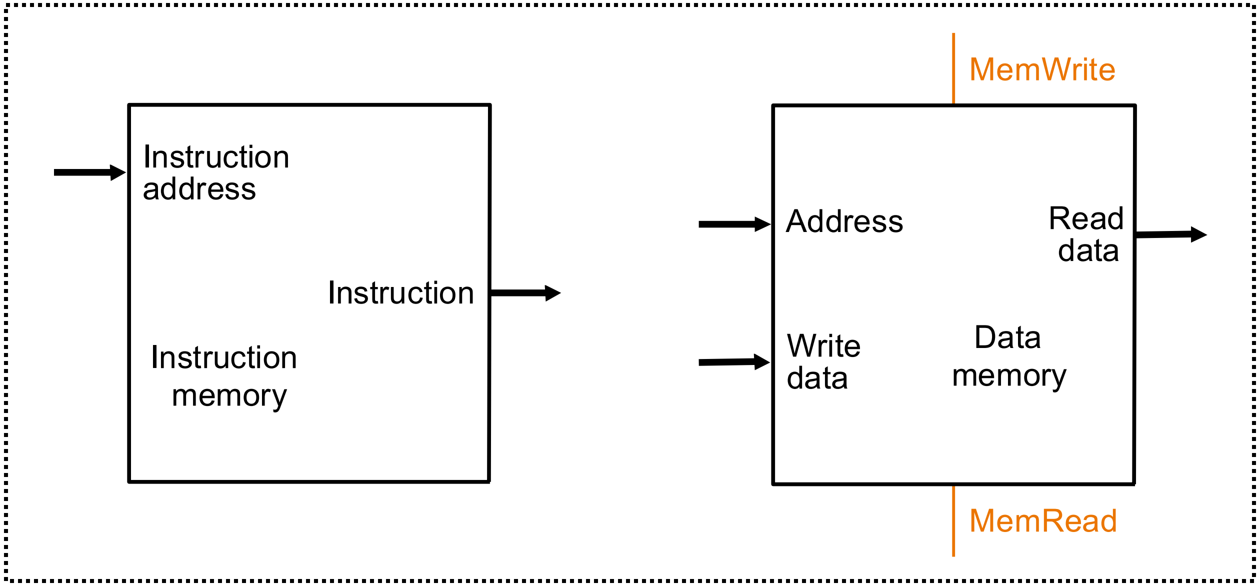
AS: Architectural State

Let’s Start with the State Elements (MIPS)

- Data and control inputs

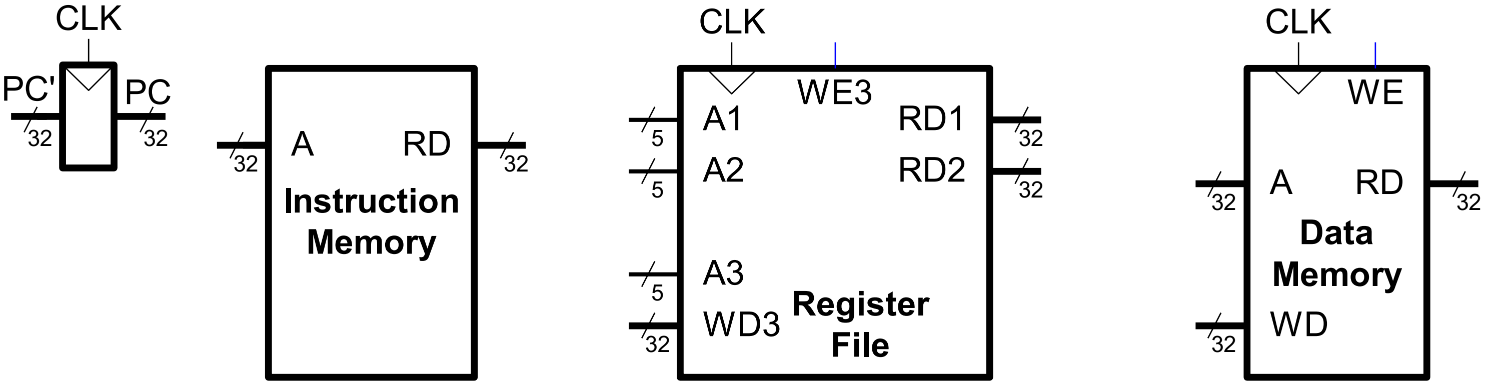






**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

MIPS State Elements



- Program counter:

  32-bit register

- Instruction memory:

  Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD

- Register file:

  The 32-element, 32-bit register file has 2 read ports and 1 write port

- Data memory:

  If the write enable, WE, is 1, it writes 32-bit data WD into memory location at 32-bit address A on the rising edge of the clock.

  If the write enable is 0, it reads 32-bit data from address A onto RD.

This notation is used in H&H single-cycle MIPS implementation (H&H Chapter 7.3)

For Now, We Will Assume

- “Magic” memory and register file

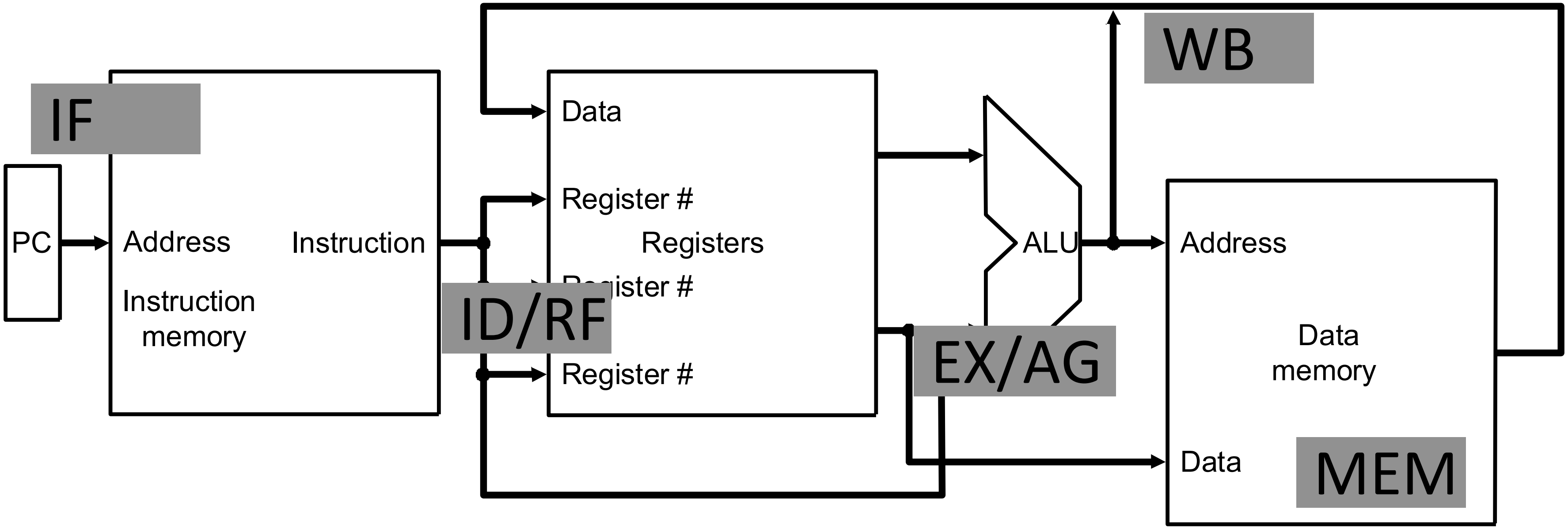
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port

- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is asserted
    - Cannot affect read output in between clock edges

- Single-cycle, synchronous memory
  - Contrast this with memory that tells when the data is ready
    - i.e., Ready signal: indicating the read or write is done
      - See P&P Appendix C (LC3-b) for multi-cycle memory

Instruction Processing

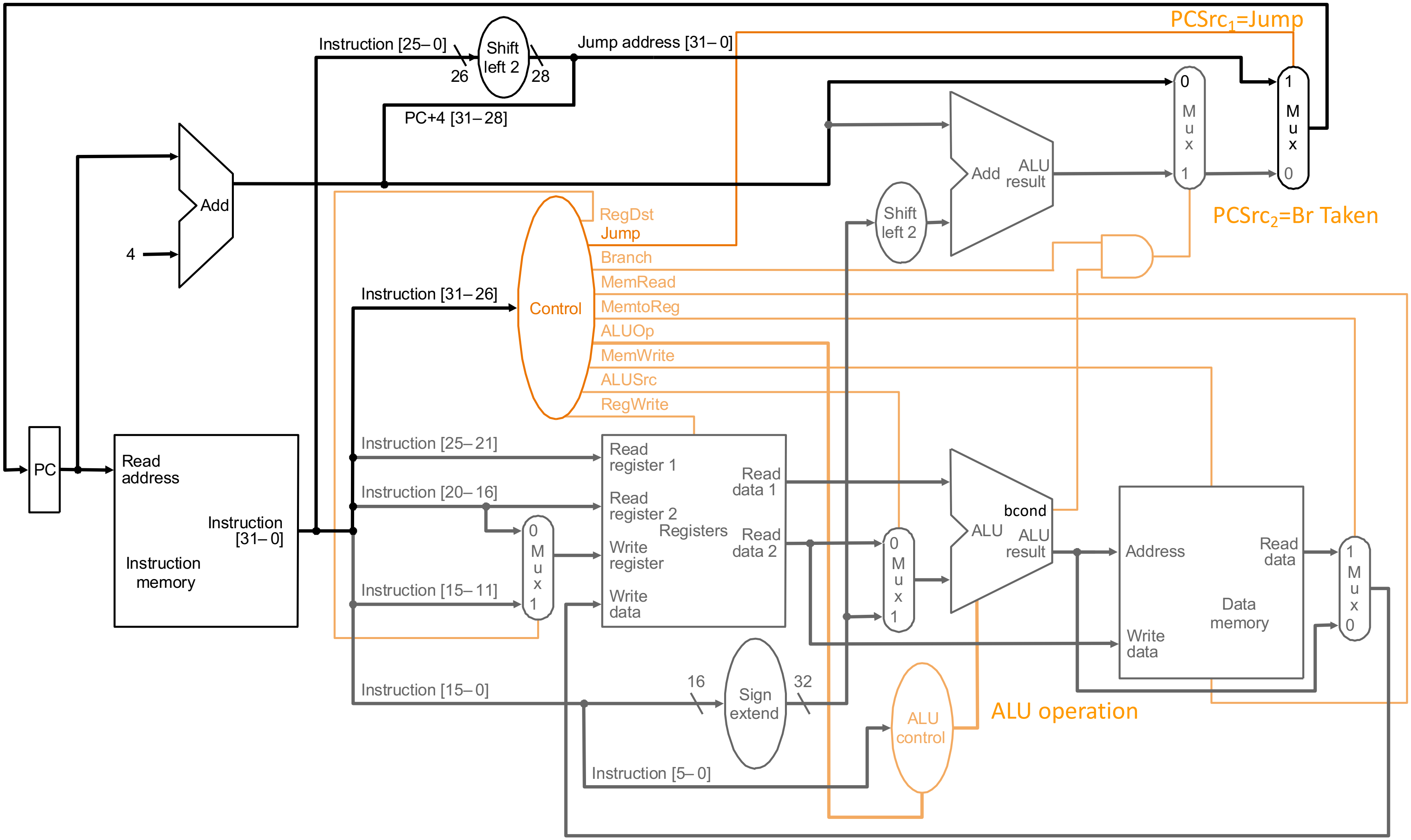
- 5 generic steps (P&H book)
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)



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

We Need to Provide the Datapath+Control Logic to Execute All ISA Instructions

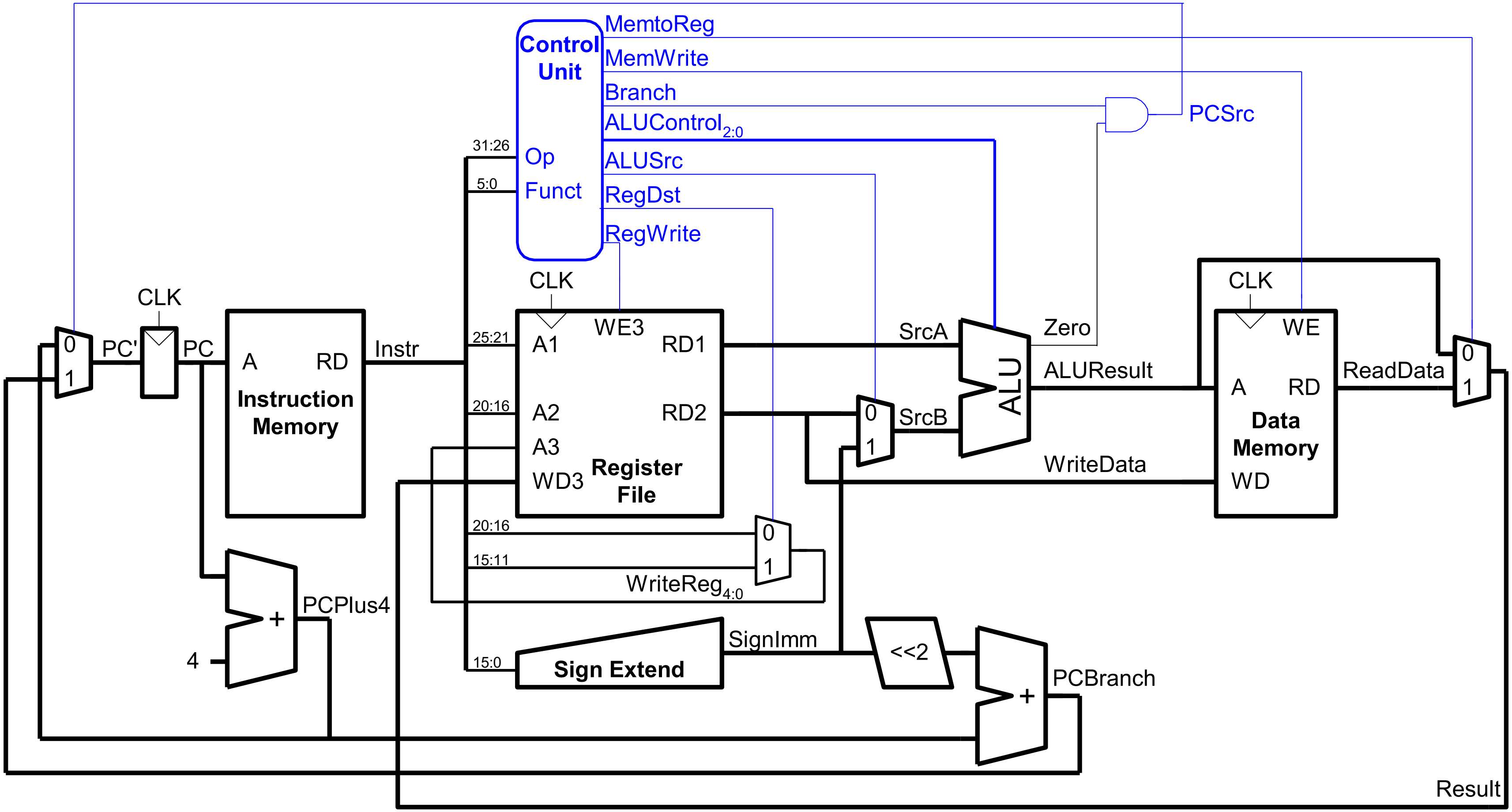
What Is To Come: Single-Cycle MIPS Processor



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

JAL, JR, JALR omitted

Another Complete Single-Cycle Processor



Single-cycle processor. Harris and Harris, Chapter 7.3.

Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

- R-type: 3 register operands

MIPS assembly (e.g., register-register signed addition)

```
add  $s0, $s1, $s2        #$s0=rd, $s1=rs, $s2=rt
```

Machine Encoding

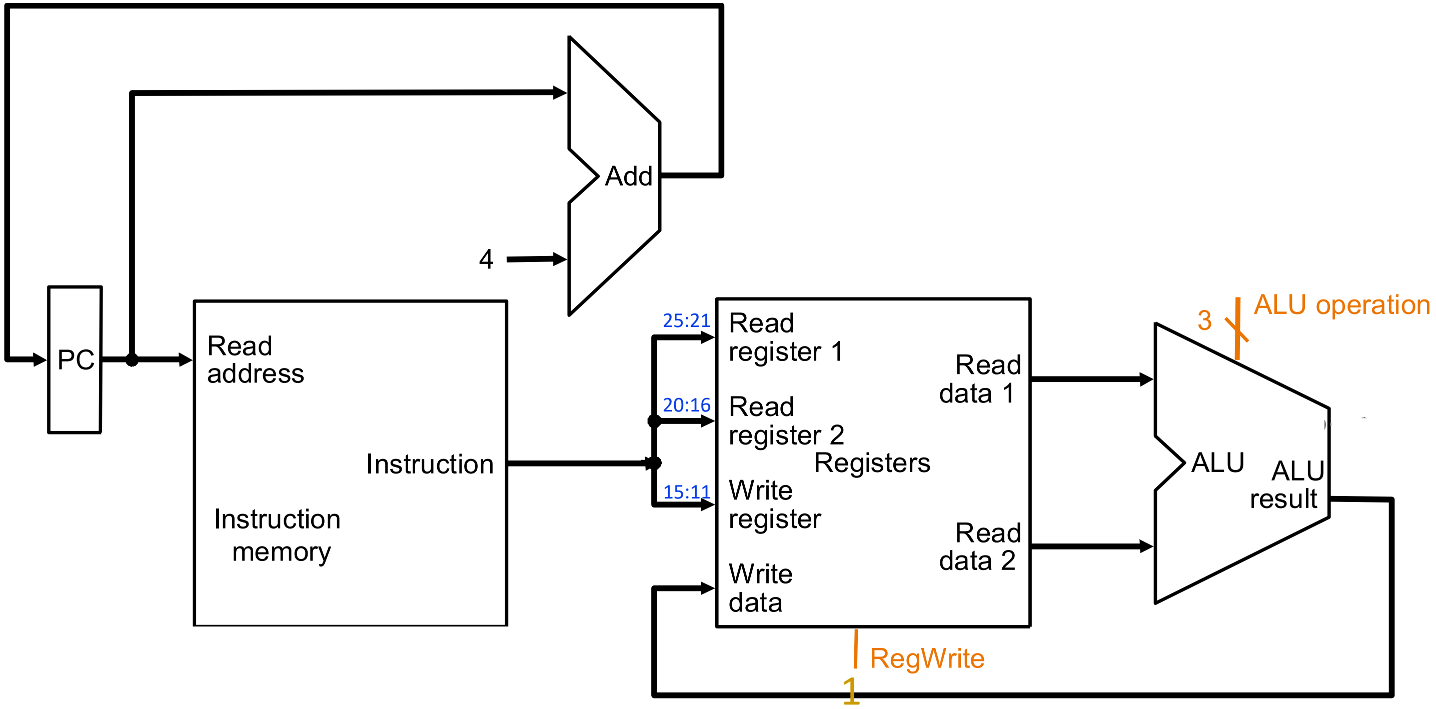
0	rs	rt	rd	0	add (32)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

- Semantics

```
if MEM[PC] == add rd rs rt
    GPR[rd] ← GPR[rs] + GPR[rt]
    PC ← PC + 4
```

(R-Type) ALU Datapath



```
if MEM[PC] == ADD rd rs rt
        GPR[rd] ← GPR[rs] + GPR[rt]
        PC ← PC + 4
```

IF	ID	EX	MEM	WB

Combinational state update logic

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

We Covered Until This Point in Lecture

Digital Design & Computer Arch.

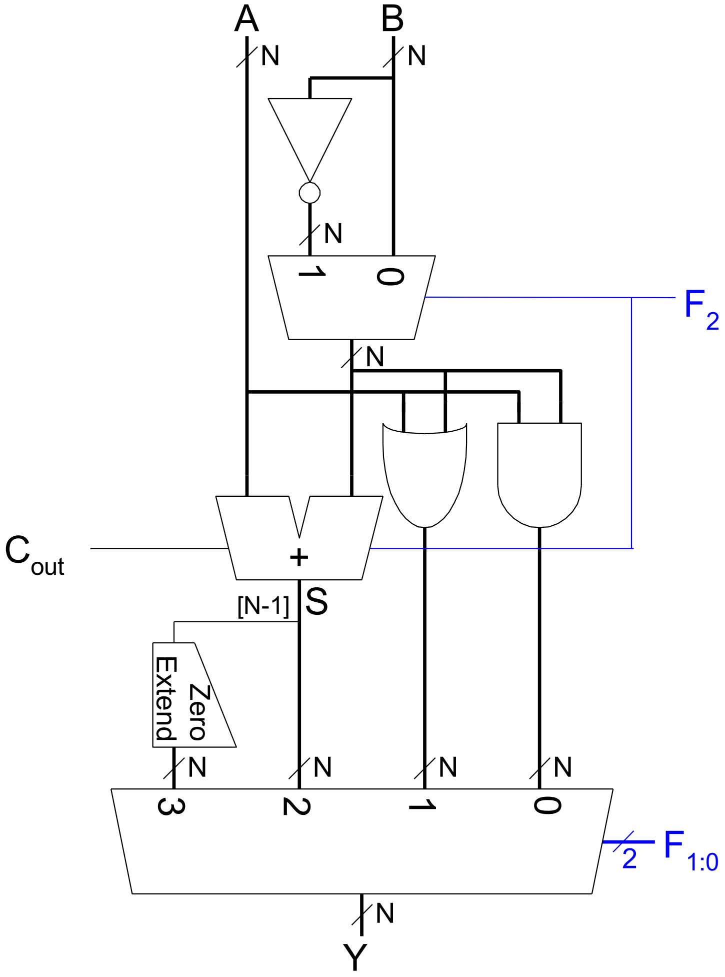
Lecture 11: Microarchitecture Fundamentals

Prof. Onur Mutlu

ETH Zürich
Spring 2022
31 March 2022

Example: ALU Design

- ALU operation ($F_{2:0}$) comes from the control logic



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

I-Type ALU Instructions

- I-type: 2 register operands and 1 immediate

MIPS assembly (e.g., register-immediate signed addition)

```
addi $s0, $s1, 5                #$s0=rt, $s1=rs
```

Machine Encoding

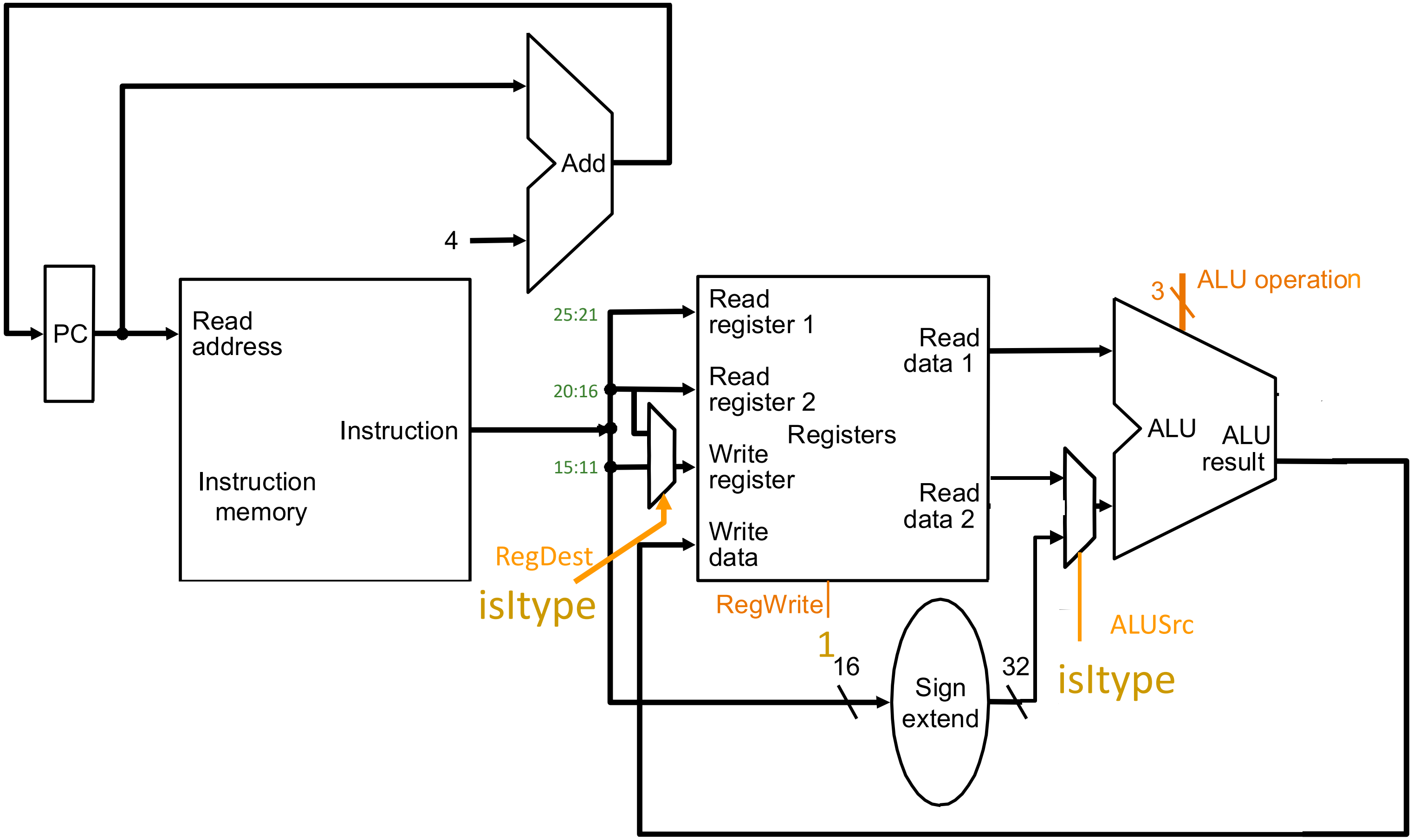
addi (0)	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

I-Type

- Semantics

```
if MEM[PC] == addi rs rt immediate
    PC ← PC + 4
    GPR[rt] ← GPR[rs] + sign-extend(immediate)
```

Datapath for R- and I-Type ALU Insts.



```
if MEM[PC] == ADDI rt rs immediate
    GPR[rt] ← GPR[rs] + sign-extend (immediate)
    PC ← PC + 4
```

IF | ID | EX | MEM | WB

Combinational
state update logic

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

Recall: ADD with one Literal in LC-3

- ADD assembly and machine code

LC-3 assembly

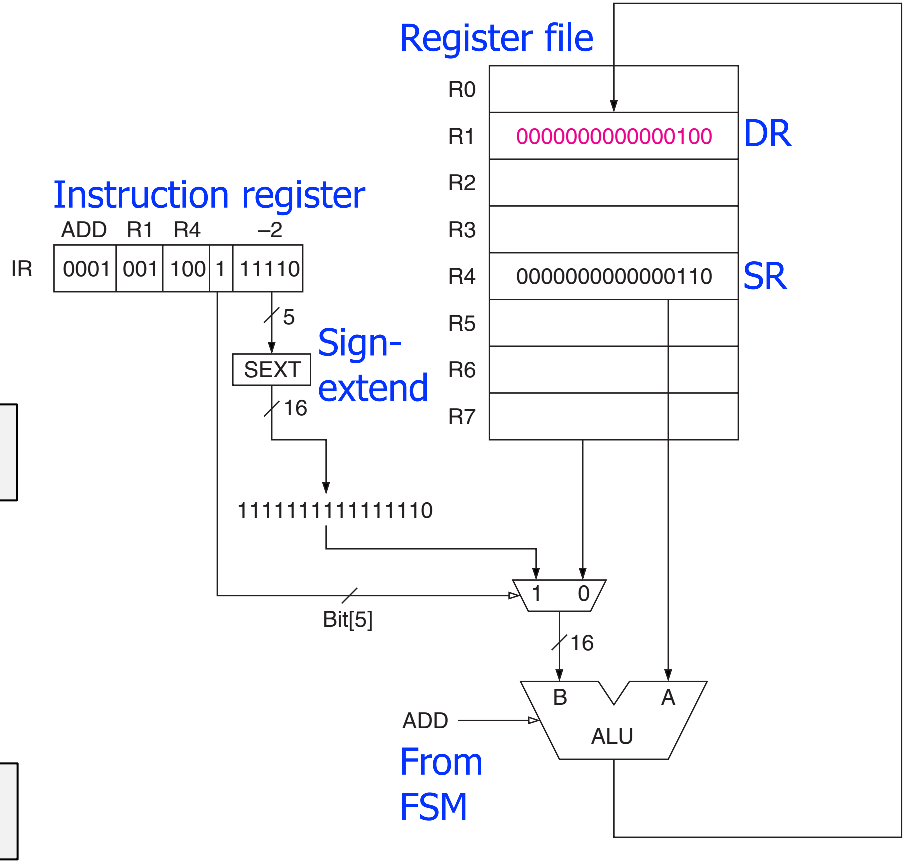
```
ADD R1, R4, #-2
```

Field Values

OP	DR	SR		imm5
1	1	4	1	-2

Machine Code

OP	DR	SR		imm5
0 0 0 1	0 0 1	1 0 0	1	1 1 1 1 0
15 12	11 9	8 6	5	4 0



Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

- Load 4-byte word

MIPS assembly

```
lw  $s3, 8($s0)                     #$s0=rs, $s3=rt
```

Machine Encoding

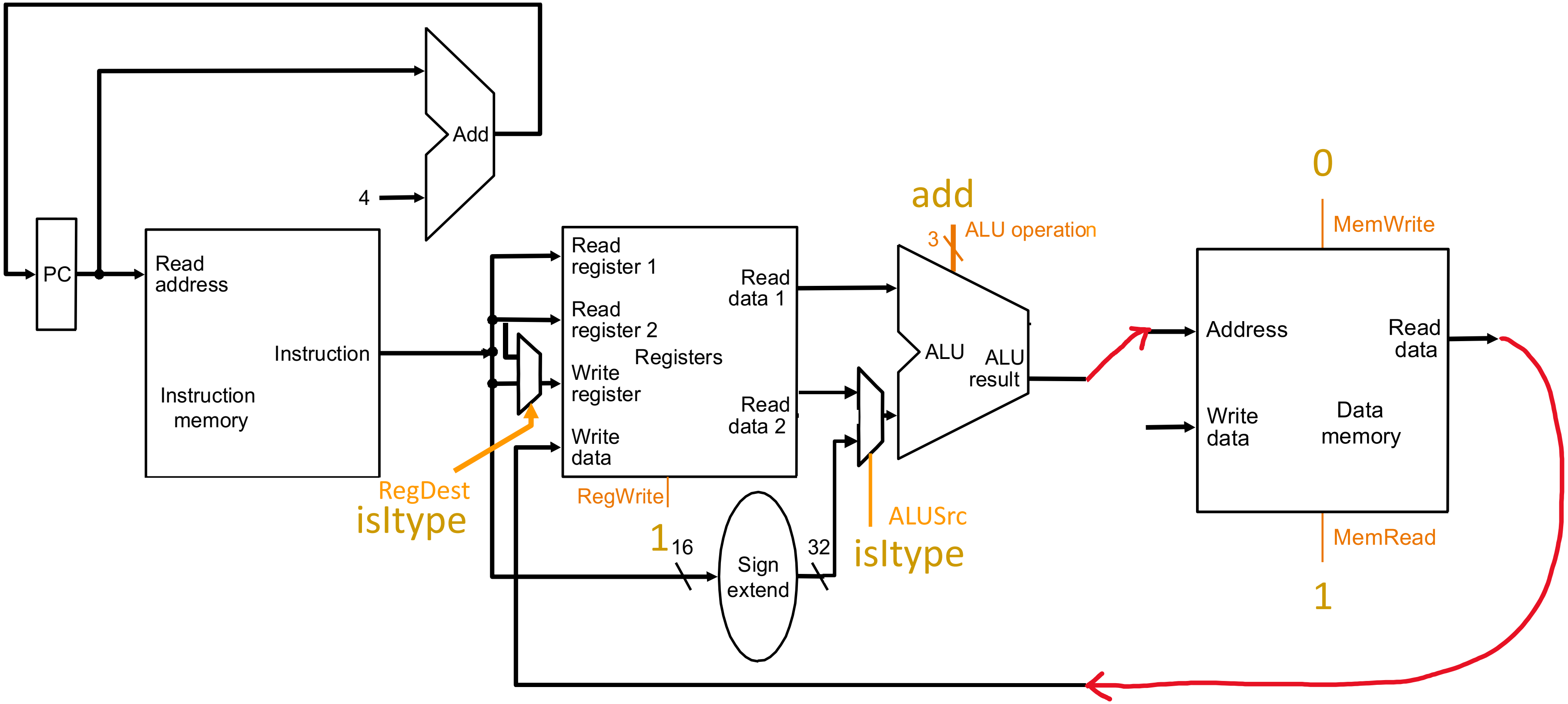
op	rs=base	rt	imm=offset
lw (35)	base	rt	offset
31 26	25 21	20 16	15 0

I-Type

- Semantics

```
if MEM[PC] == lw rt $\text{offset}_{16}$ (base)
    PC ← PC + 4
    EA = sign-extend(offset) + GPR(base)
    GPR[rt] ← MEM[ translate(EA) ]
```

LW Datapath



if MEM[PC]==LW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

GPR[rt] ← MEM[translate(EA)]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational

state update logic

Store Instructions

- Store 4-byte word

MIPS assembly

```
sw   $s3, 8($s0)                    #$s0=rs, $s3=rt
```

Machine Encoding

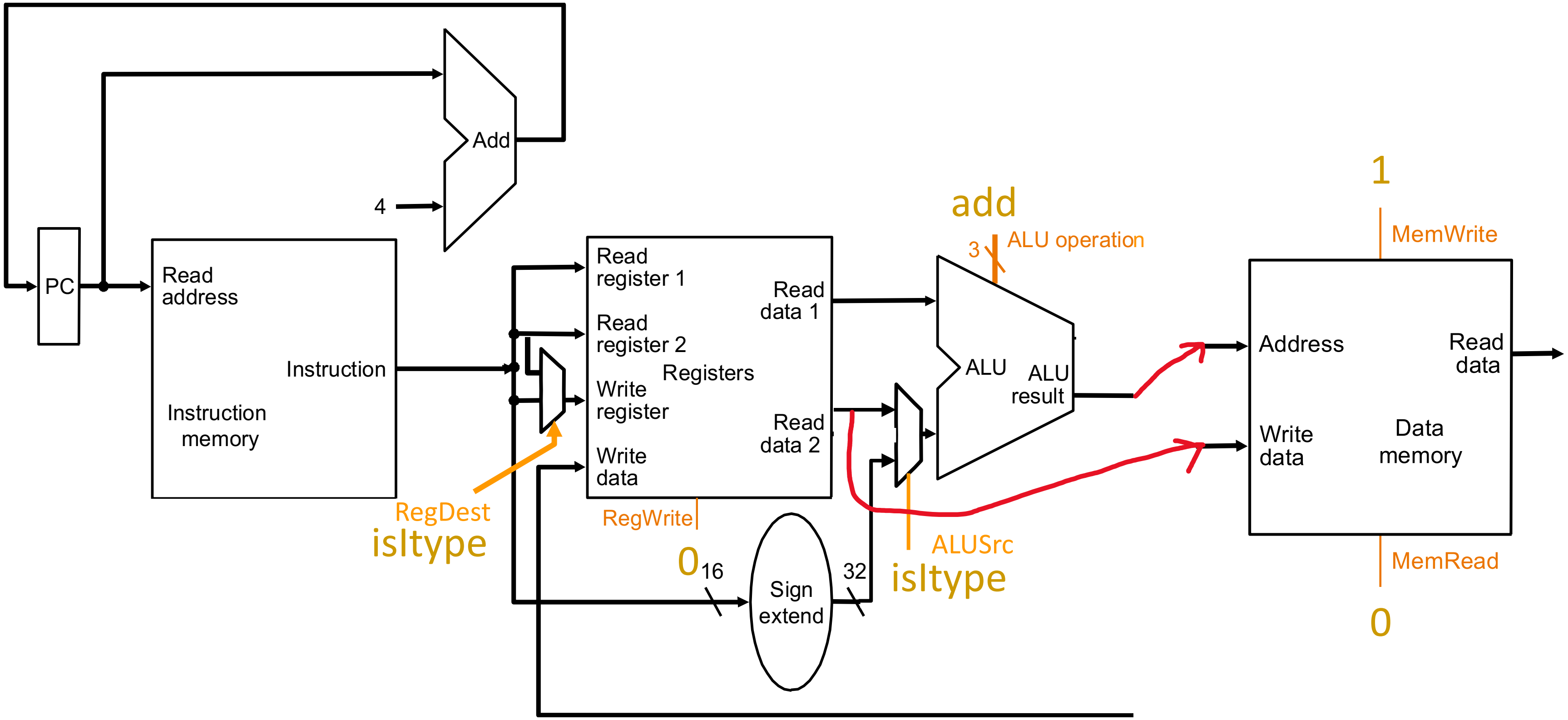
op	rs=base	rt	imm=offset
sw (43)	base	rt	offset
31 26	25 21	20 16	15 0

I-Type

- Semantics

```
if Mem[PC] == sw rt $\text{offset}_{16}$ (base)
    PC ← PC + 4
    EA = sign-extend(offset) + GPR(base)
    MEM[ translate(EA) ] ← GPR[rt]
```

SW Datapath

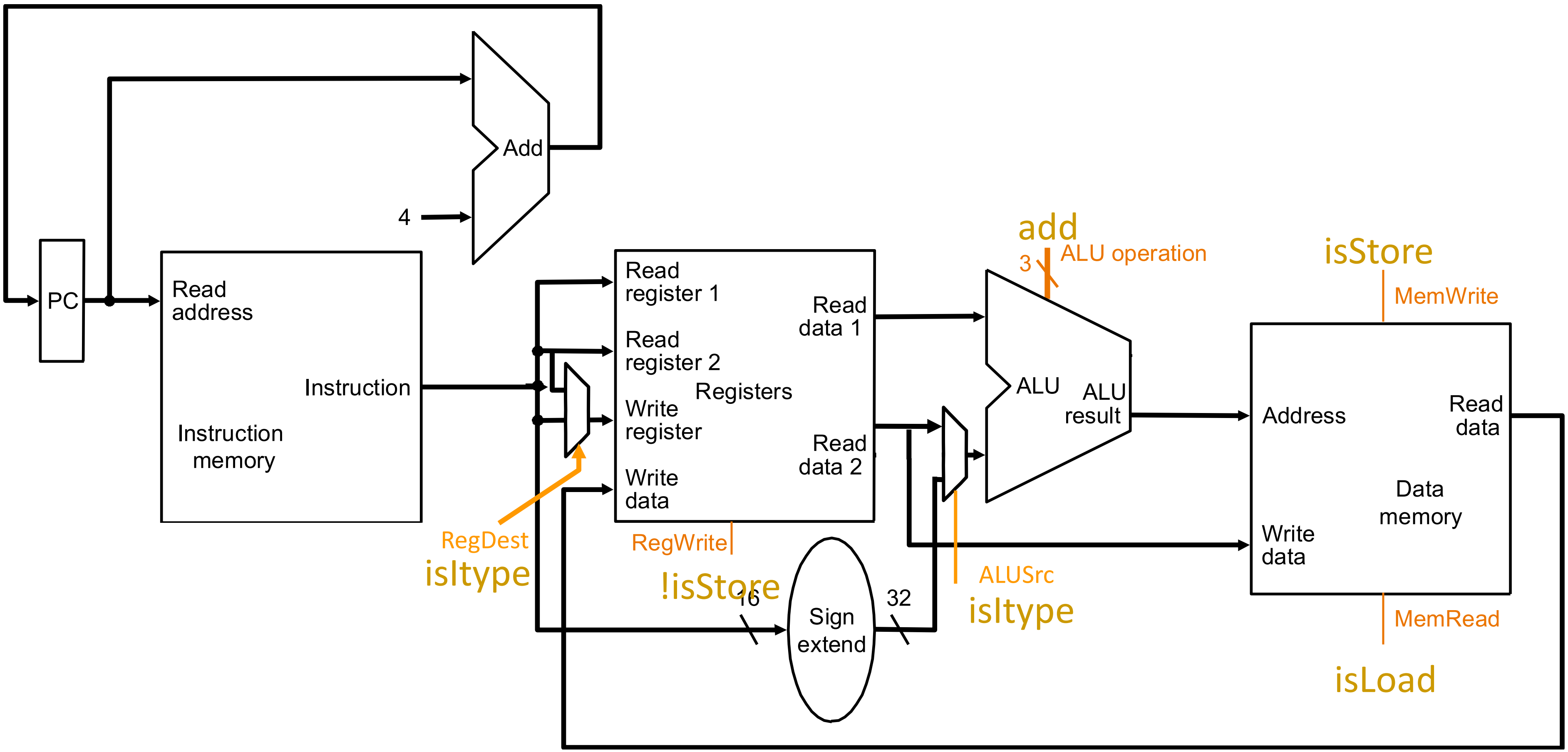


if MEM[PC]==SW rt offset$_{16}$ (base)
EA = sign-extend(offset) + GPR[base]
MEM[translate(EA)] ← GPR[rt]
PC ← PC + 4

IF | ID | EX | MEM | WB

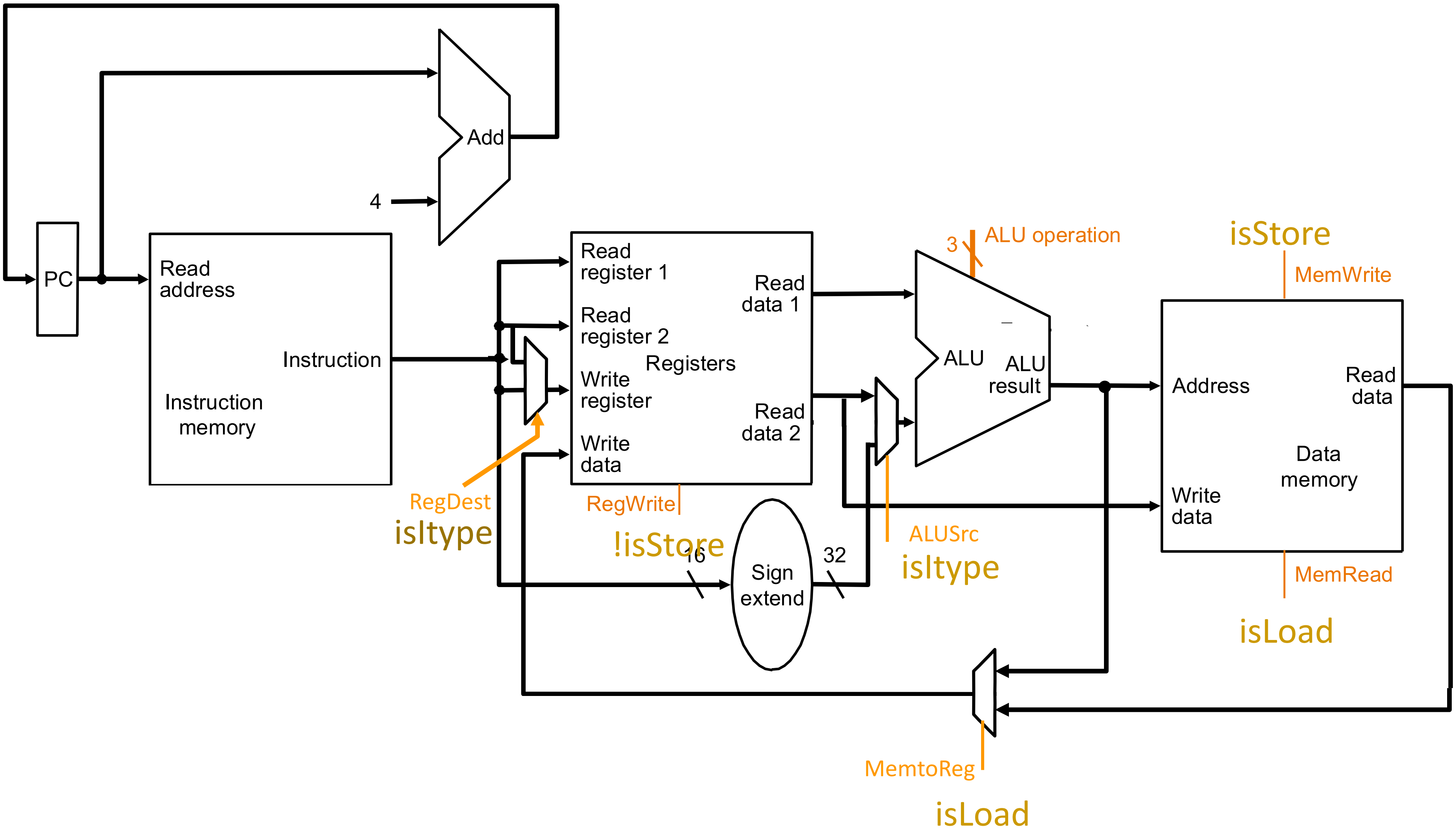
Combinational
state update logic

Load-Store Datapath



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

Datapath for Non-Control-Flow Insts.



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

Single-Cycle Datapath for *Control Flow Instructions*

Jump Instruction

- Unconditional branch or jump

```
j   target
```

j (2)	immediate
6 bits	26 bits

J-Type

  - 2 = opcode
  - immediate (target) = target address

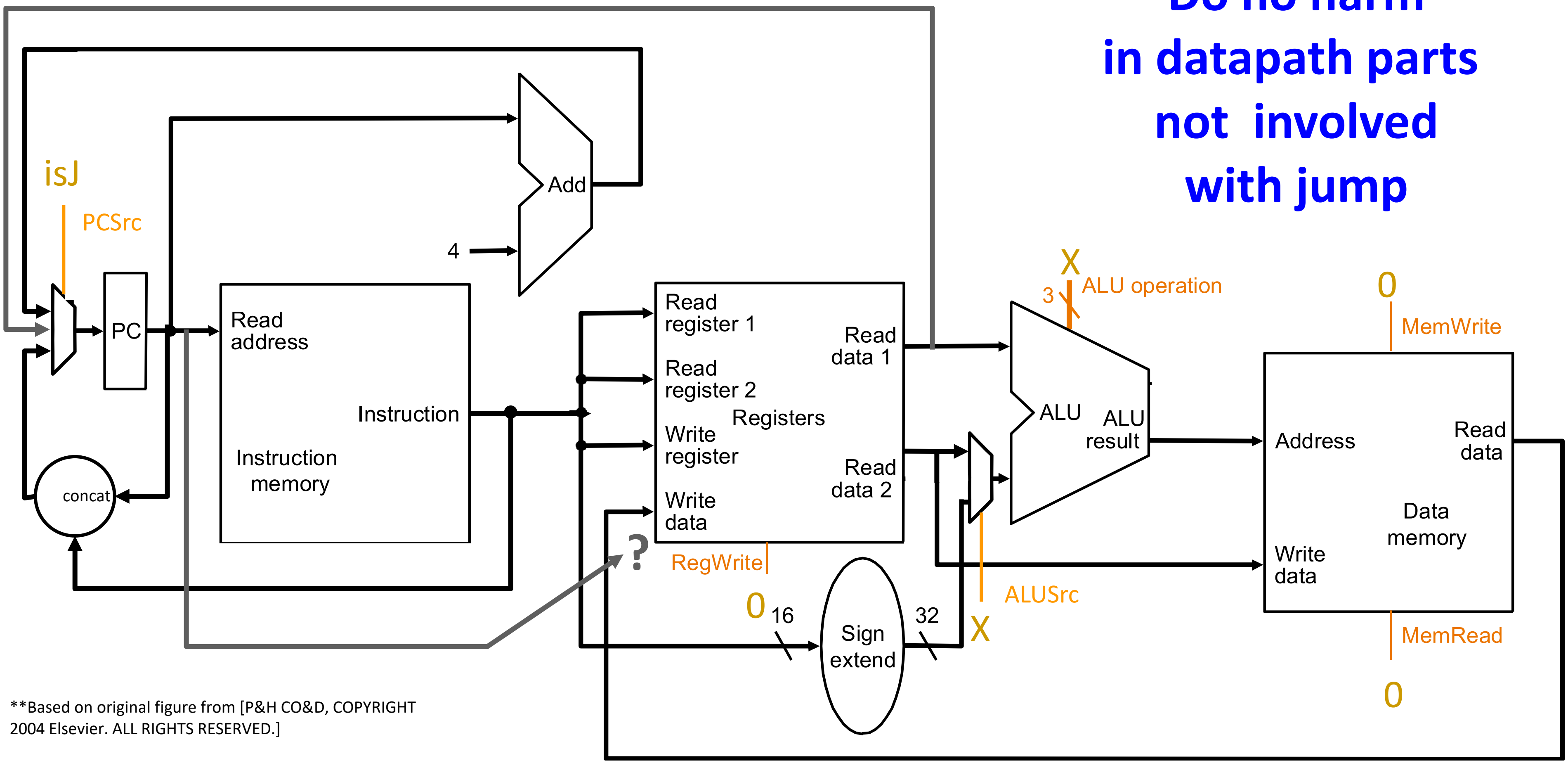
- Semantics

```
if MEM[PC]== j $\text{immediate}_{26}$
    target = { PC[†] [31:28], $\text{immediate}_{26}$, 2’b00 }
    PC ← target
```

[†] This is the incremented PC

Unconditional Jump Datapath

Do no harm in datapath parts not involved with jump



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

```
if MEM[PC]==J immediate26
   PC = { PC[31:28], immediate26, 2’b00 }
```

What about JR, JAL, JALR?

Other Jumps in MIPS

- jr: jump register

  Semantics

  ```
  if MEM[PC]== jr rs
      PC ← GPR(rs)
  ```

- jal: jump and link (function calls)

  Semantics

  ```
  if MEM[PC]== jal $immediate_{26}$
      $ra ← PC + 4
      target = { PC [†][31:28], $immediate_{26}$, 2'b00 }
      PC ← target
  ```

- jalr: jump and link register

  Semantics

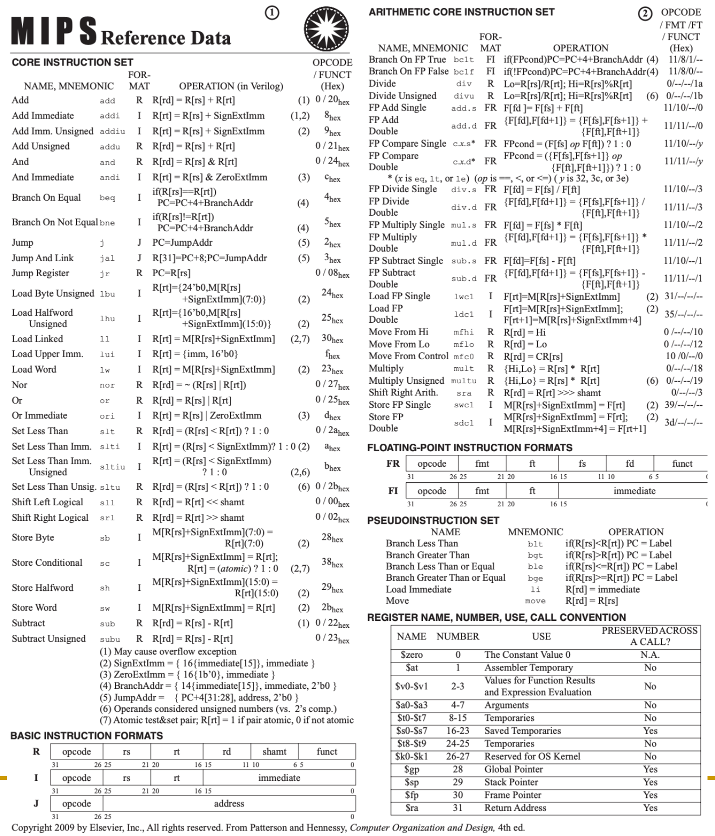
  ```
  if MEM[PC]== jalr rs
      $ra ← PC + 4
      PC ← GPR(rs)
  ```

[†] This is the incremented PC

Aside: MIPS Cheat Sheet

- https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=mips_reference_data.pdf

- On the course website



MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20$_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8$_{hex}$
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9$_{hex}$
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21$_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24$_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c$_{hex}$
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4$_{hex}$
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5$_{hex}$
Jump	j	J	PC=JumpAddr	(5)	2$_{hex}$
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3$_{hex}$
Jump Register	jr	R	PC=R[rs]		0 / 08$_{hex}$
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24$_{hex}$
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25$_{hex}$
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30$_{hex}$
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f$_{hex}$
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23$_{hex}$
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27$_{hex}$
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25$_{hex}$
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d$_{hex}$
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / 2a$_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a$_{hex}$
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b$_{hex}$
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / 2b$_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00$_{hex}$
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02$_{hex}$
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28$_{hex}$
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38$_{hex}$
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29$_{hex}$
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	2b$_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22$_{hex}$
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23$_{hex}$

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

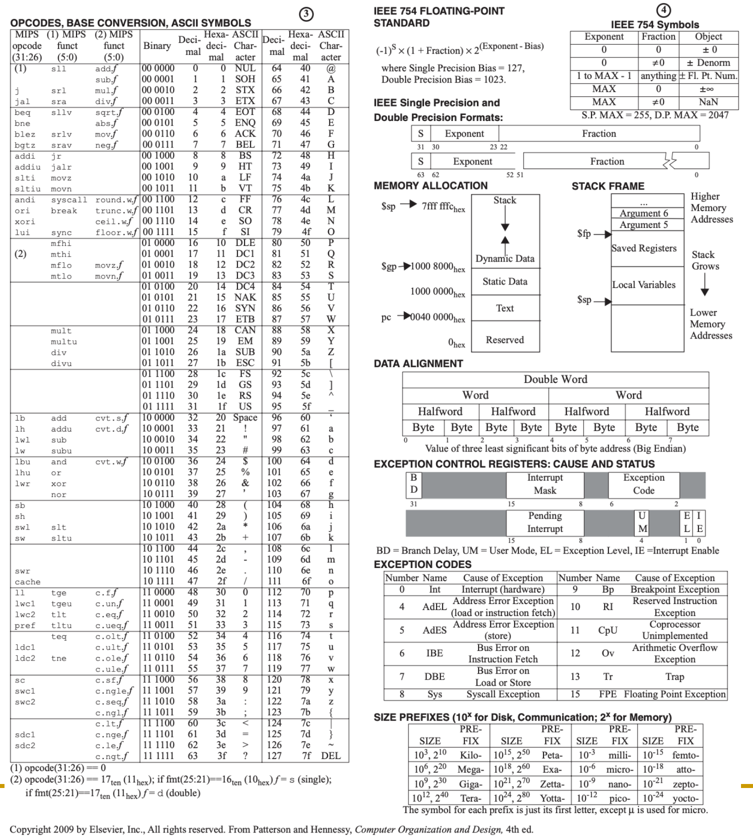
FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15 0		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes



OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	’	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0
(2) opcode(31:26) == 17$_{ten}$ (11$_{hex}$); if fmt(25:21)==16$_{ten}$ (10$_{hex}$) *f* = s (single);
if fmt(25:21)==17$_{ten}$ (11$_{hex}$) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD ④

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:

S	Exponent	Fraction
31	30 23	22 0
S	Exponent	Fraction
63	62 52	51 0

MEMORY ALLOCATION

$sp → 7fff fffc$_{hex}$	Stack
	Dynamic Data
$gp → 1000 8000$_{hex}$	Static Data
1000 0000$_{hex}$	Text
pc → 0040 0000$_{hex}$	Reserved
0$_{hex}$	

STACK FRAME

	...	Higher Memory Addresses
	Argument 6	
	Argument 5	
$fp →	Saved Registers	Stack Grows ↓
	Local Variables	
$sp →		Lower Memory Addresses

DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS

B D	Interrupt Mask	Exception Code	
31	15 8	6 2	
	Pending Interrupt	U M	E L / I E
	15 8	4	1 0

BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
$10^3, 2^{10}$	Kilo-	$10^{15}, 2^{50}$	Peta-	10^{-3}	milli-	10^{-15}	femto-
$10^6, 2^{20}$	Mega-	$10^{18}, 2^{60}$	Exa-	10^{-6}	micro-	10^{-18}	atto-
$10^9, 2^{30}$	Giga-	$10^{21}, 2^{70}$	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
$10^{12}, 2^{40}$	Tera-	$10^{24}, 2^{80}$	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except µ is used for micro.

Conditional Branch Instructions

- beq (Branch if Equal)

```
beq  $s0, $s1, offset          #$s0=rs,$s1=rt
```

beq (4)	rs	rt	immediate=offset
6 bits	5 bits	5 bits	16 bits

I-Type

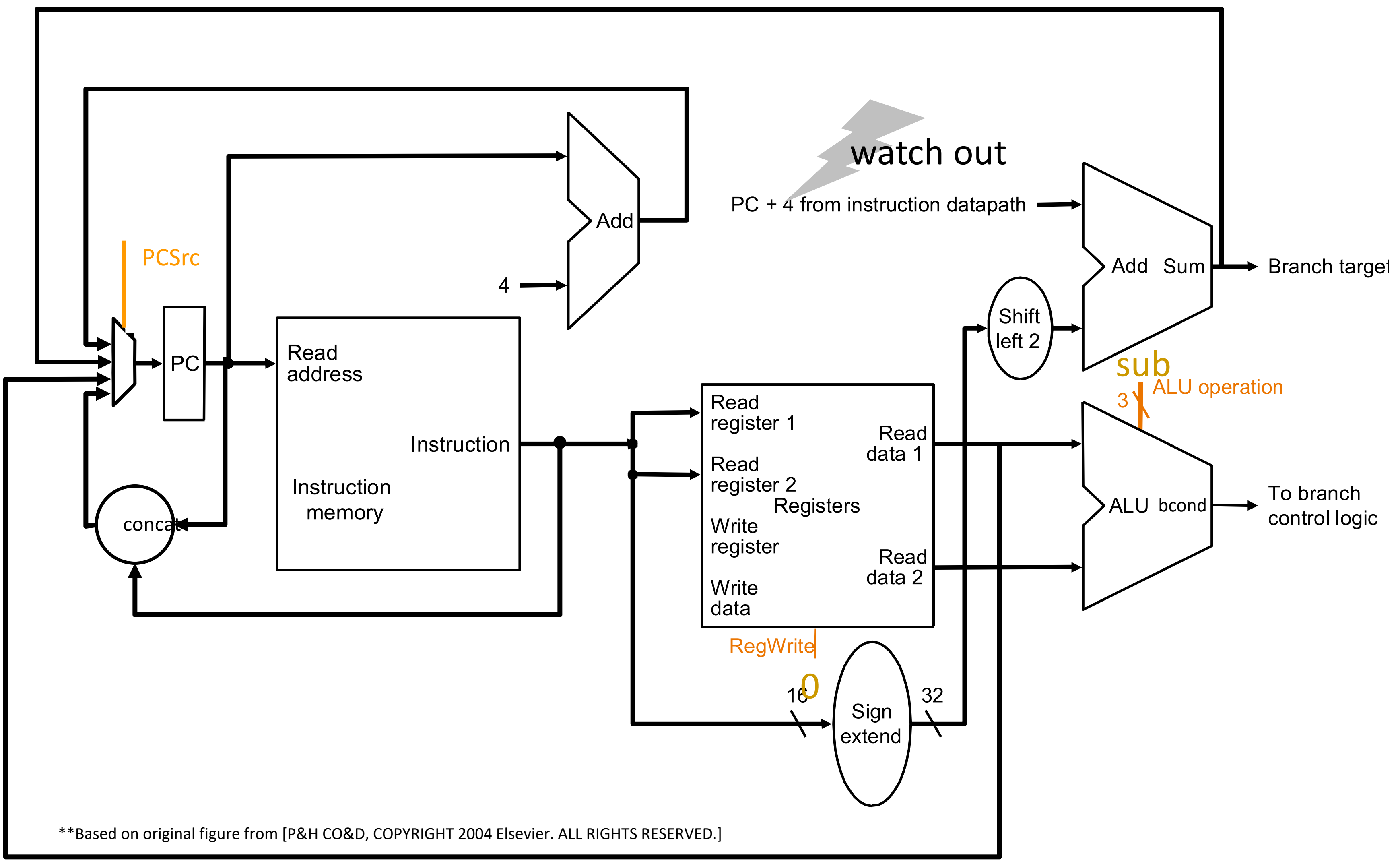
- Semantics (assuming no branch delay slot)

```
if MEM[PC] == beq rs rt immediate_16
    target = PC† + sign-extend(immediate) x 4
    if GPR[rs]==GPR[rt] then PC ← target
    else PC ← PC + 4
```

  - Variations: beq, bne, blez, bgtz

[†] This is the incremented PC

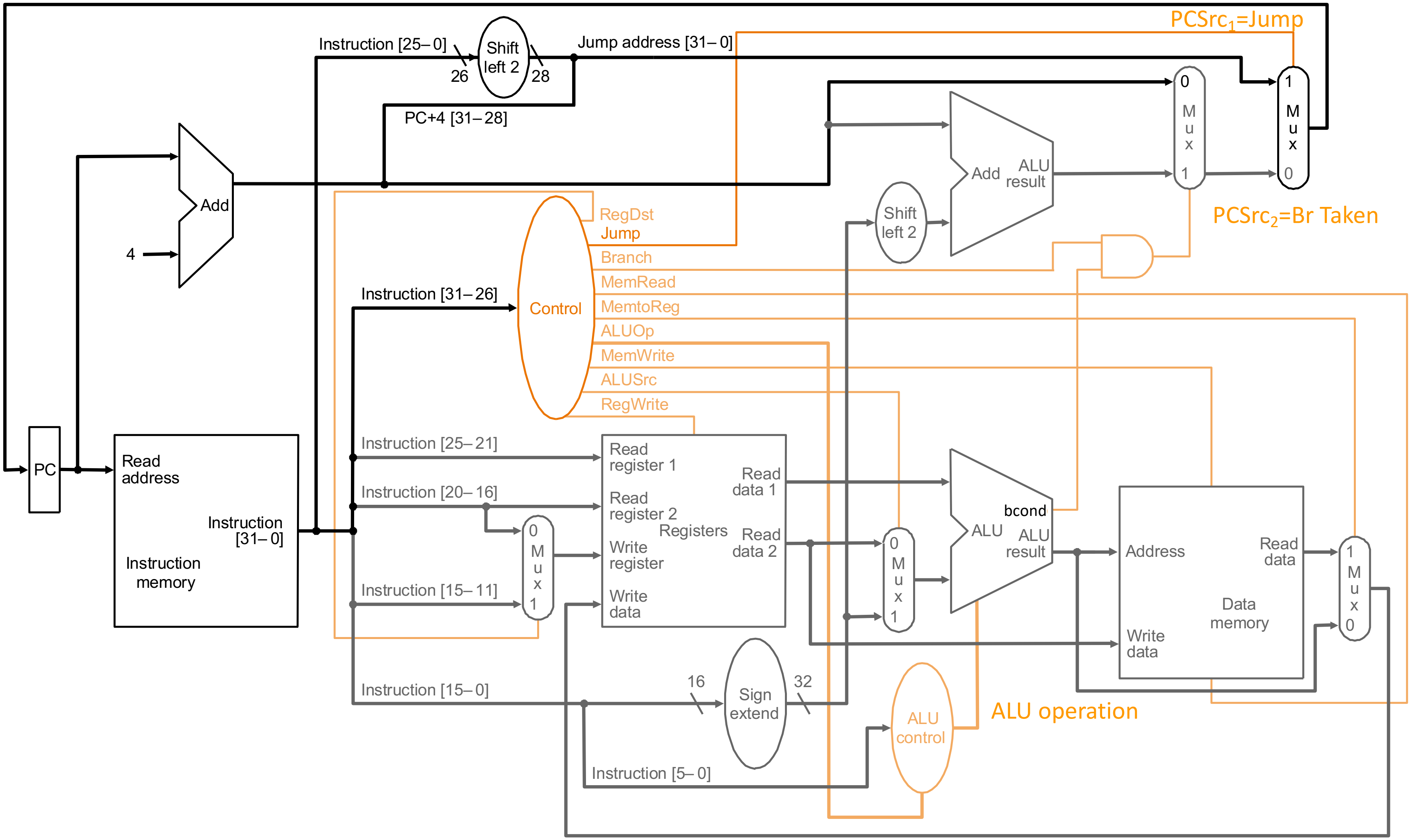
Conditional Branch Datapath (for you to finish)



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

How to uphold the delayed branch semantics?

Putting It All Together



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

JAL, JR, JALR omitted

Single-Cycle Control Logic

Single-Cycle Hardwired Control

- **As combinational function** of Inst=MEM[PC]

31 26	25 21	20 16	15 11	10 6	5 0
0	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

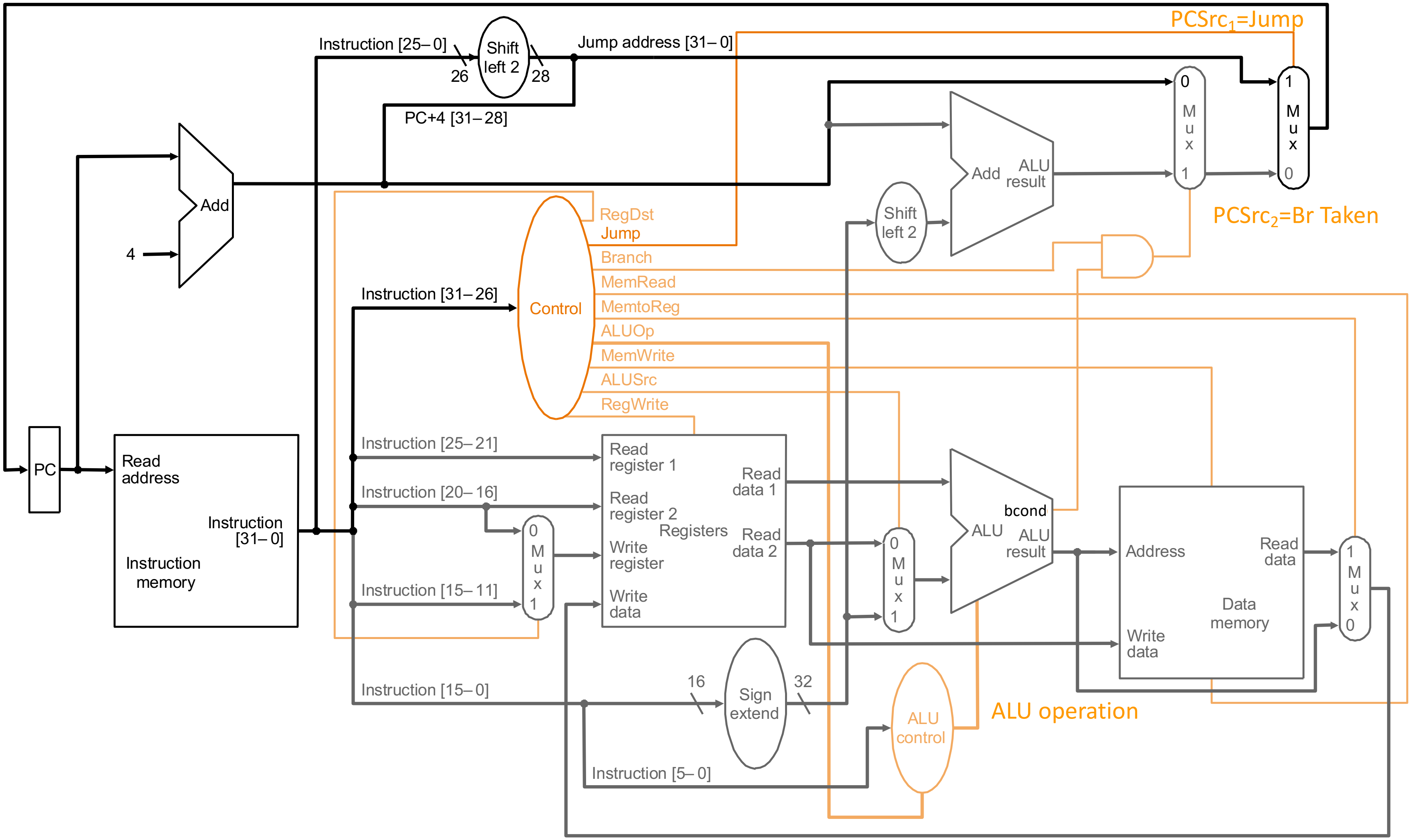
I-Type

31 26	25 0
opcode	immediate
6 bits	26 bits

J-Type

- Consider
  - All R-type and I-type ALU instructions
  - lw and sw
  - beq, bne, blez, bgtz
  - j, jr, jal, jalr

Generate Control Signals (in Orange Color)



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

JAL, JR, JALR omitted

Single-Bit Control Signals (I)

	When De-asserted	When asserted	Equation
RegDest	GPR write select according to rt, i.e., inst[20:16]	GPR write select according to rd, i.e., inst[15:11]	opcode==0
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) **&&** (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	Steer memory output to GPR write port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

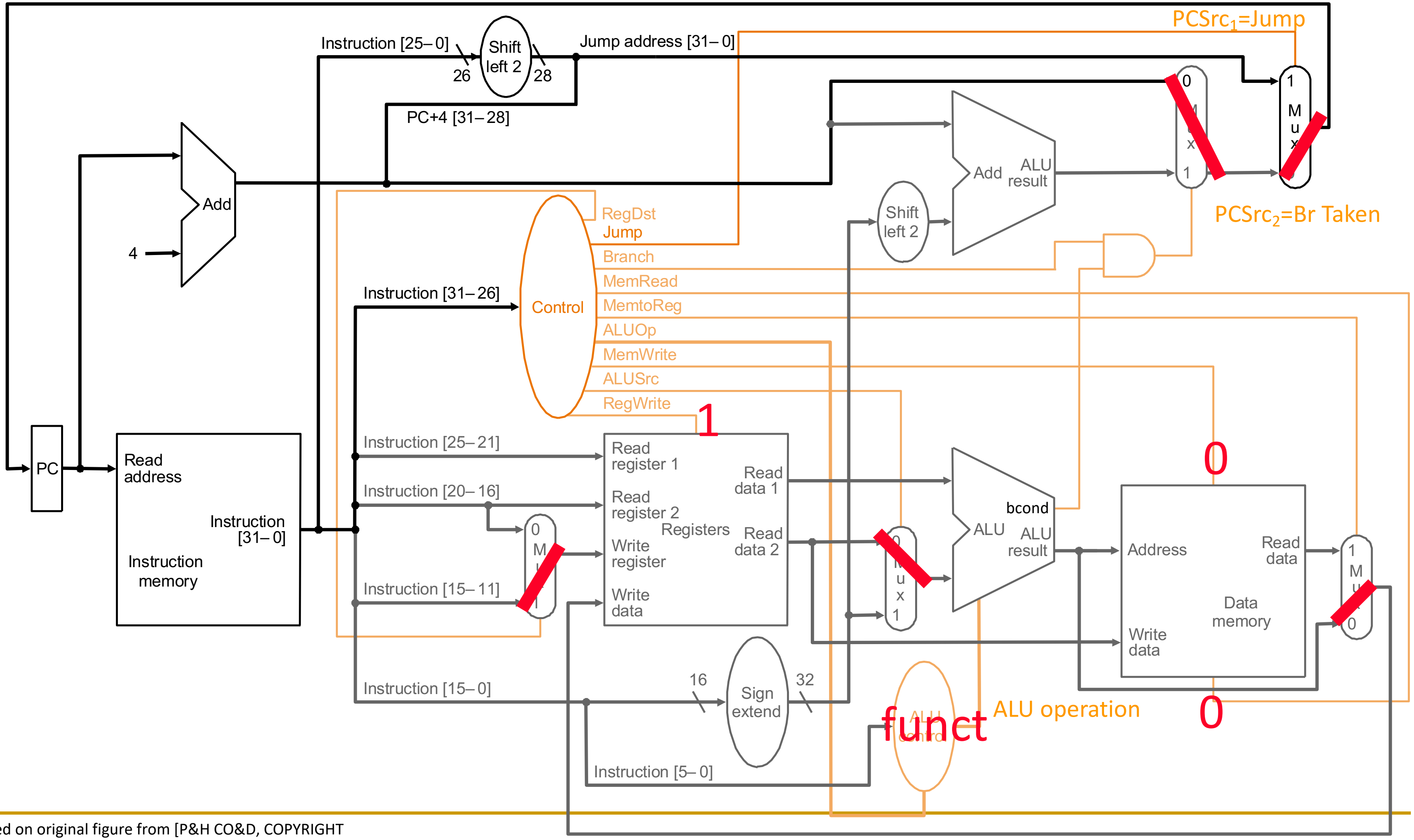
JAL and JALR require additional RegDest and MemtoReg options

Single-Bit Control Signals (II)

	When De-asserted	**When asserted**	**Equation**
MemRead	Memory read disabled	Memory read port returns load value	opcode==LW
MemWrite	Memory write disabled	Memory write enabled	opcode==SW
$PCSrc_1$	According to $PCSrc_2$	next PC is based on 26-bit immediate jump target	(opcode==J) || (opcode==JAL)
$PCSrc_2$	next PC = PC + 4	next PC is based on 16-bit immediate branch target	(opcode==Bxx) && "bcond is satisfied"

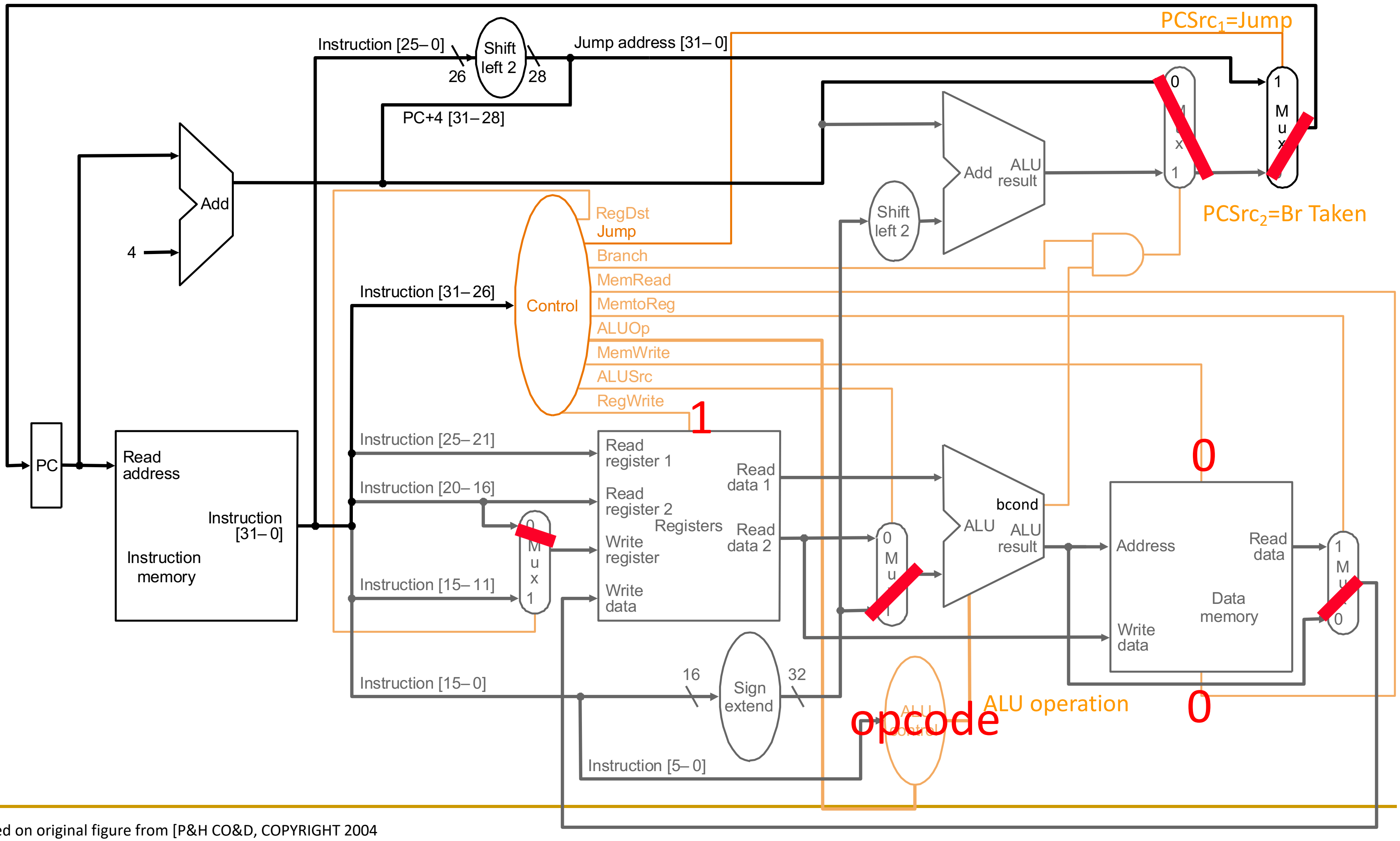
JR and JALR require additional PCSrc options

R-Type ALU



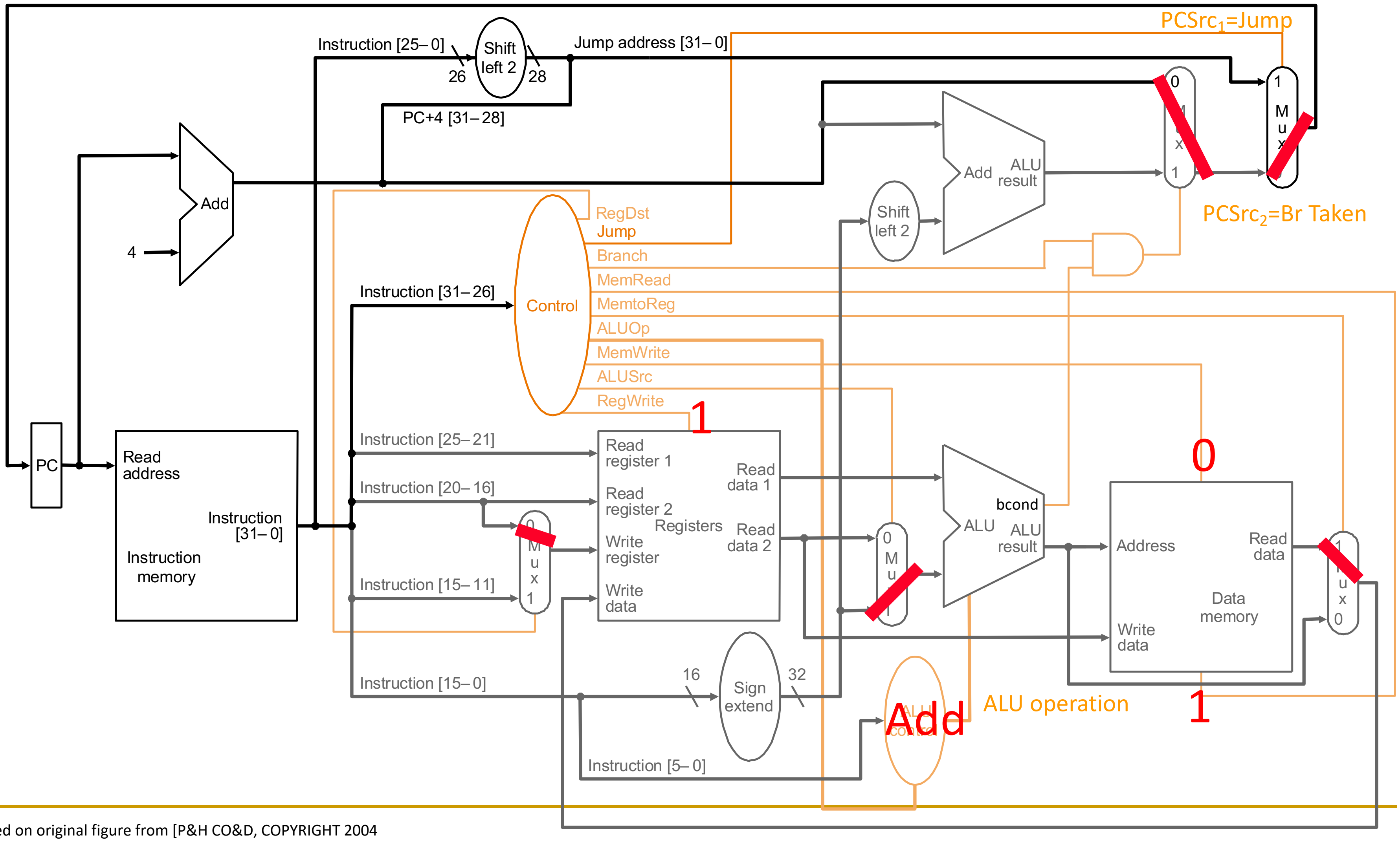
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



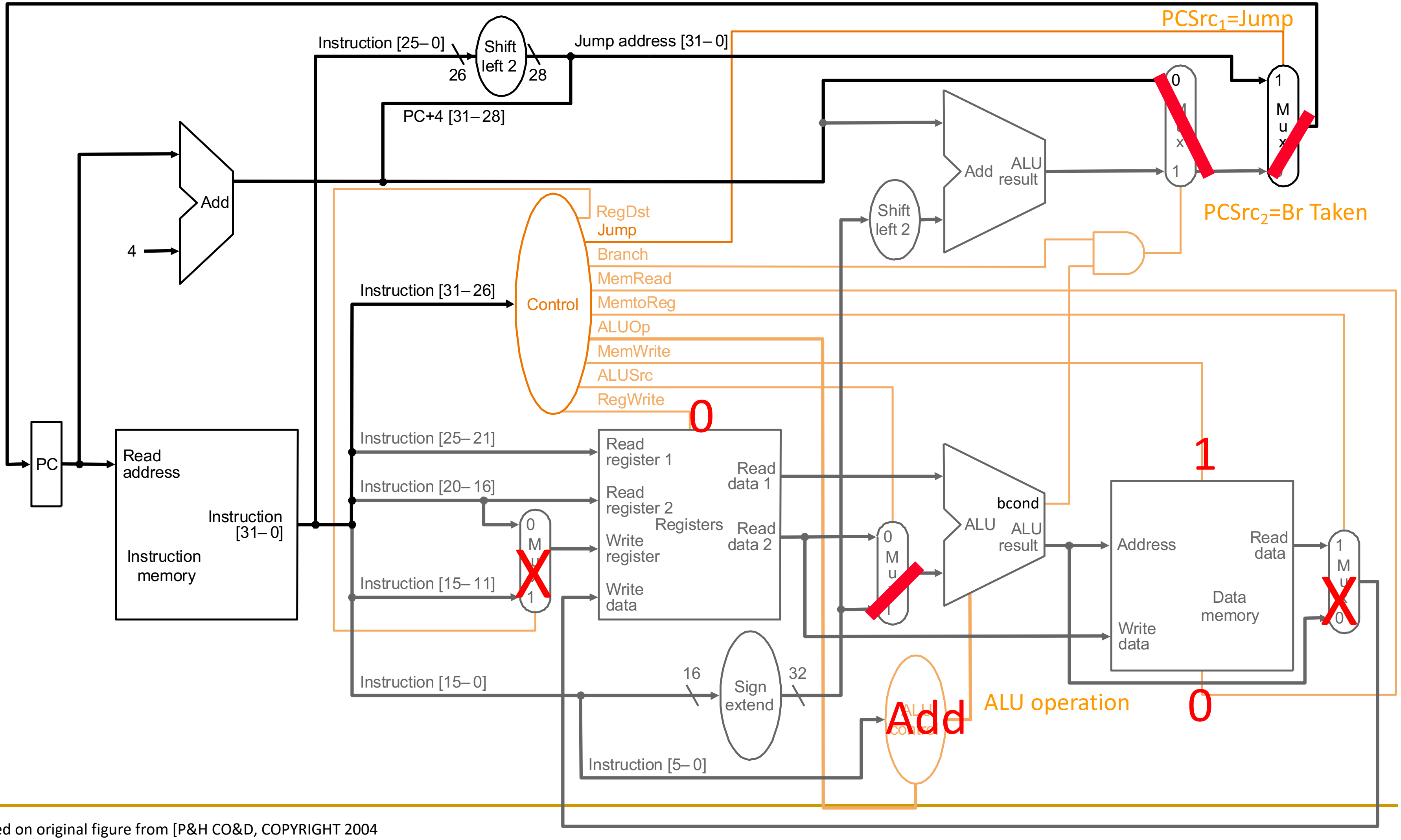
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



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

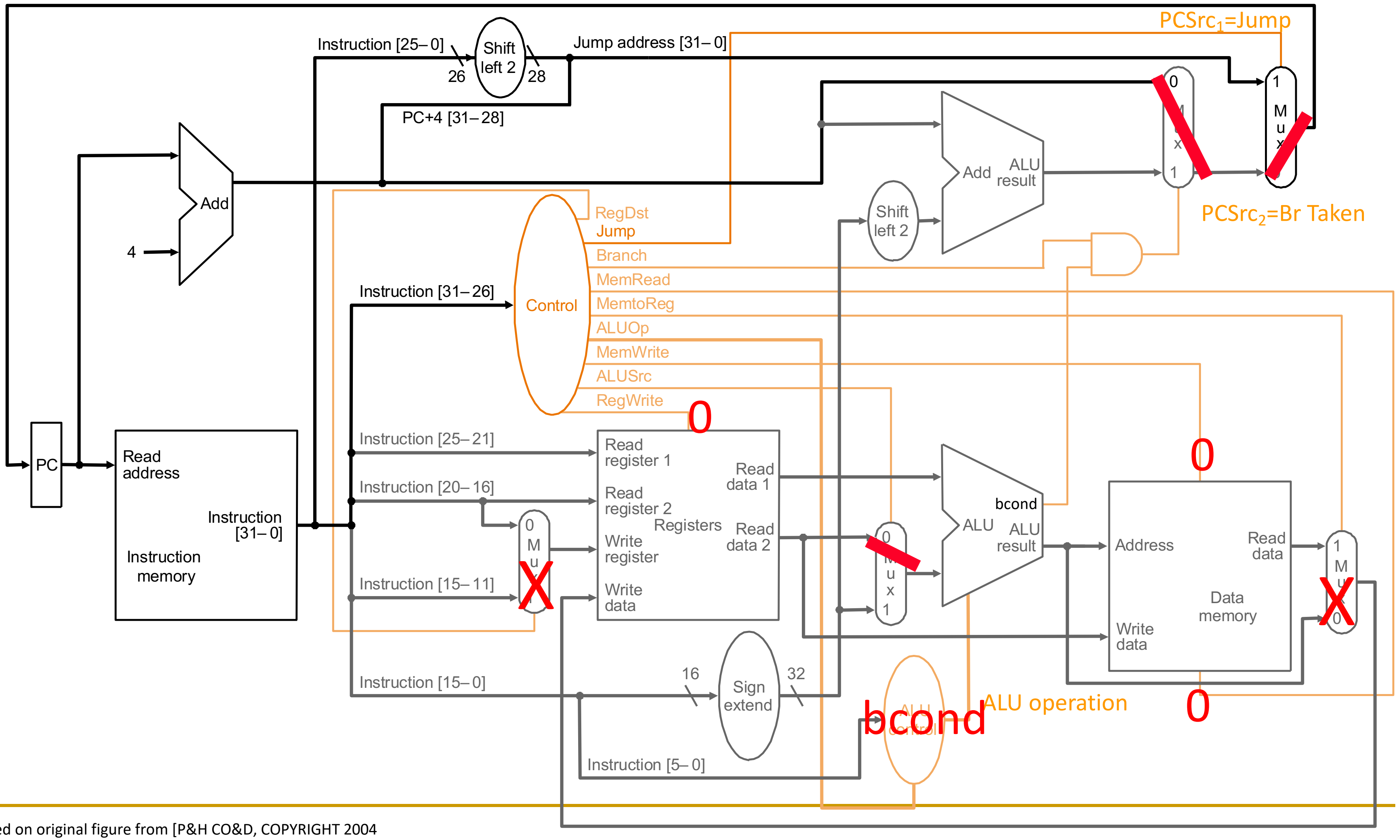
SW



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

Branch (Not Taken)

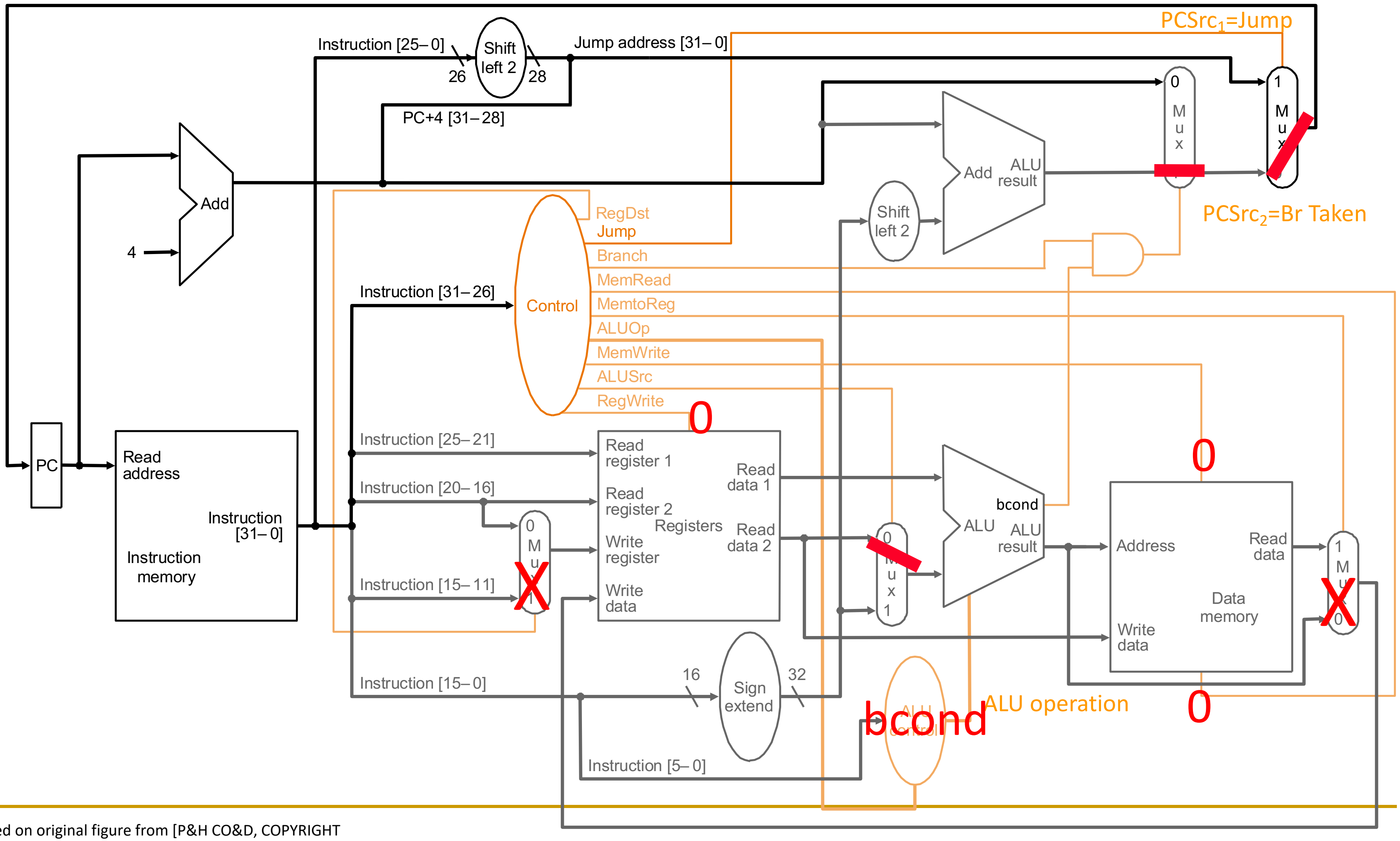
Some control signals are dependent on the processing of data



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

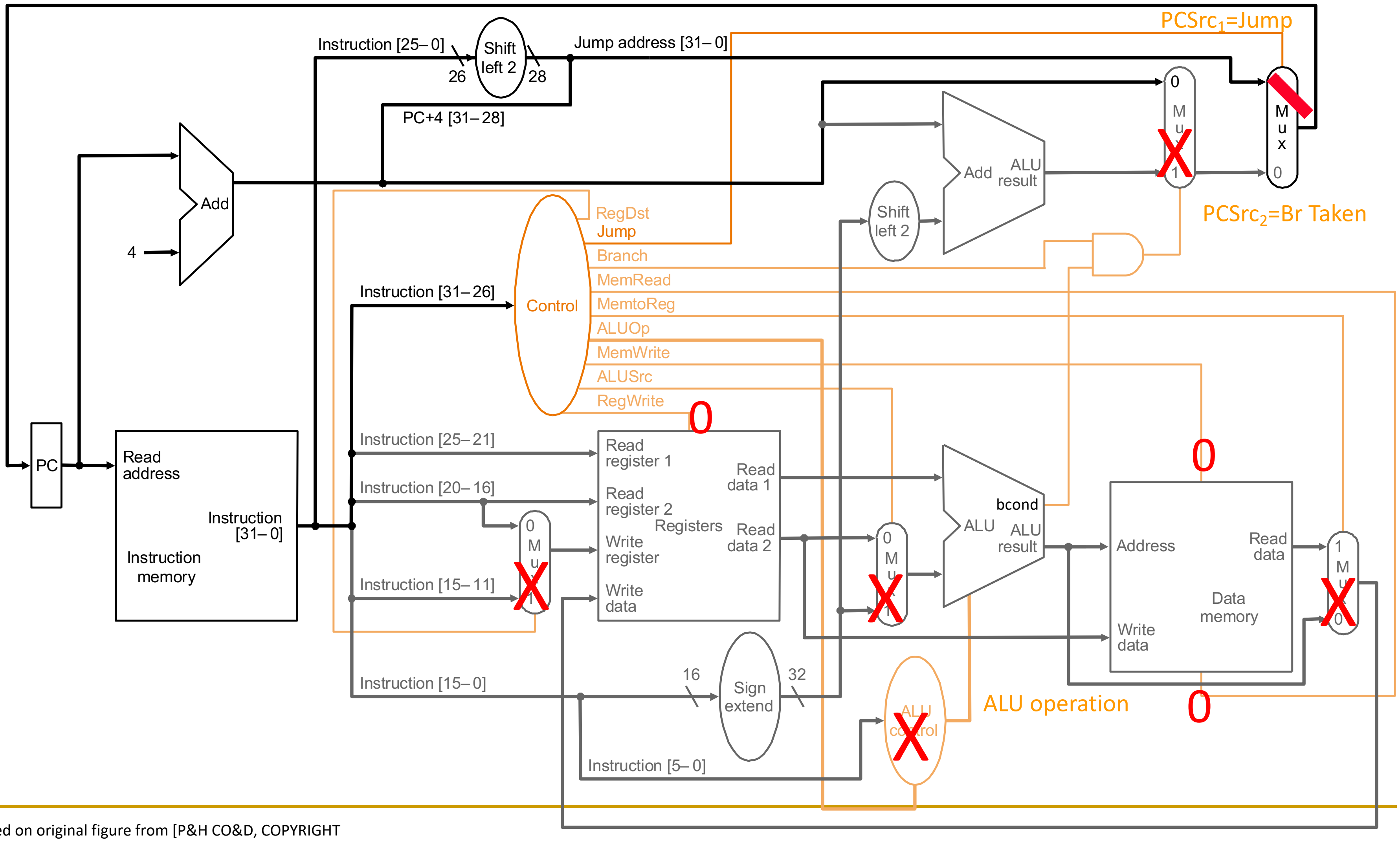
Branch (Taken)

Some control signals are dependent on the processing of data



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

Jump



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

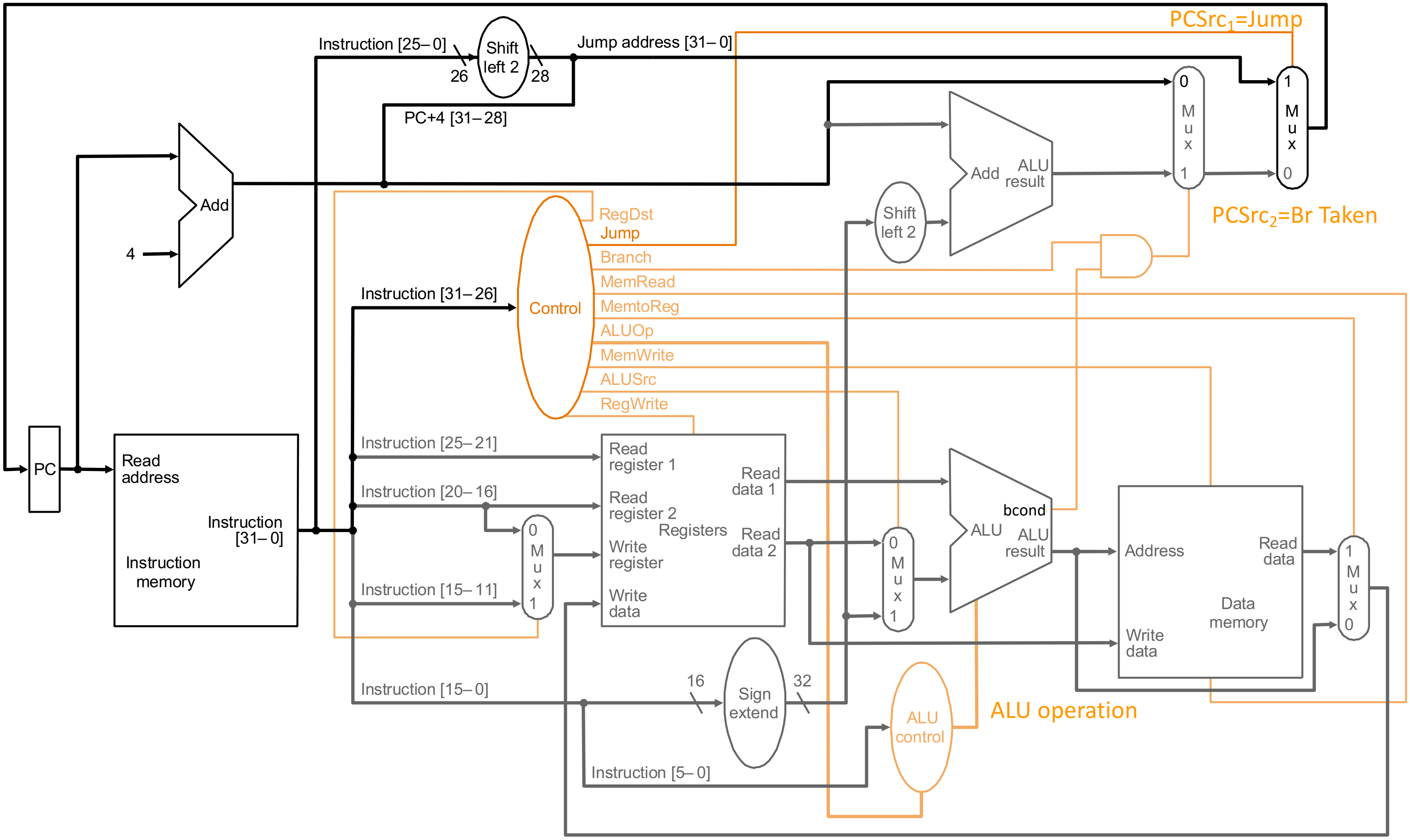
What is in That Control Box?

- Combinational Logic → Hardwired Control
  - Idea: Control signals generated combinationally based on bits in instruction encoding

- Sequential Logic → Sequential Control
  - Idea: A memory structure contains the control signals associated with an instruction
    - Called Control Store

- Both types of control structure can be used in single-cycle processors
  - Choice depends on latency of each structure + how much on the critical path control signal generation is, etc.

Review: Complete Single-Cycle Processor



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

JAL, JR, JALR omitted

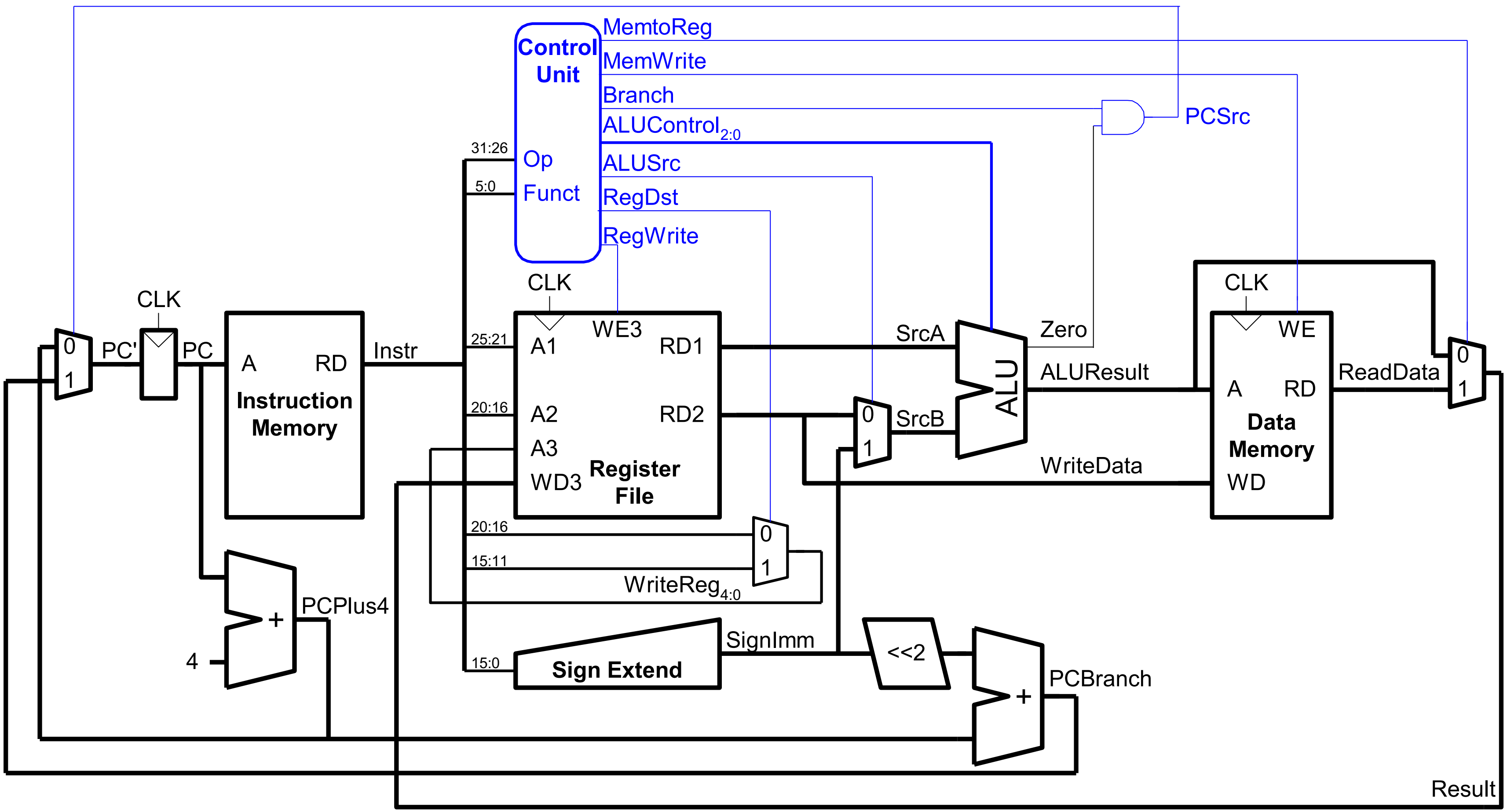
Another Single-Cycle MIPS Processor (from H&H)

See backup slides to reinforce the concepts we have covered.

They are to complement your reading:

H&H, Chapter 7.1-7.3, 7.6

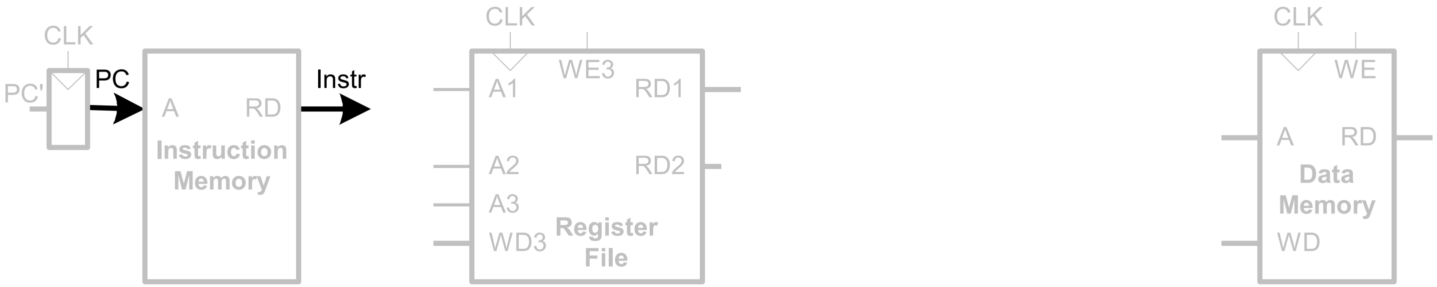
Another Complete Single-Cycle Processor



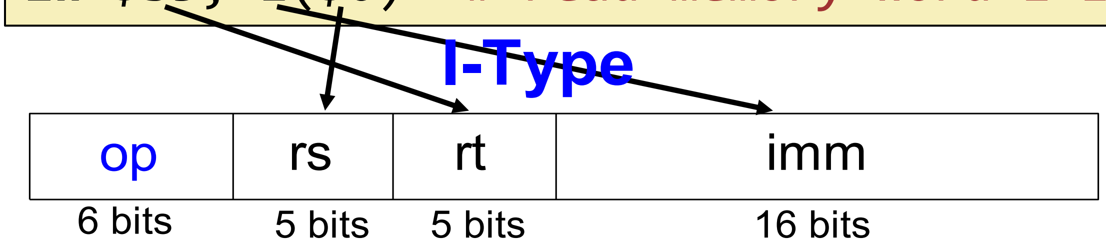
Single-cycle processor. Harris and Harris, Chapter 7.3.

Example: Single-Cycle Datapath: `lw` fetch

- *STEP 1:* **Fetch instruction**



```
lw $s3, 1($0)   # read memory word 1 into $s3
```

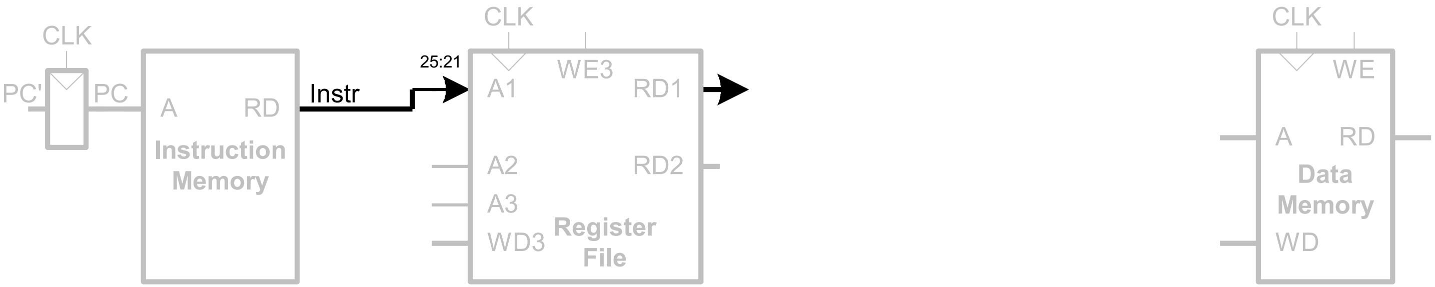


I-Type

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

Single-Cycle Datapath: `lw` register read

- ***STEP 2:*** **Read source operands from register file**



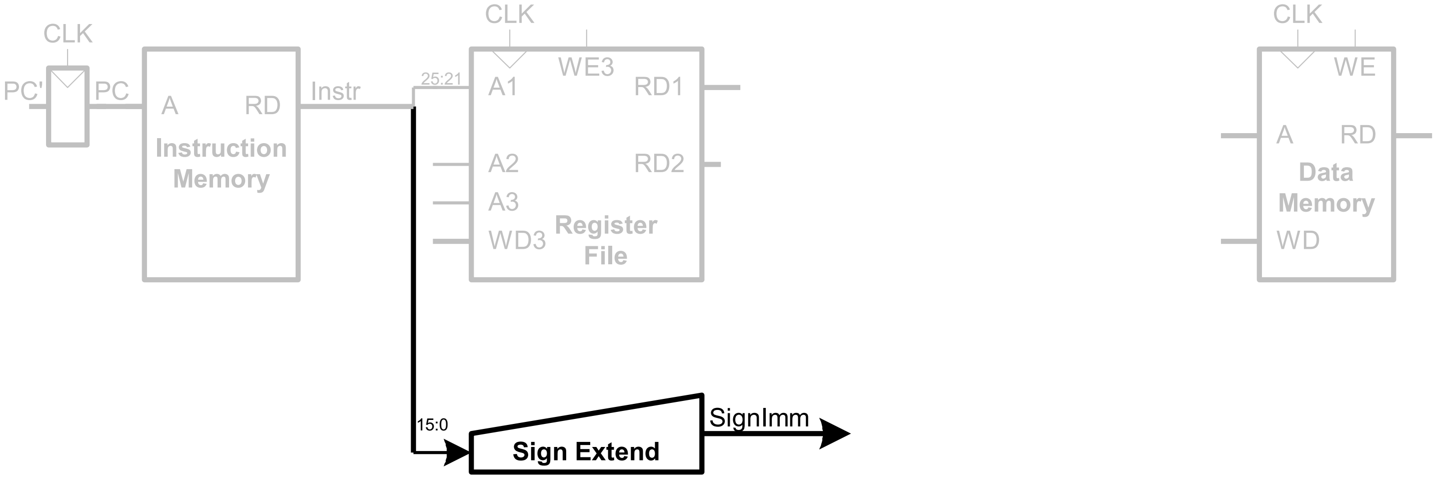
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` immediate

- ***STEP 3:*** **Sign-extend the immediate**



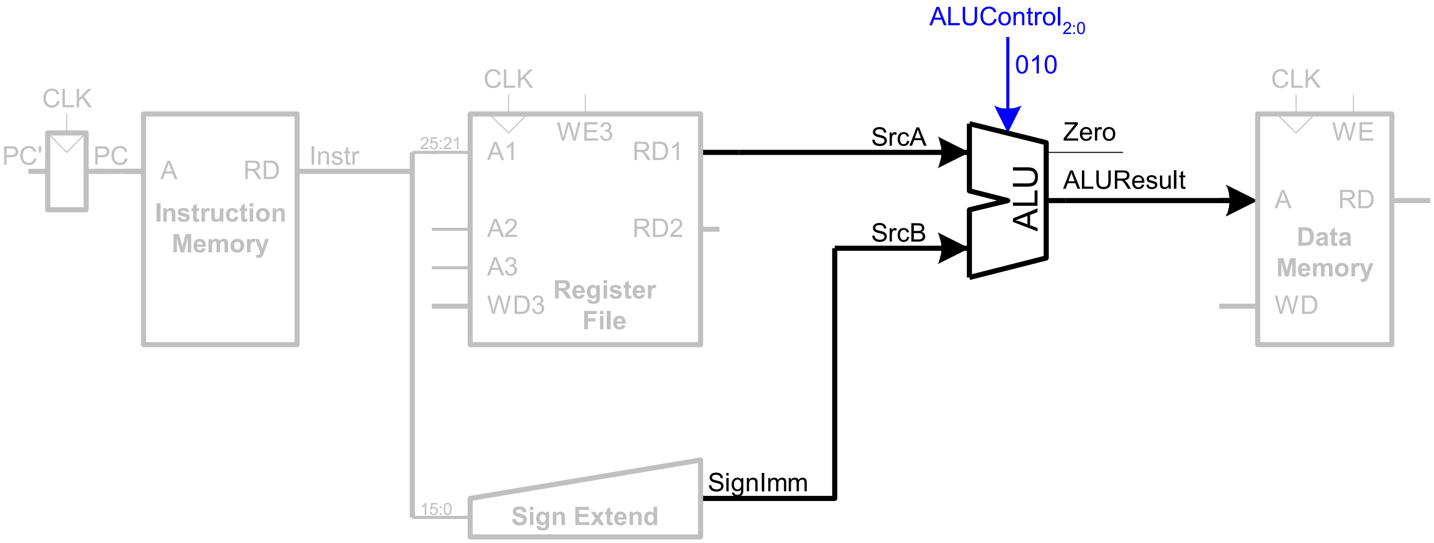
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` address

- *STEP 4:* **Compute the memory address**



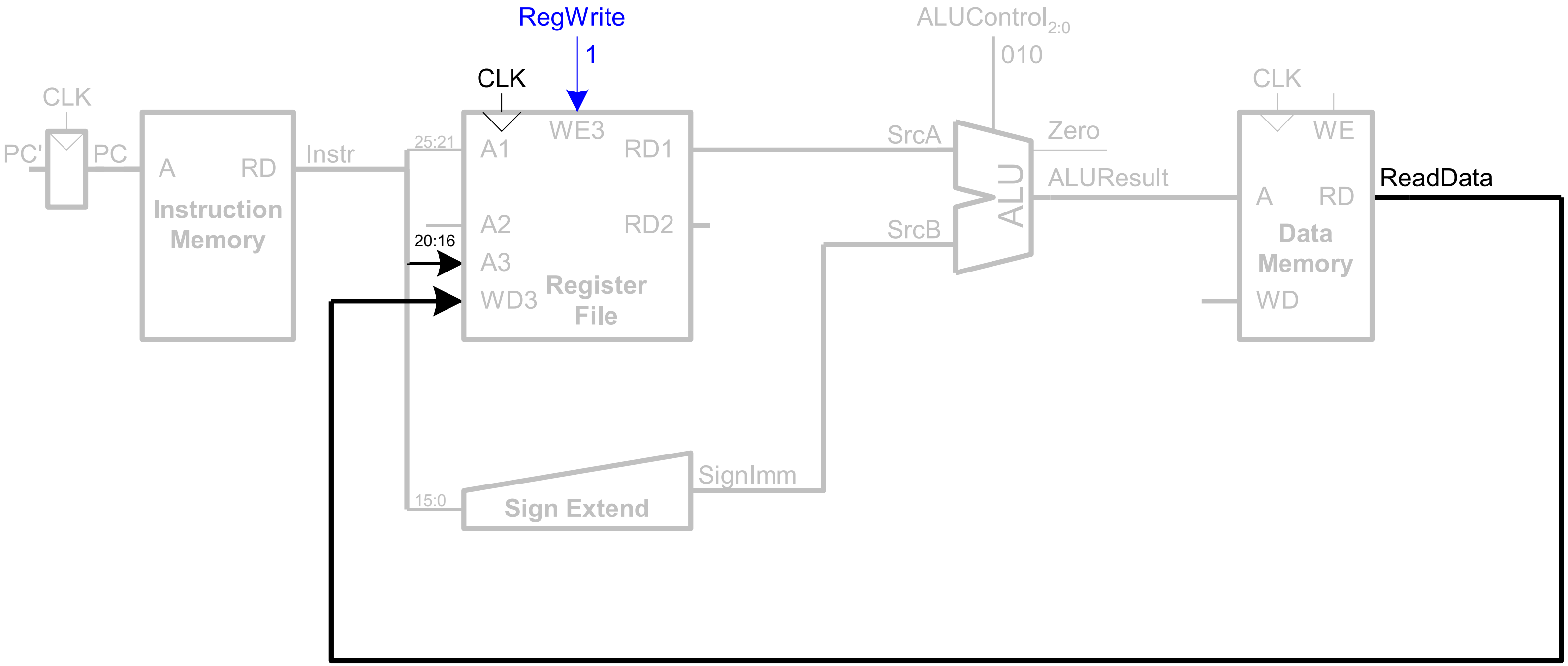
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` memory read

- ***STEP 5:*** **Read from memory and write back to register file**



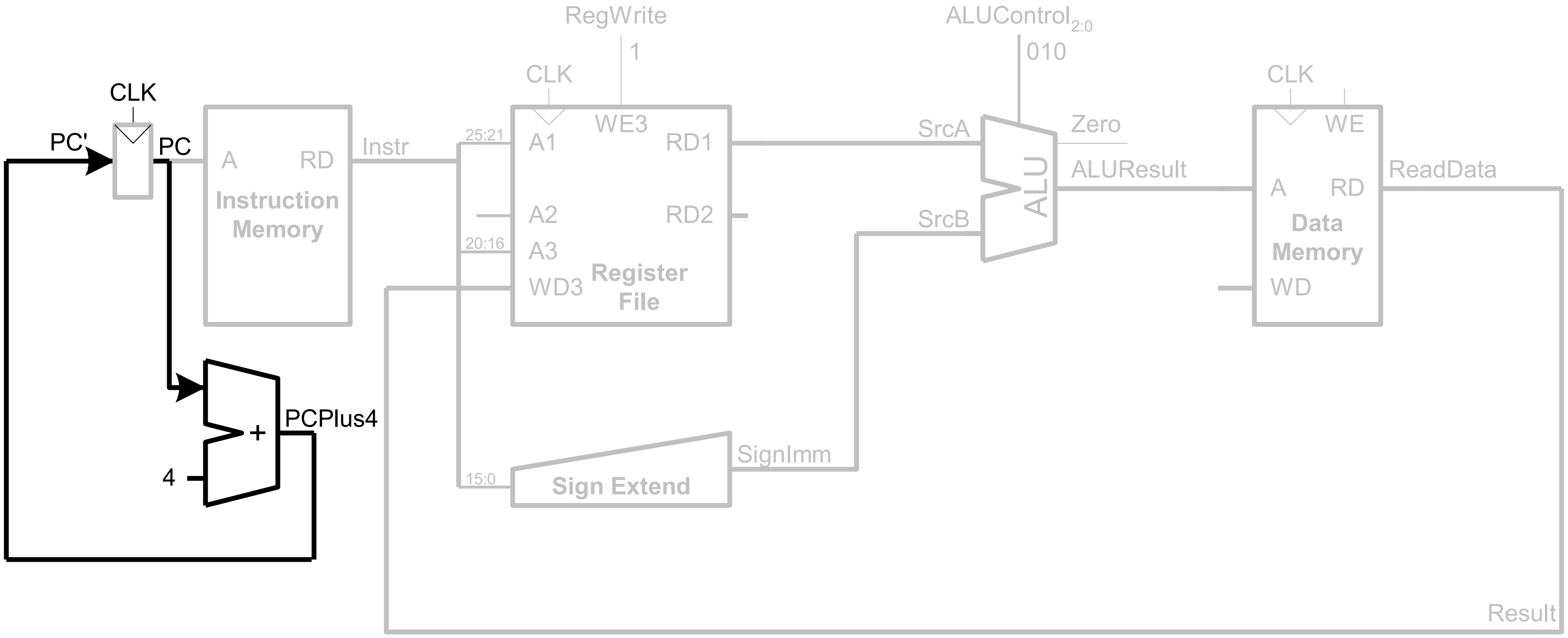
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` PC increment

- *STEP 6:* **Determine address of next instruction**



```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

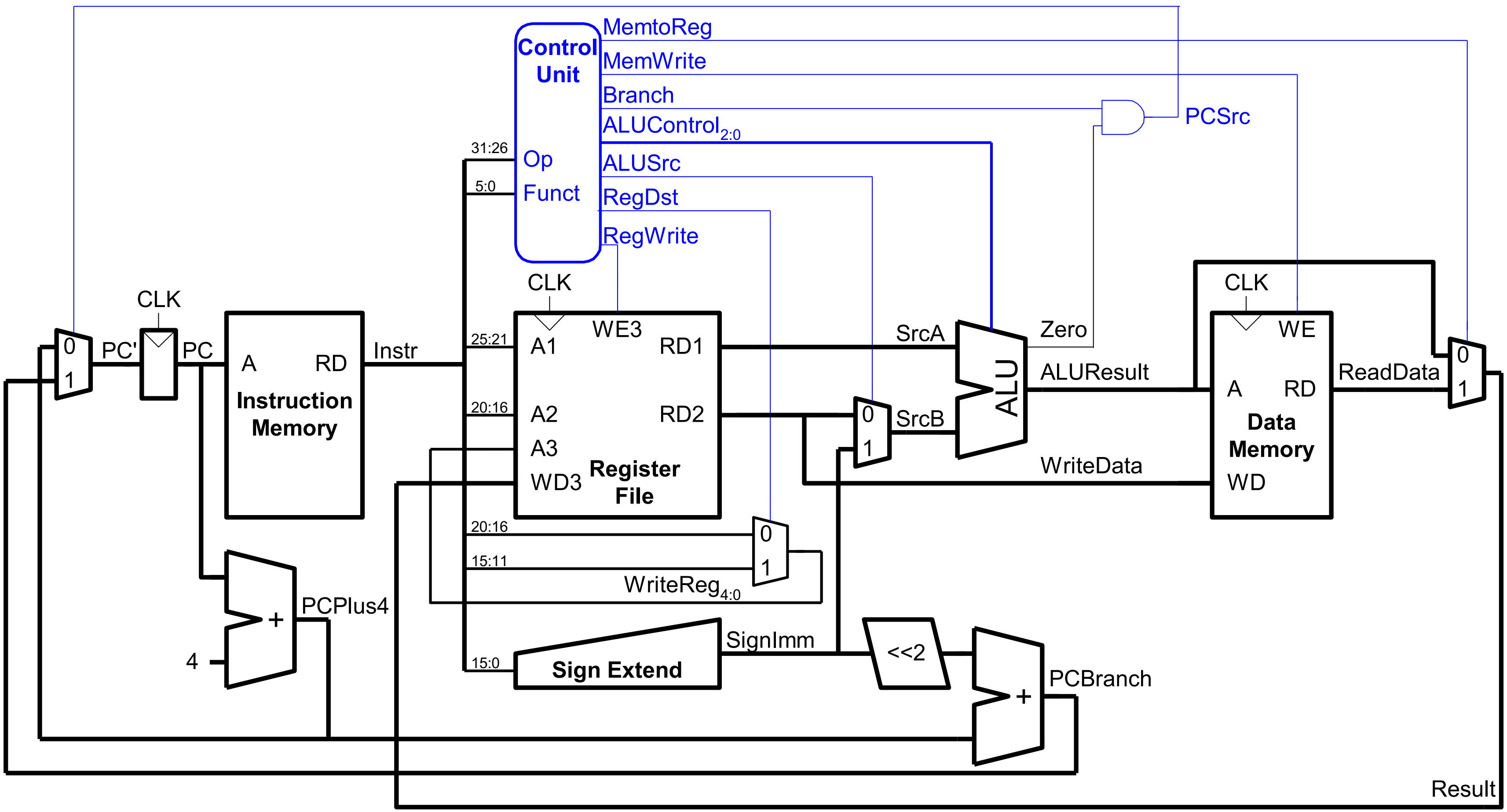
Similarly, We Need to Design the Control Unit

- Control signals are generated by the decoder in control unit

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

Single-cycle processor. Harris and Harris, Chapter 7.3.

Another Complete Single-Cycle Processor (H&H)

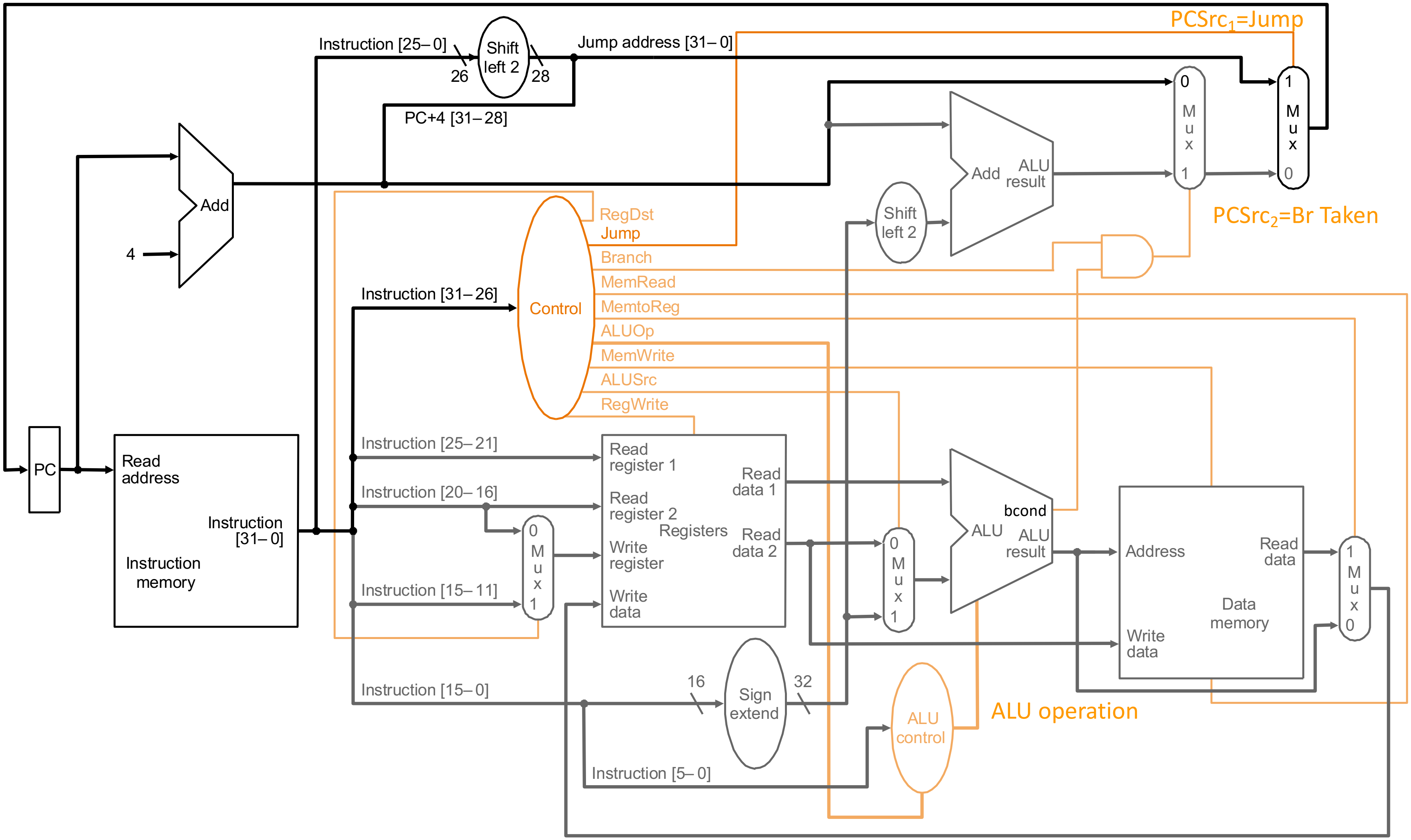


Your Reading Assignment

- Please read the Lecture Slides & the Backup Slides

- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6

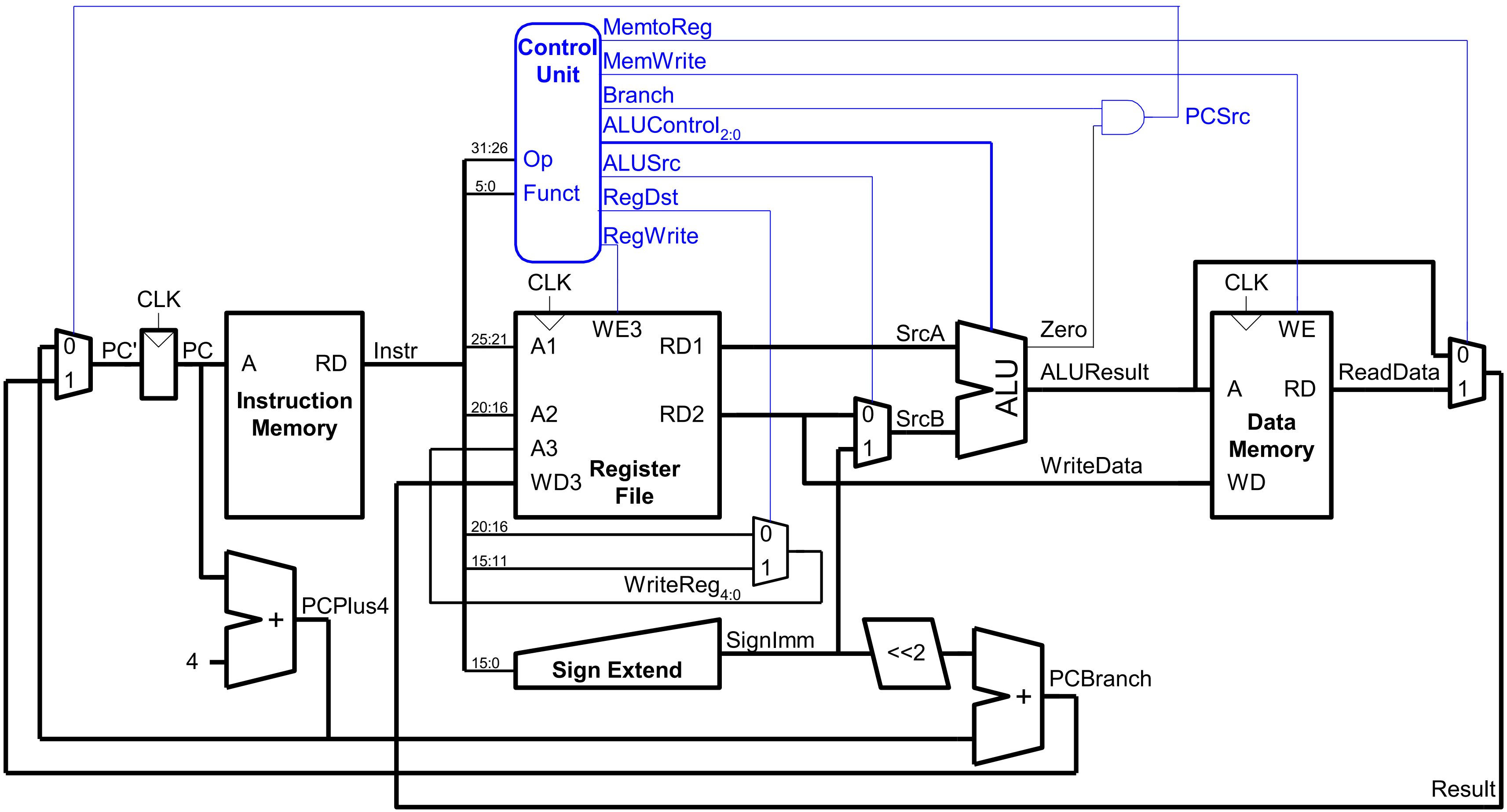
Single-Cycle Uarch I (We Developed in Lectures)



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

JAL, JR, JALR omitted

Single-Cycle Uarch II (In Your Readings)



Single-cycle processor. Harris and Harris, Chapter 7.3.

Evaluating the Single-Cycle Microarchitecture

A Single-Cycle Microarchitecture

- Is *this* a good idea/design?
- When is this a good design?
- When is this a bad design?
- How can we design a better microarchitecture?

Performance Analysis Basics

Recall: Performance Analysis Basics

- Execution time of a single instruction
  - **{CPI} x {clock cycle time}**
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of an entire program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

- **How fast are my instructions?**
  - Instructions are realized on the hardware
  - Each instruction can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed

- **How fast are my instructions?**
  - Instructions are realized on the hardware
  - Each instruction can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

- **How long is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second

Processor Performance

- **As a general formula**
  - Our program consists of executing **N** instructions
  - Our processor needs **CPI** cycles (on average) for each instruction
  - The clock frequency of the processor is **f**

    → the clock period is therefore **T**=1/f

Processor Performance

- **As a general formula**
  - Our program consists of executing **N** instructions
  - Our processor needs **CPI** cycles (on average) for each instruction
  - The clock frequency of the processor is **f**

    → the clock period is therefore **T**=1/f

- **Our program executes in**

N x CPI x (1/f) =

N x CPI x T seconds

Performance Analysis of Our Single-Cycle Design

A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

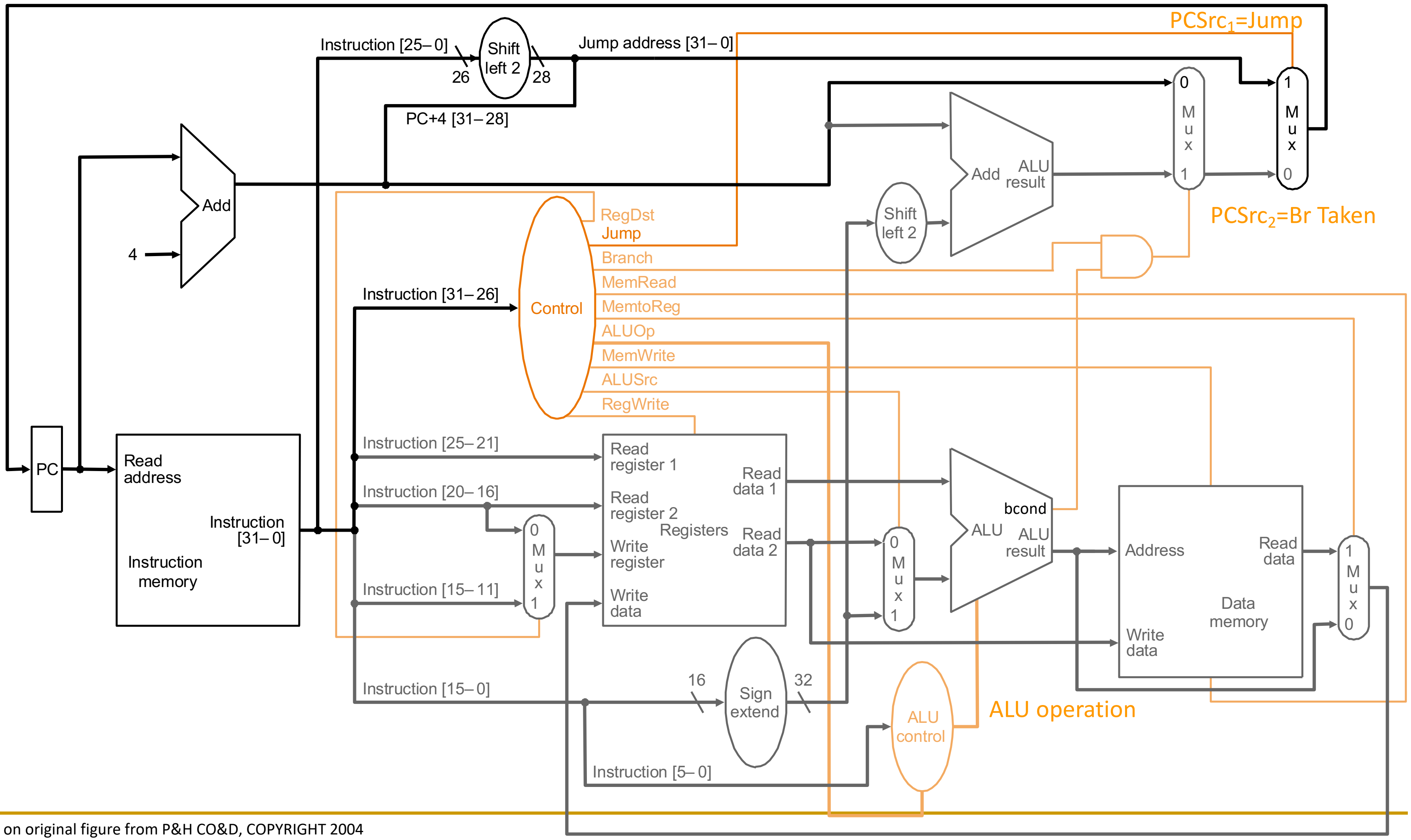
- Let’s go back to the basics

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

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

- Do each of the above phases take the same time (latency) for all instructions?

Let’s Find the Critical Path



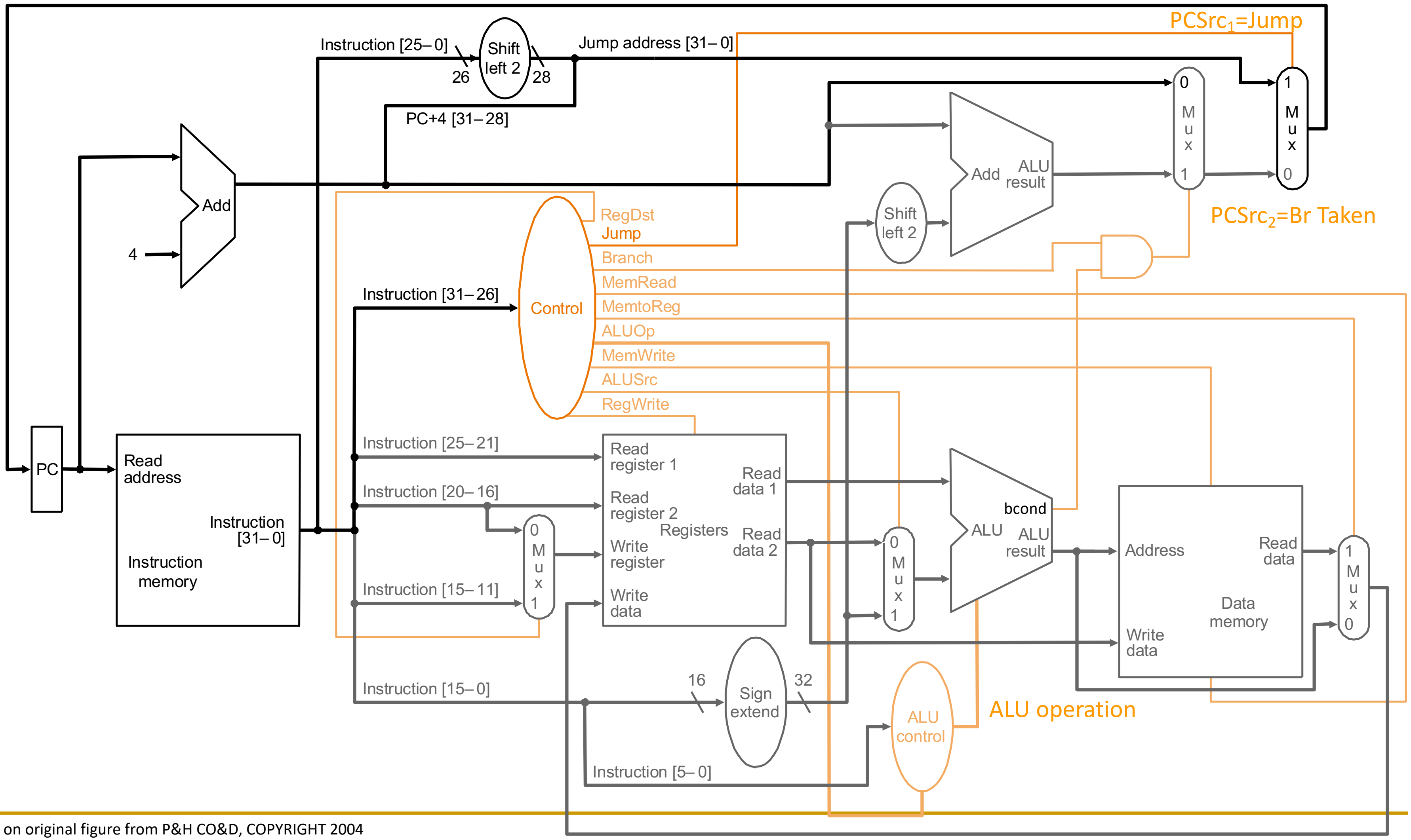
[Based on original figure from P&H CO&D, COPYRIGHT 2004
Elsevier. ALL RIGHTS RESERVED.]

Example Single-Cycle Datapath Analysis

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

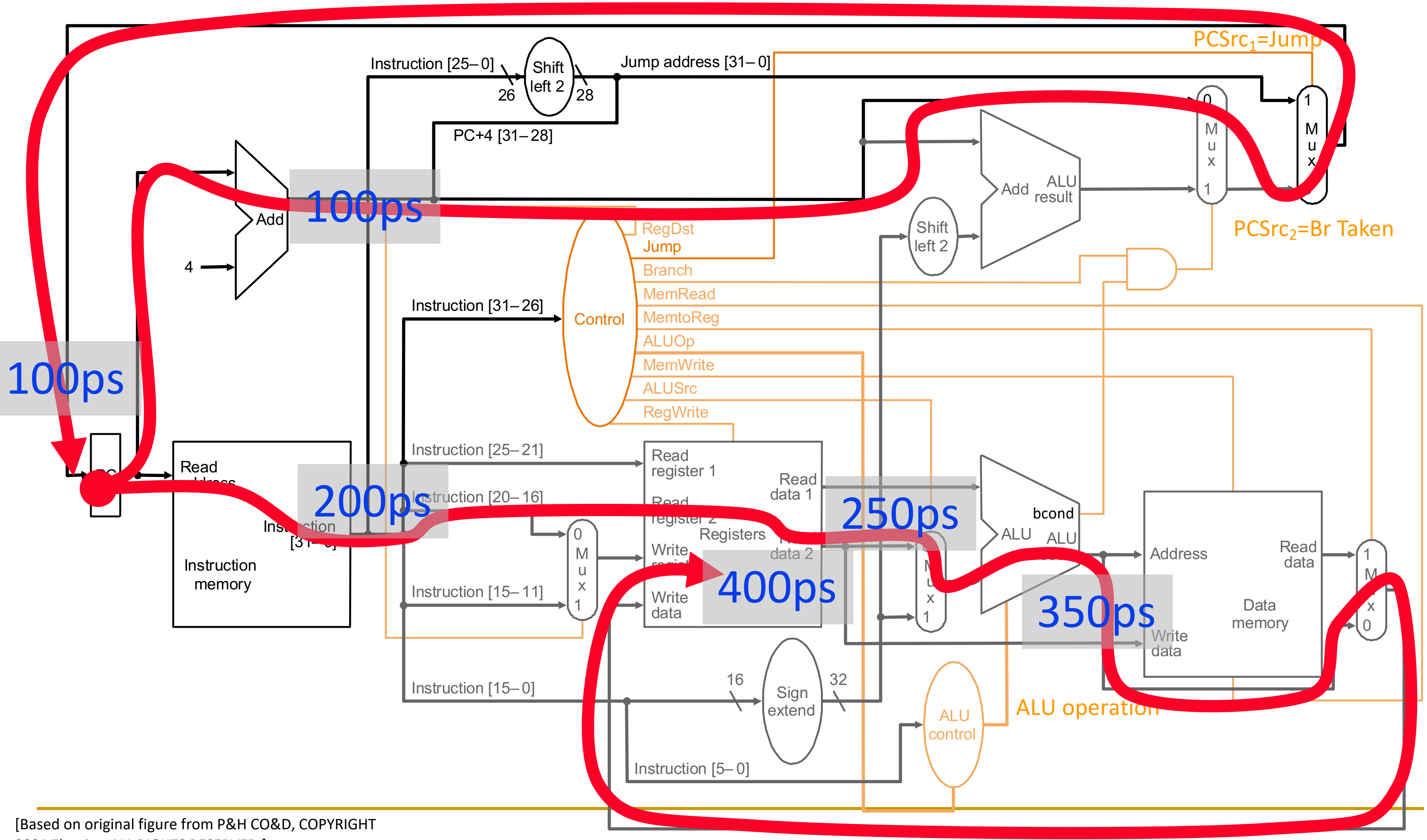
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

Let's Find the Critical Path



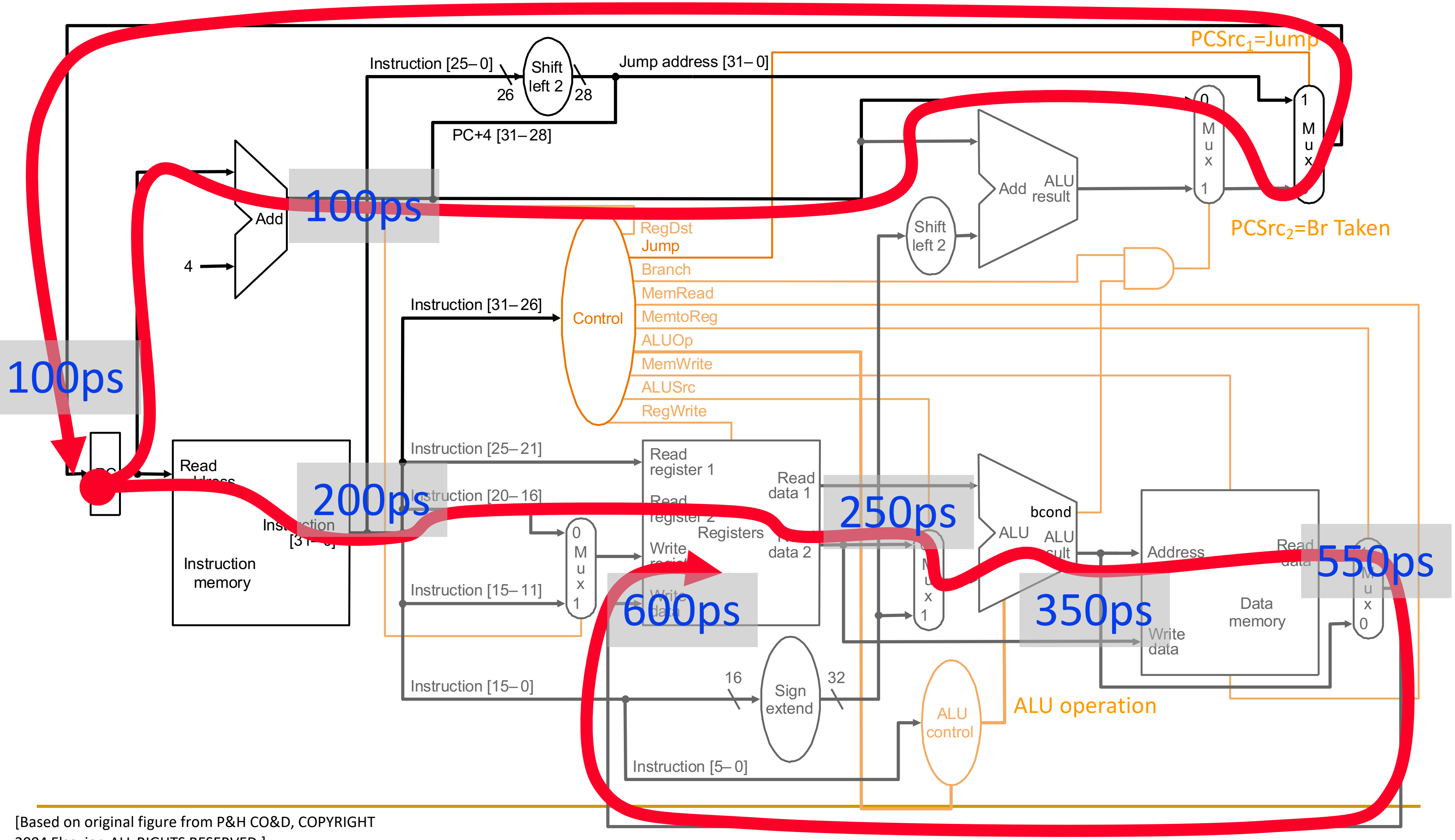
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

R-Type and I-Type ALU



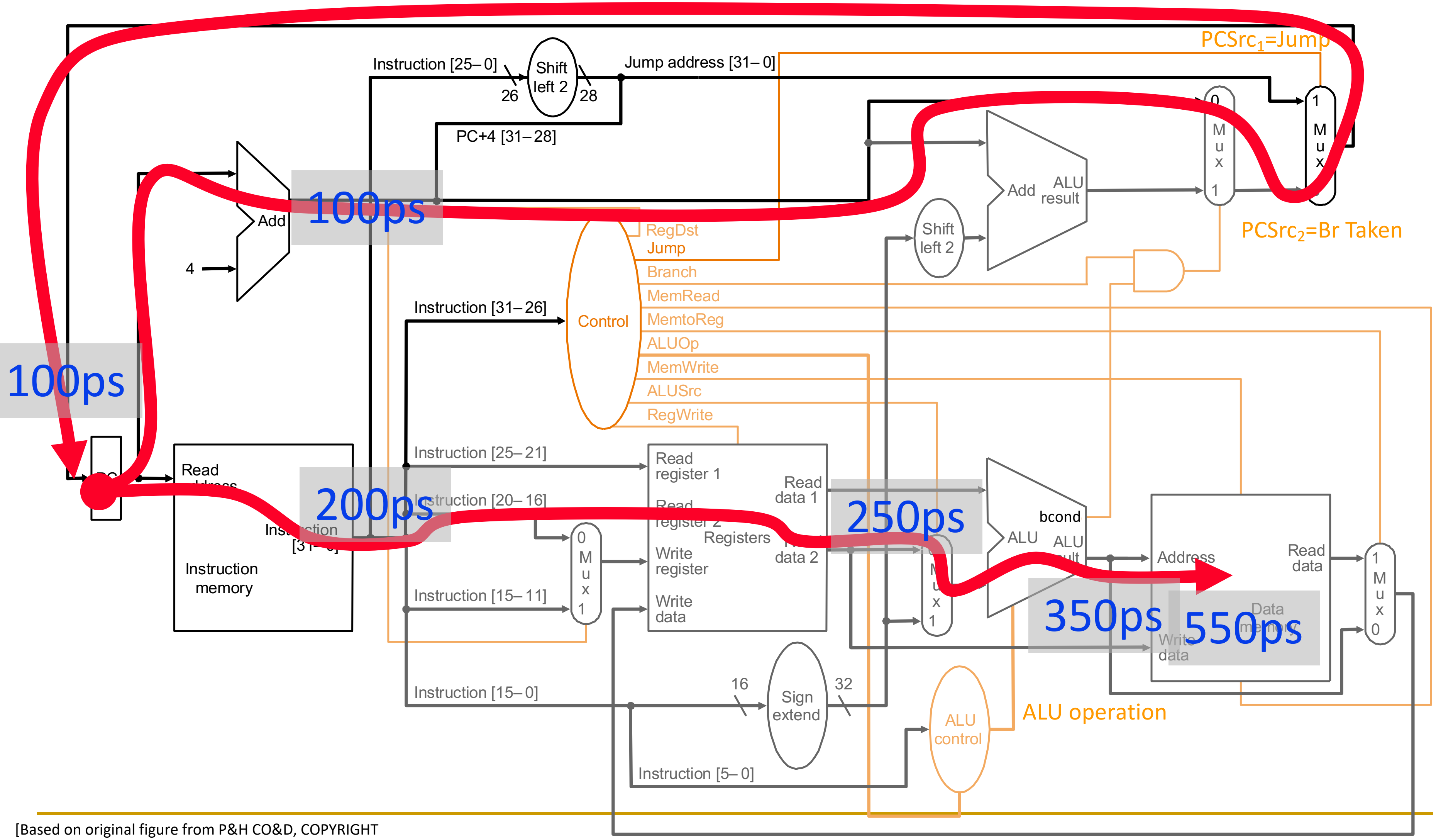
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



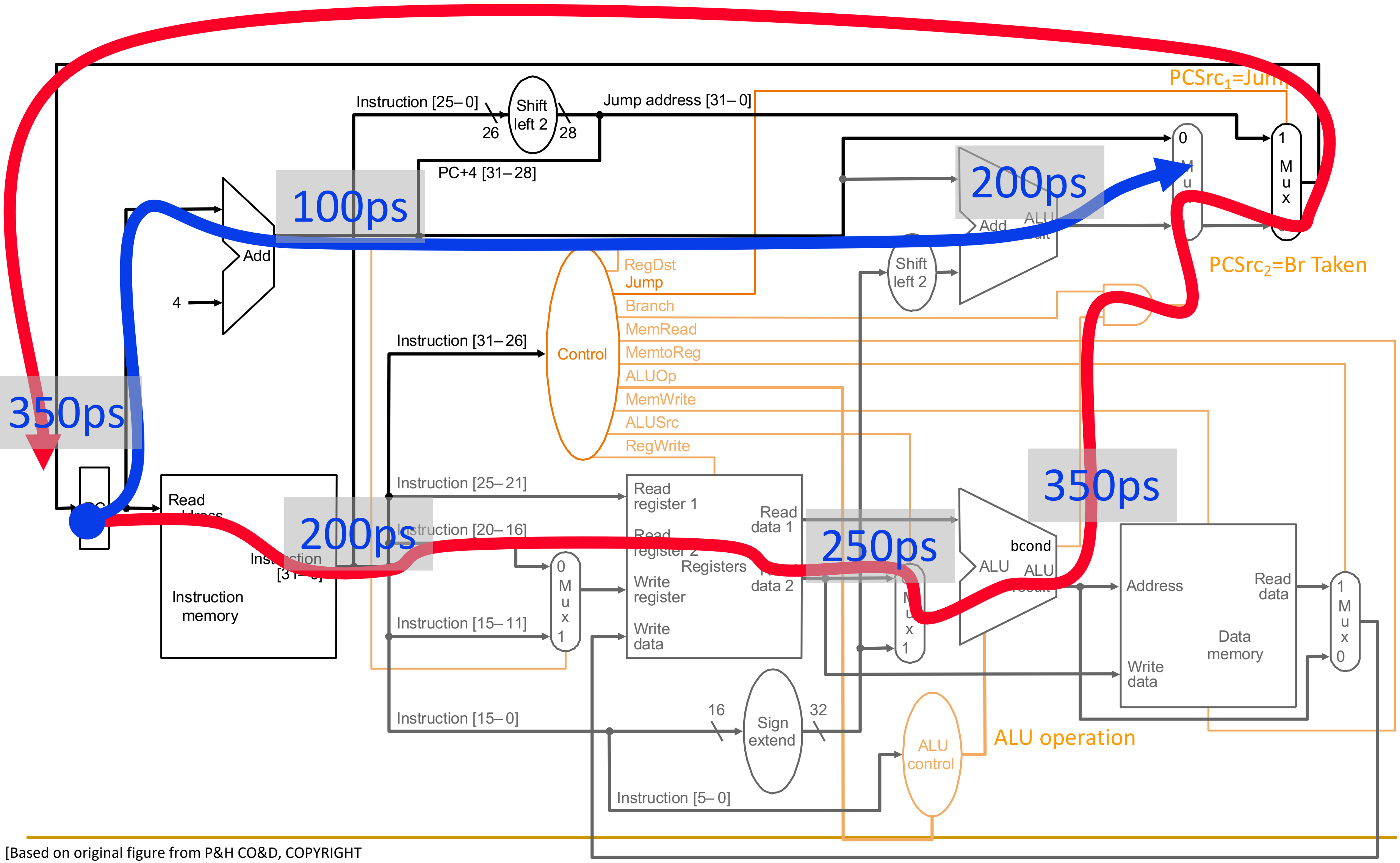
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

SW



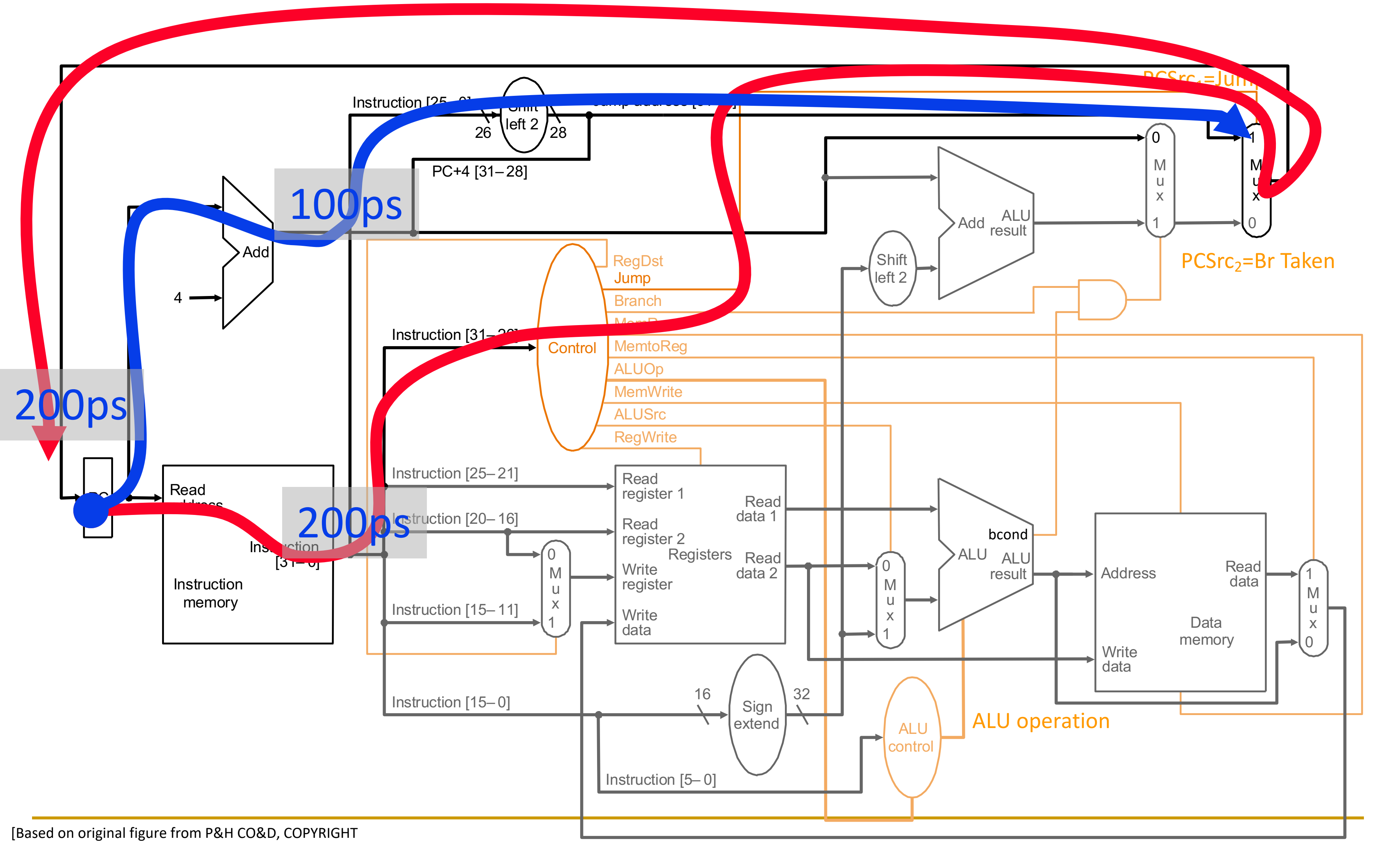
[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Branch Taken



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

Jump



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

What About Control Logic?

- How does that affect the critical path?

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

What is the Slowest Instruction to Process?

- Real world: **Memory is slow (not magic)**

- What if memory *sometimes* takes 100ms to access?

- Does it make sense to have a simple register to register add or jump to take {100ms+all else to do a memory operation}?

- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?

Single Cycle uArch: Complexity

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

(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long

- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

Single-Cycle Design vs. Design Principles

- Critical path design

- Bread and butter (common case) design

- Balanced design

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

Aside: System Design Principles

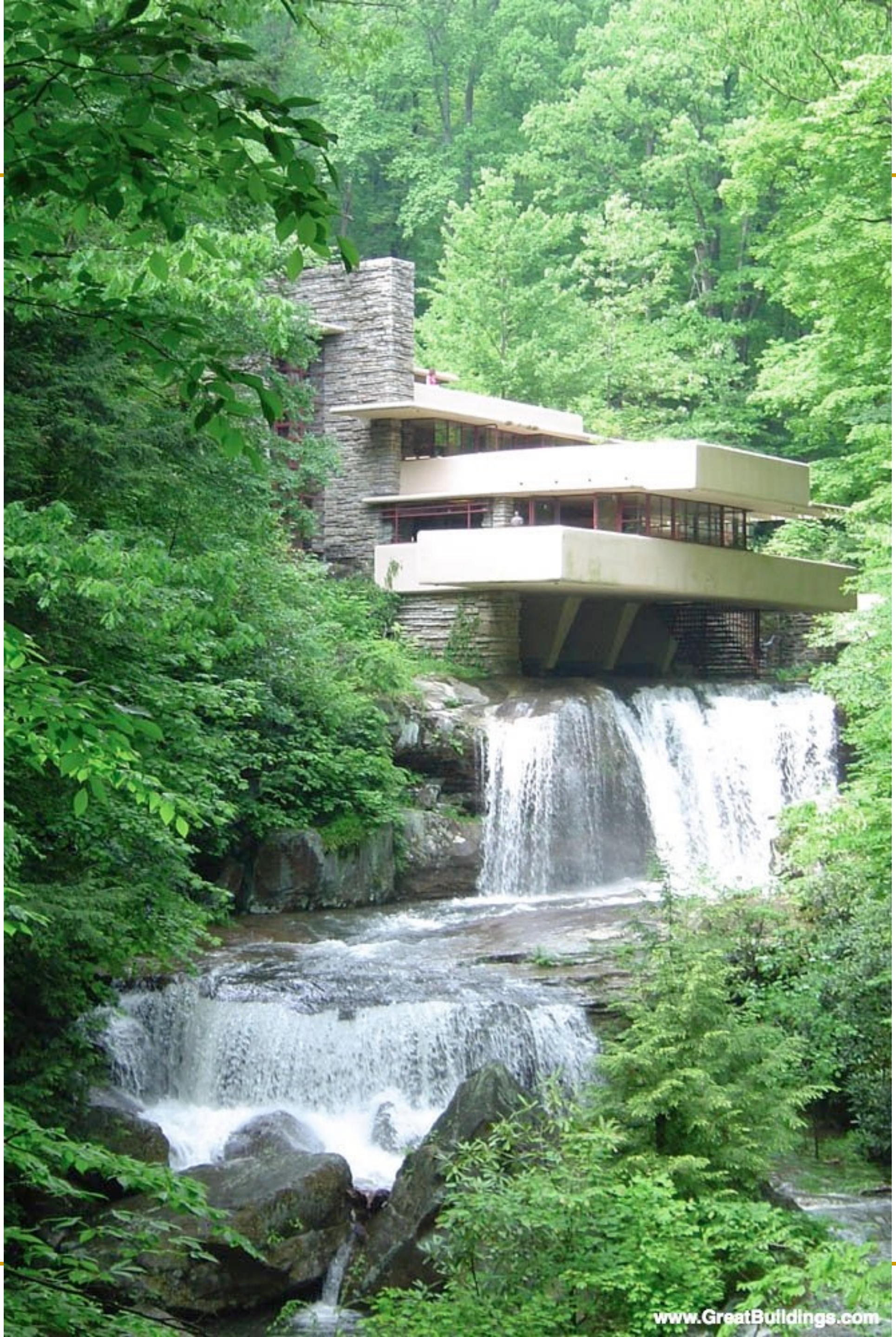
- When designing computer systems/architectures, it is important to follow good principles
  - Actually, this is true for *any* system design
    - Real architectures, buildings, bridges, …
    - Good consumer products
    - Mechanisms for security/safety-critical systems
    - …

- Remember: “principled design” from our second lecture
  - Frank Lloyd Wright: “architecture […] based upon principle, and not upon precedent”

Aside: From Lecture 2

- “architecture […] based upon principle, and not upon precedent”

This



That

Recall: Takeaways

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

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

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

Aside: System Design Principles

- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Yale Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proc. of IEEE, 2001. (Levels of transformation, design point, etc)

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966. (Flynn's Bottleneck → Balanced design)

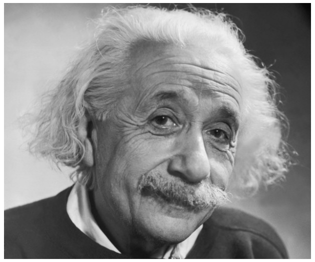
- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl's Law → Common-case design)

- Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.

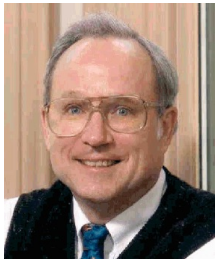
A Key System Design Principle

- Keep it simple

- "Everything should be made as simple as possible, but no simpler."
  - Albert Einstein



- And, keep it low cost: "An engineer is a person who can do for a dime what any fool can do for a dollar."



- For more, see:
  - Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

Can We Do Better?

Multi-Cycle Microarchitectures

Backup Slides on Single-Cycle Uarch for Your Own Study

Please study these to reinforce the concepts
we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6

Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study.
They are to complement your reading
H&H, Chapter 7.1-7.3, 7.6

What to do with the Program Counter?

- **The PC needs to be incremented by 4 during each cycle (for the time being).**
- **Initial PC value (after reset) is `0x00400000`**

```
reg [31:0] PC_p, PC_n;      // Present and next state of PC

// […]

  assign PC_n <= PC_p + 4;                   // Increment by 4;

  always @ (posedge clk, negedge rst)
    begin
       if (rst == ‘0’) PC_p <= 32’h00400000; // default
       else            PC_p <= PC_n;         // when clk
    end
```

We Need a Register File

- **Store 32 registers, each 32-bit**
  - 2^5 == 32, we need 5 bits to address each
- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)
- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)

Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description
  assign do_rs = R_arr[a_rs];          // Read RS


  assign do_rt = R_arr[a_rt];          // Read RT


  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

Register File

```
input [4:0]   a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input         we_rd;
output [31:0] do_rs, do_rt;

  reg [31:0] R_arr [31:0]; // Array that stores regs

  // Circuit description; add the trick with $0
  assign do_rs = (a_rs != 5’b00000)?   // is address 0?
                  R_arr[a_rs] : 0;     // Read RS or 0

  assign do_rt = (a_rt != 5’b00000)?   // is address 0?
                  R_arr[a_rt] : 0;     // Read RT or 0

  always @ (posedge clk)
      if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

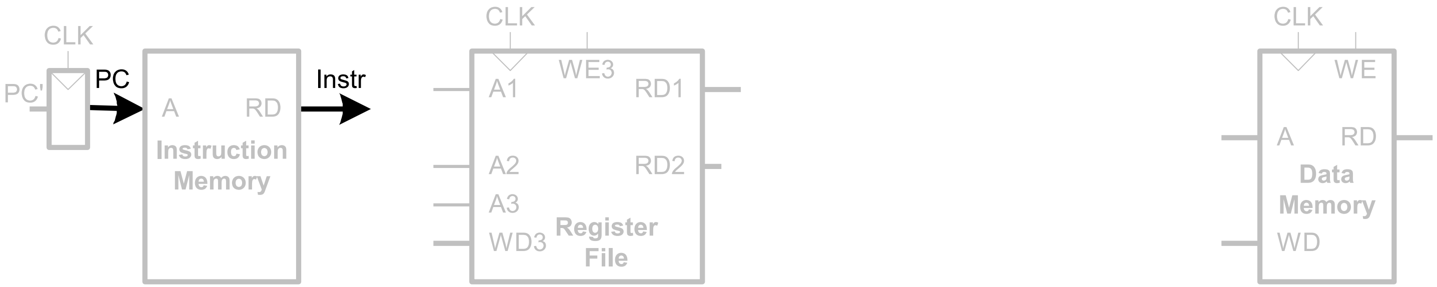
Data Memory Example

- **Will be used to store the bulk of data**

```
input [15:0]  addr; // Only 16 bits in this example
input [31:0]  di;
input         we;
output [31:0] do;

  reg [31:0] M_arr [0:65535];         // Array for Memory

  // Circuit description
  assign do = M_arr[addr];            // Read memory

  always @ (posedge clk)
      if (we) M_arr[addr] <= di;      // write memory
```

Single-Cycle Datapath: `lw` fetch

- *STEP 1:* **Fetch instruction**



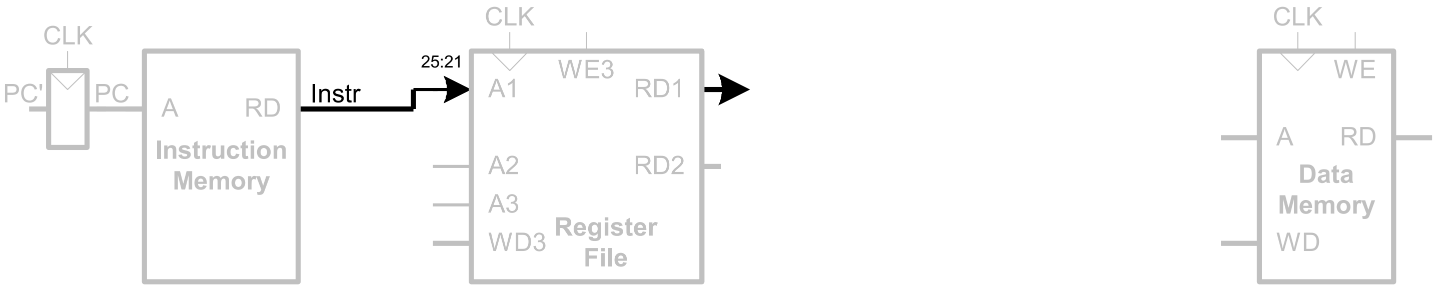
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` register read

- *STEP 2:* **Read source operands from register file**



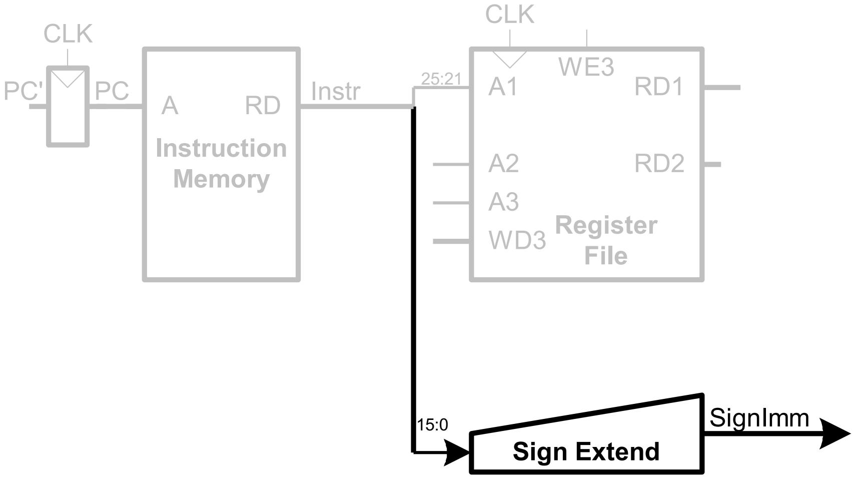
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` immediate

- ***STEP 3:*** **Sign-extend the immediate**



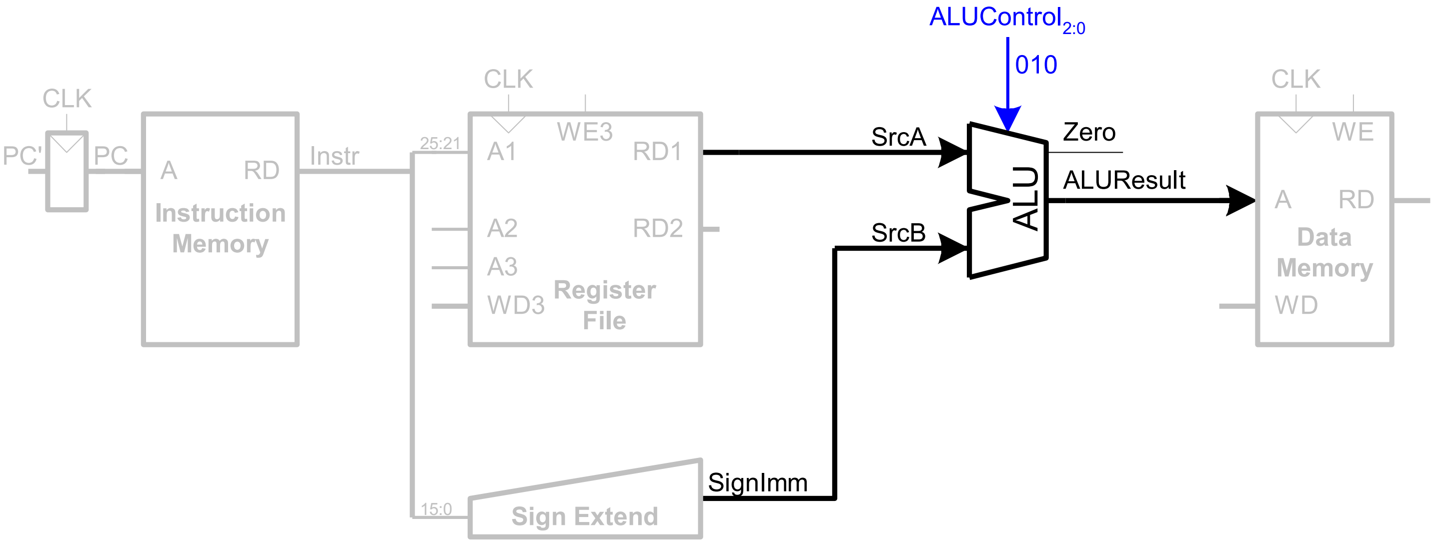
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` address

- *STEP 4:* Compute the memory address



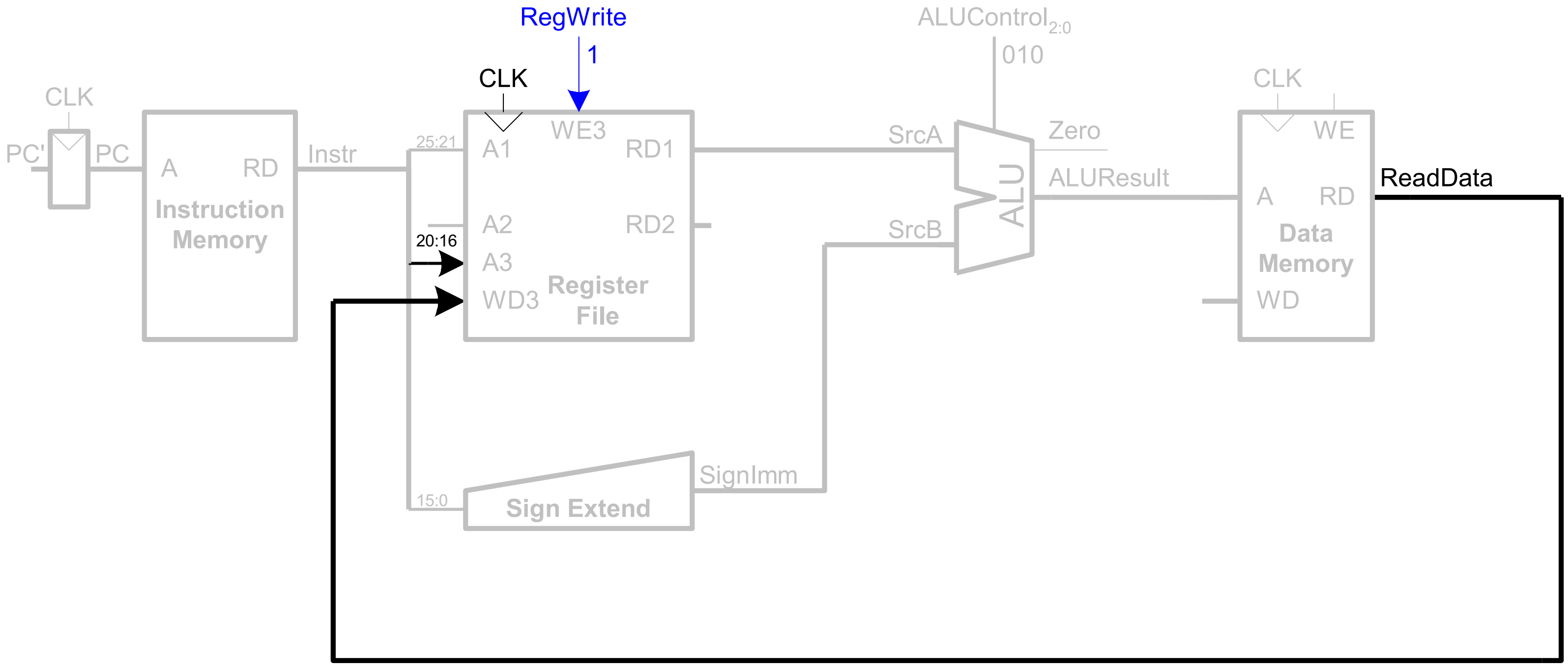
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` memory read

- ***STEP 5:*** **Read from memory and write back to register file**



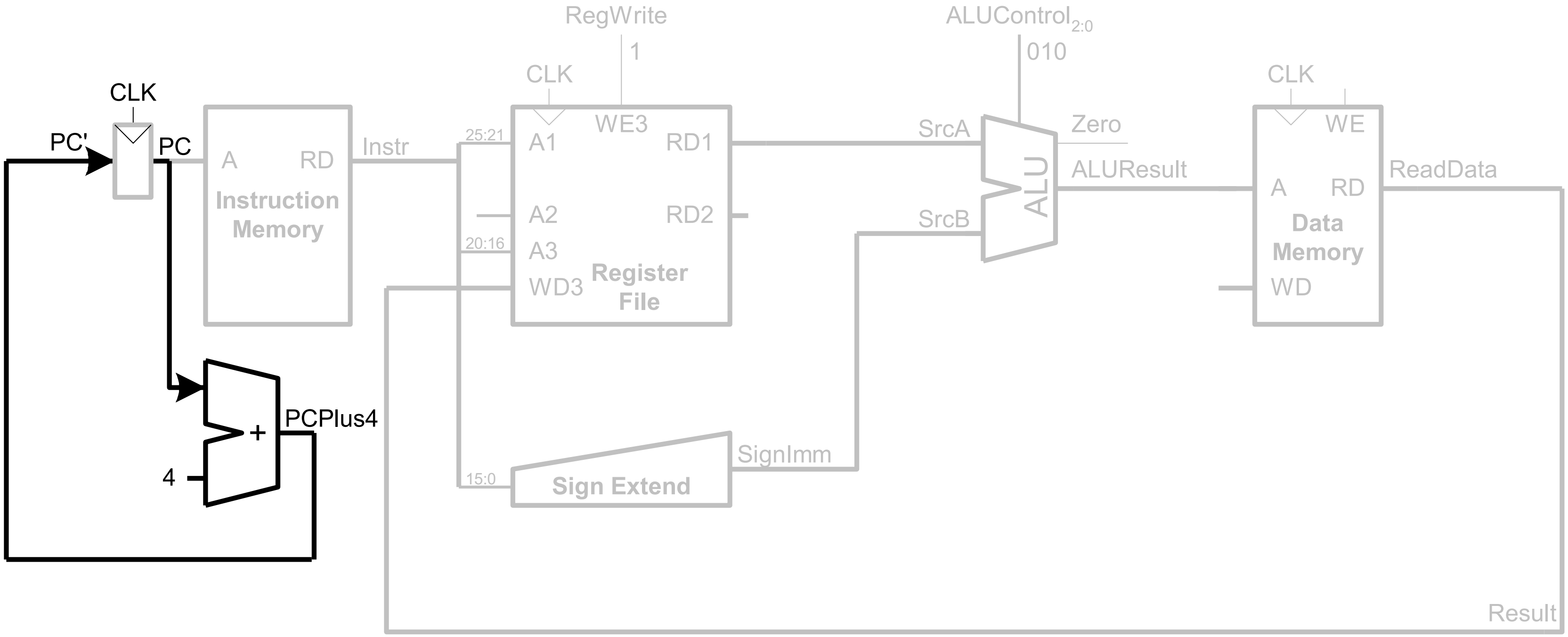
```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` PC increment

- ***STEP 6:*** **Determine address of next instruction**



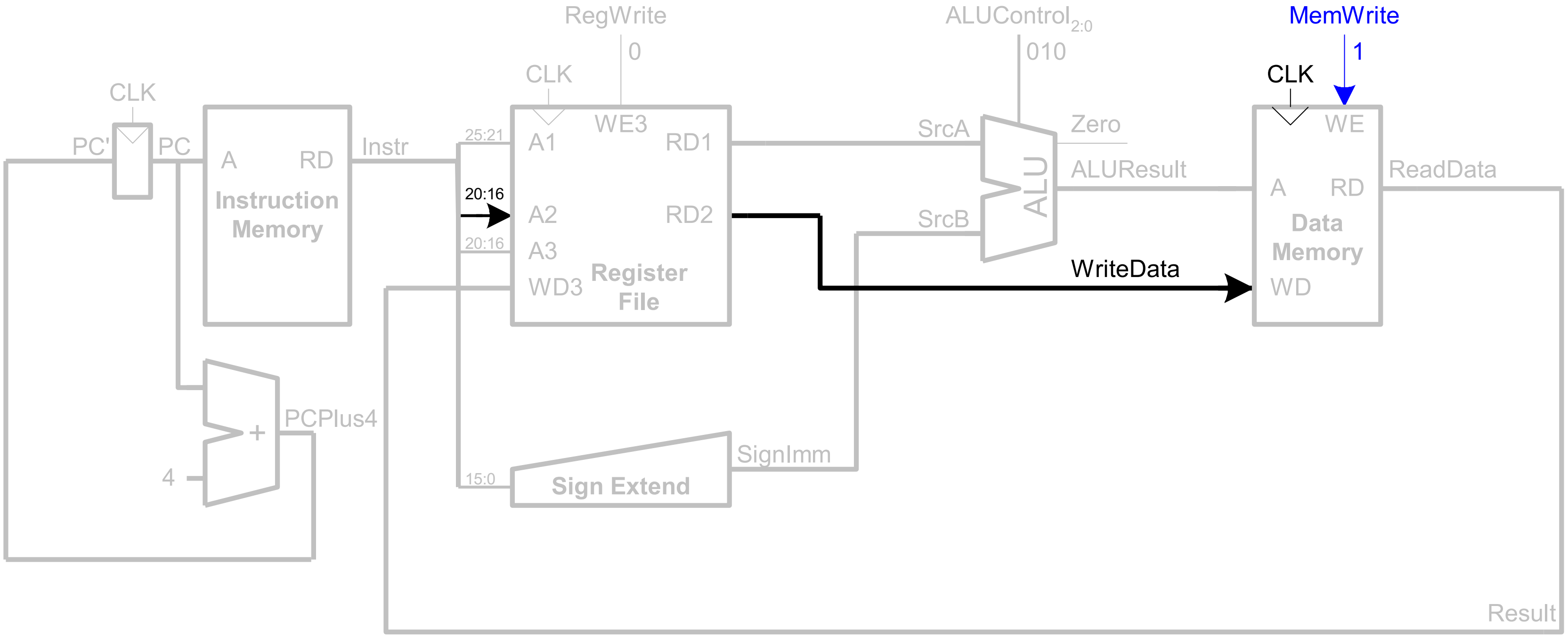
```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: sw

- Write data in rt to memory



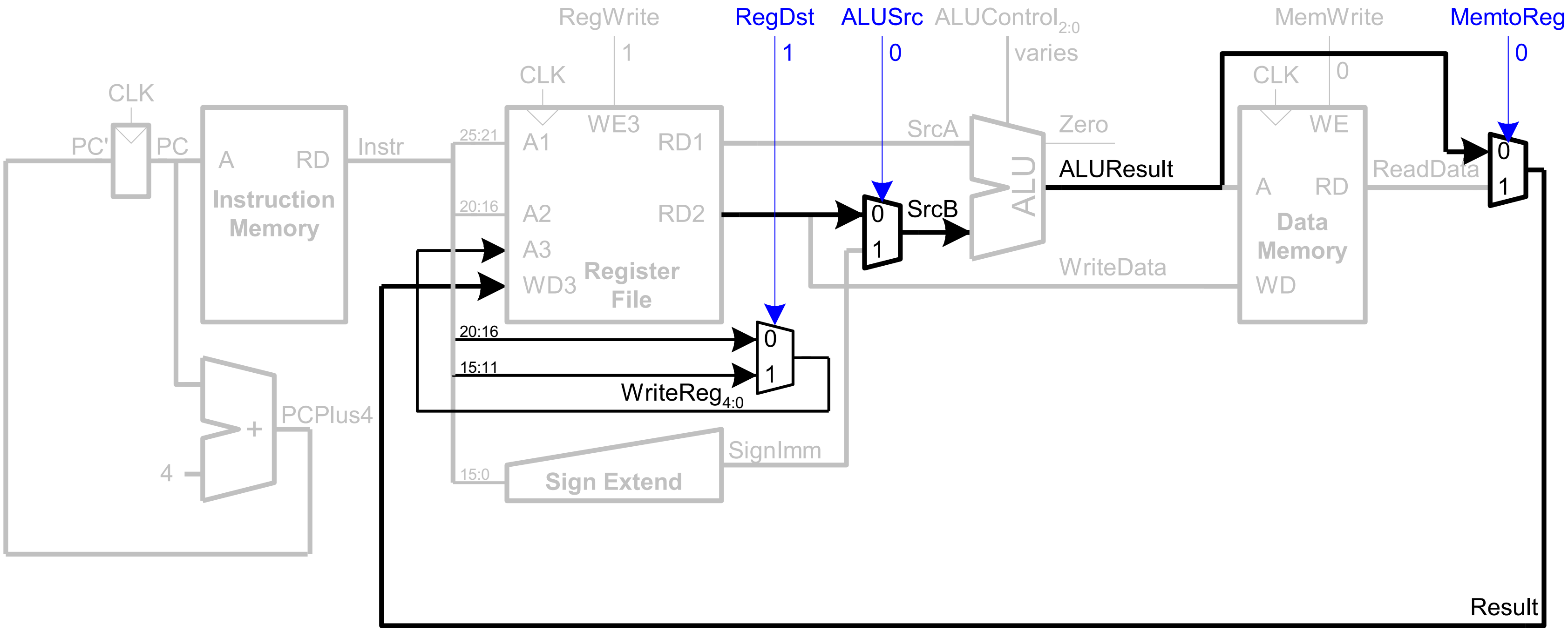
```
sw $t7, 44($0)  # write t7 into memory address 44
```

I-Type

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

Single-Cycle Datapath: R-type Instructions

- **Read from rs and rt, write ALUResult to register file**

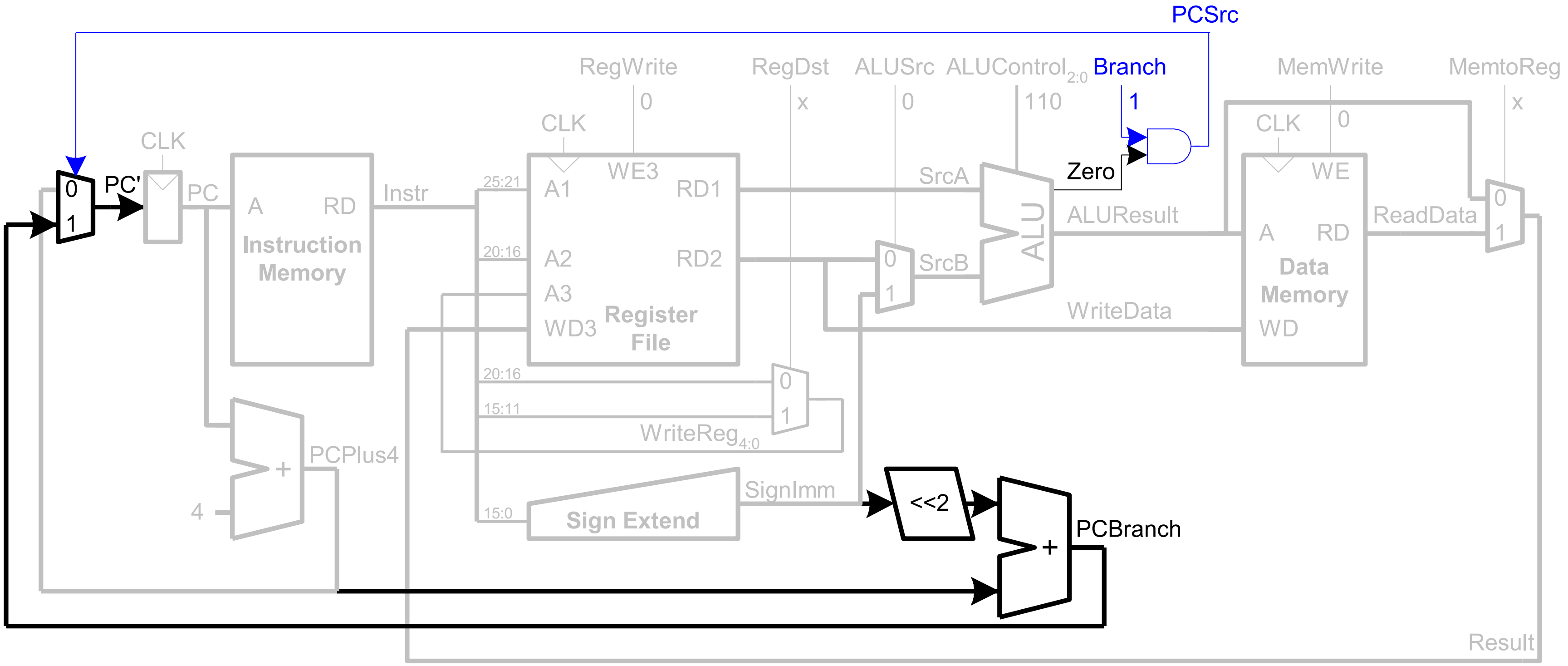


```
add t, b, c  # t = b + c
```

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

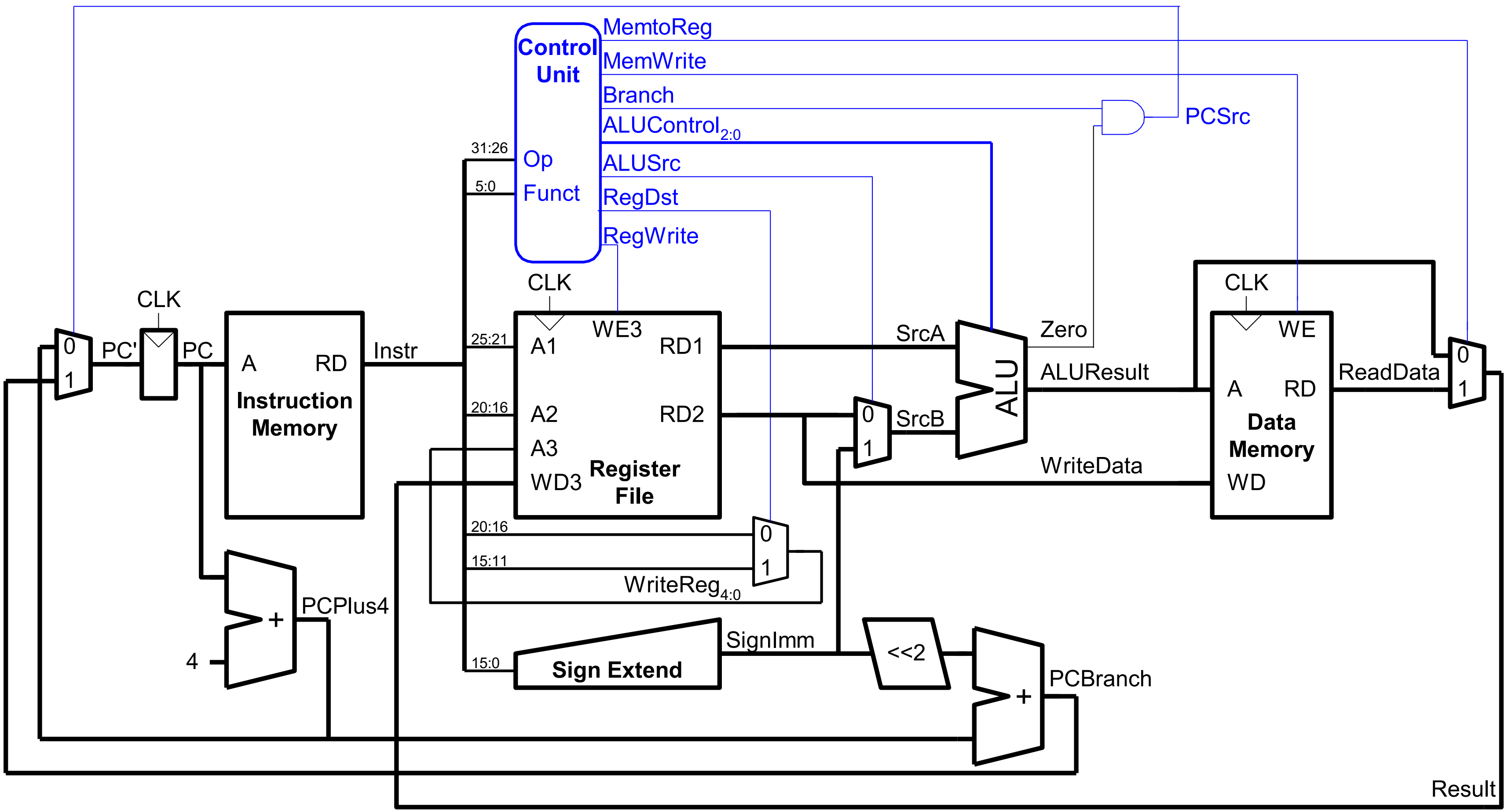
Single-Cycle Datapath: beq



```
beq  $s0, $s1, target  # branch is taken
```

- Determine whether values in rs and rt are equal
  Calculate BTA = (sign-extended immediate << 2) + (PC+4)

Complete Single-Cycle Processor

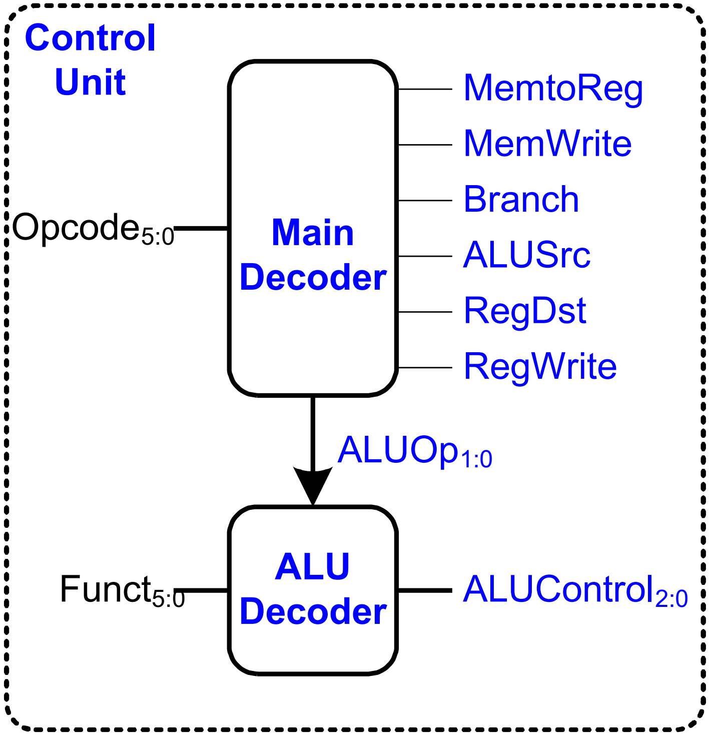


Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate ***(MUX)***
- **Write Address of Register File**
  - Either RD or RT ***(MUX)***
- **Write Data In of Register File**
  - Either ALU out or Data Memory Out ***(MUX)***
- **Write enable of Register File**
  - Not always a register write ***(MUX)***
- **Write enable of Memory**
  - Only when writing to memory (sw) ***(MUX)***

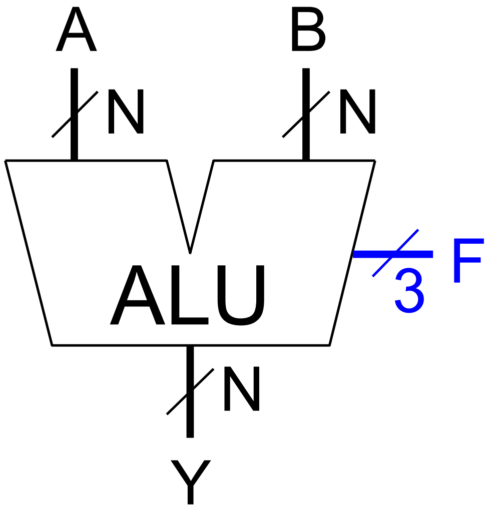
All these options are our control signals

Control Unit



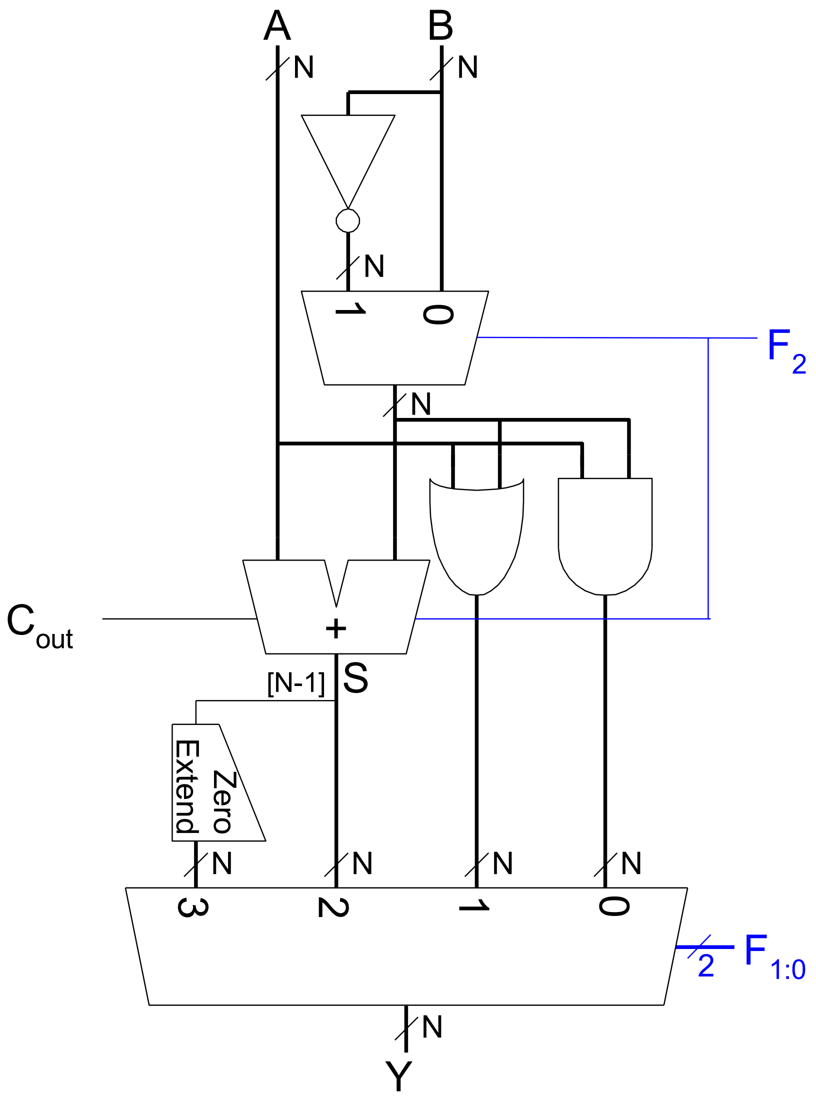
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

ALU Does the Real Work in a Processor



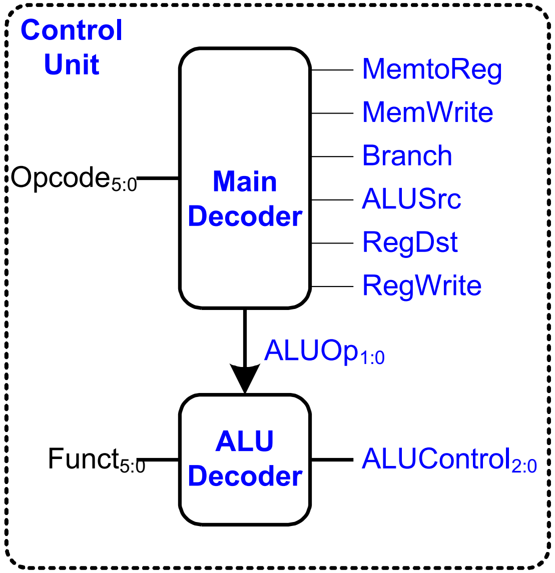
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Internals



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Control Unit: ALU Decoder



ALUOp$_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

ALUOp$_{1:0}$	Funct	ALUControl$_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$\text{Op}_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
`lw`	100011	1	0	1	0	1	add

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

Let us Develop our Control Table

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
`lw`	100011	1	0	1	0	1	add
`sw`	101011	0	X	1	1	X	add

- ***RegWrite:*** Write enable for the register file
- ***RegDst:*** Write to register RD or RT
- ***AluSrc:*** ALU input RT or immediate
- ***MemWrite:*** Write Enable
- ***MemtoReg:*** Register data in from Memory or ALU
- ***ALUOp:*** What operation does ALU do

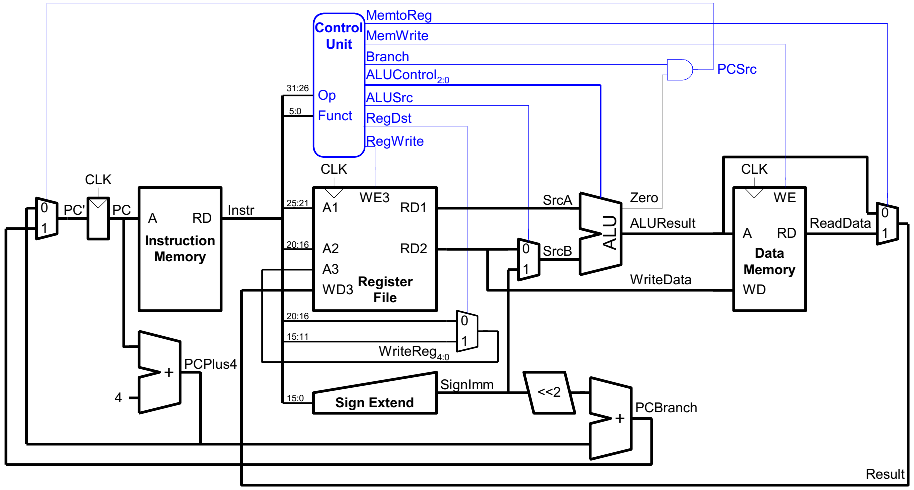
More Control Signals

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	funct
lw	100011	1	0	1	0	0	1	add
sw	101011	0	X	1	0	1	X	add
beq	000100	0	X	0	1	0	X	sub

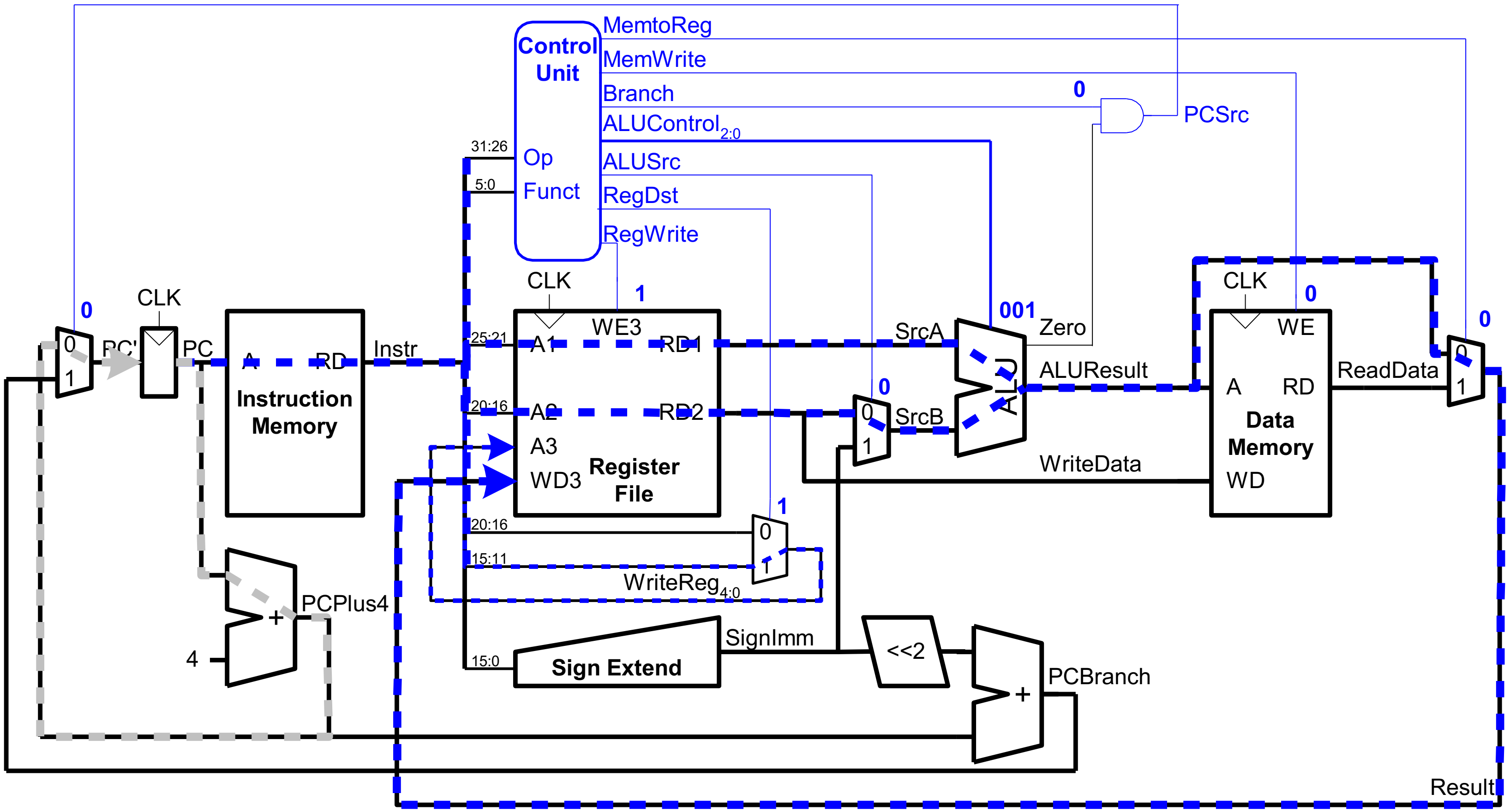
- **New Control Signal**
  - ***Branch*:** Are we jumping or not ?

Control Unit: Main Decoder

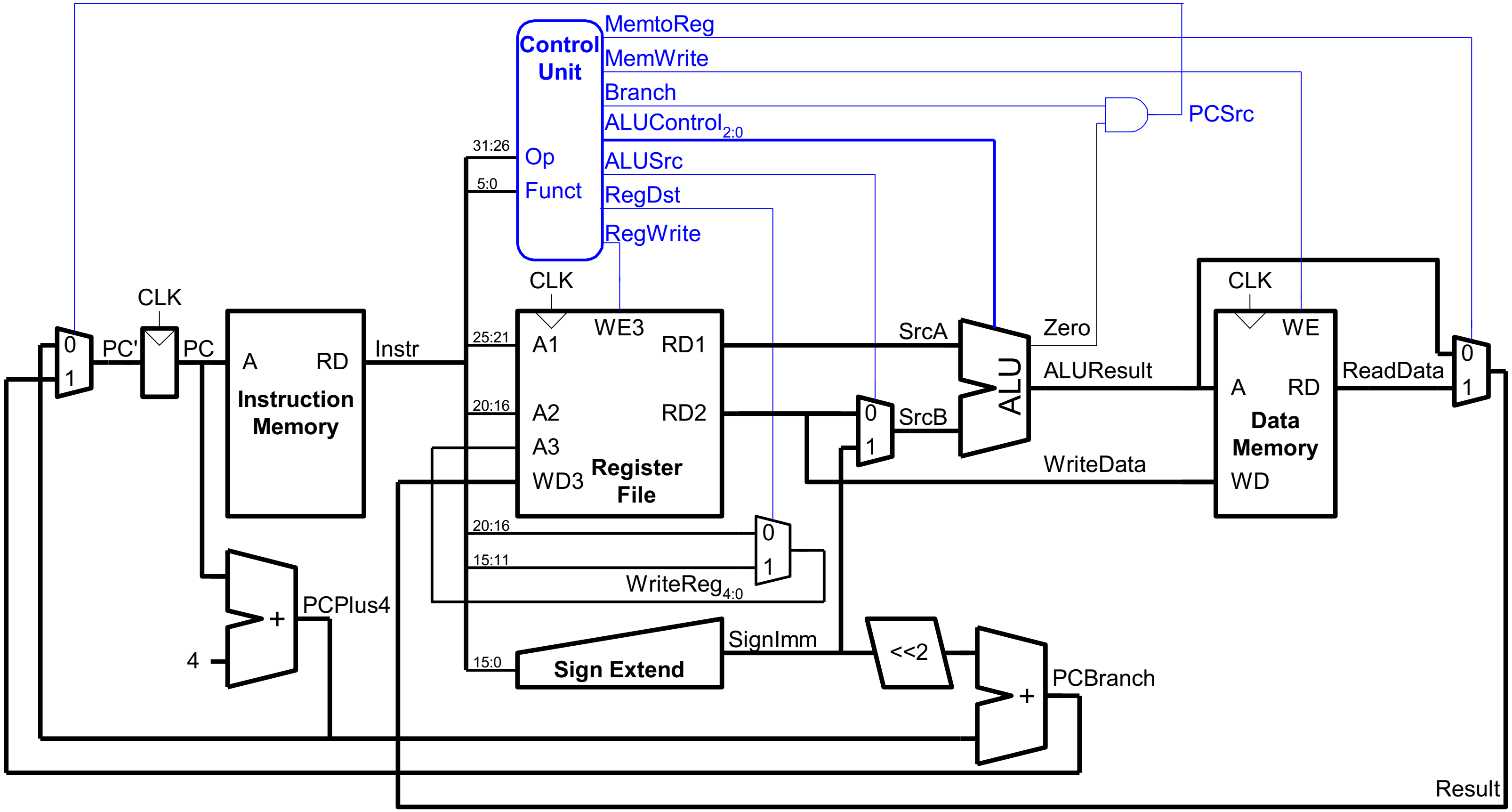
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

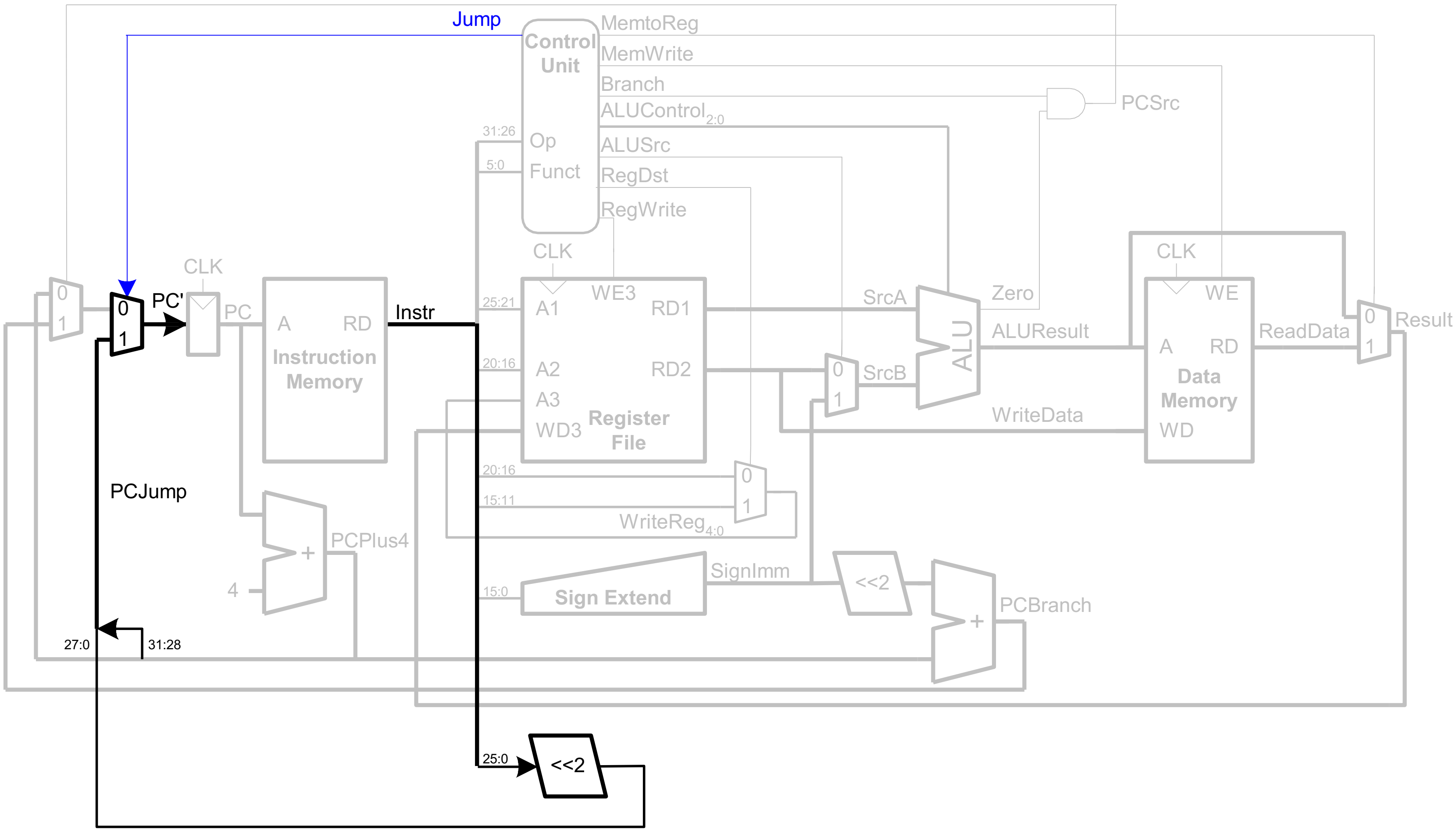


- **No change to datapath**

Control Unit: addi

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

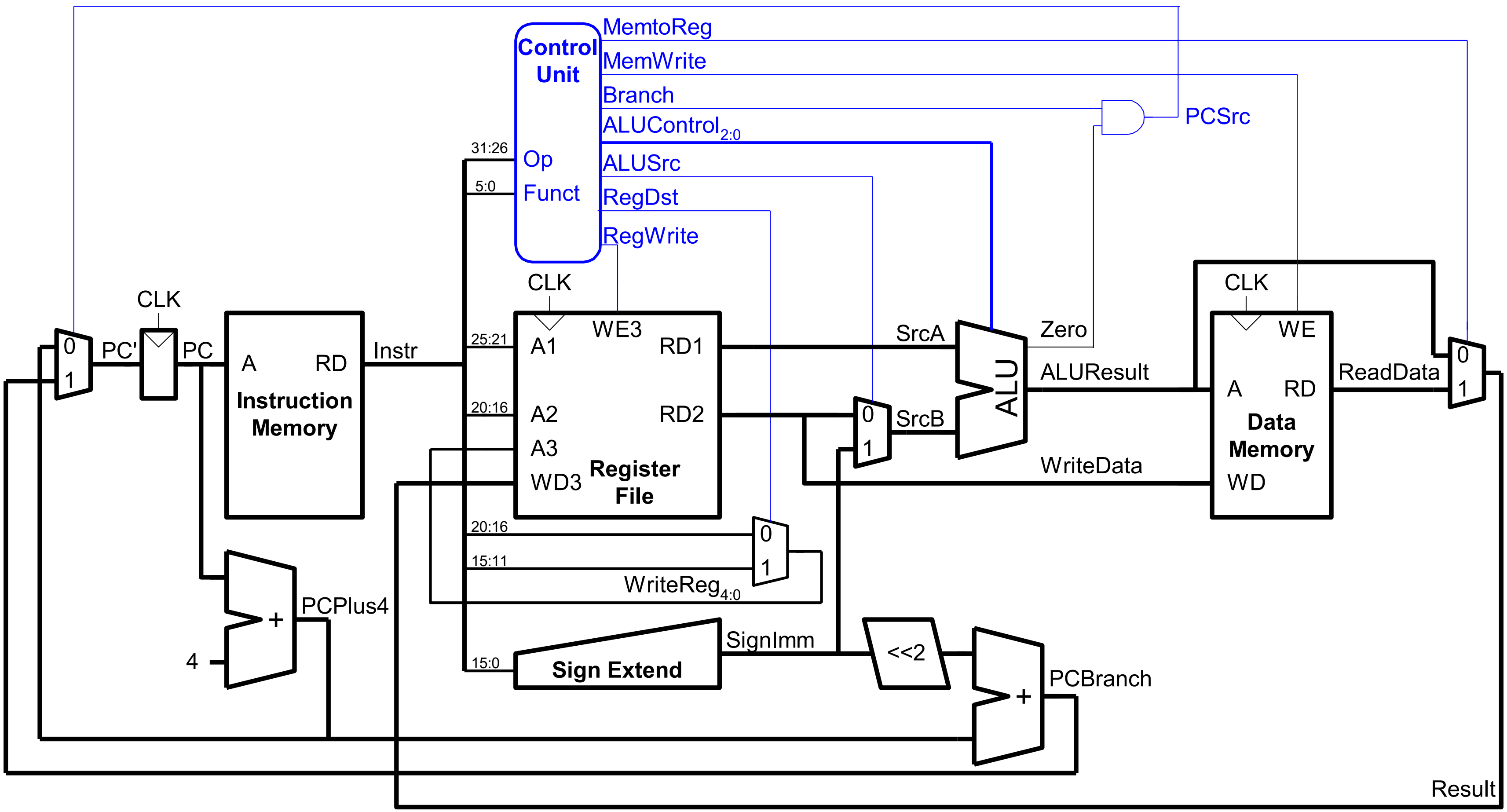
Extended Functionality: j



Control Unit: Main Decoder

Instruction	Op$_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp$_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000100	0	X	X	X	0	X	XX	1

Review: Complete Single-Cycle Processor (H&H)



A Bit More on Performance Analysis

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

Processor Performance

- **How fast is my program?**
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- **So how fast are my instructions ?**
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - ***Cycles per Instruction = CPI***

- **How much time is one clock cycle?**
  - The critical path determines how much time one cycle requires = ***clock period***.
  - 1/clock period = ***clock frequency*** = how many cycles can be done each second.

Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}

- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - **{# of instructions} x {Average CPI} x {clock cycle time}**

Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**,
    and the clock period is therefore **T**=1/f

Processor Performance

- **Now as a general formula**
  - Our program consists of executing **N** instructions.
  - Our processor needs **CPI** cycles for each instruction.
  - The maximum clock speed of the processor is **f**,
    and the clock period is therefore **T**=1/f

- **Our program will execute in**

$$N \times CPI \times (1/f) = N \times CPI \times T \text{ seconds}$$

How can I Make the Program Run Faster?

$$N \times CPI \times (1/f)$$

How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel

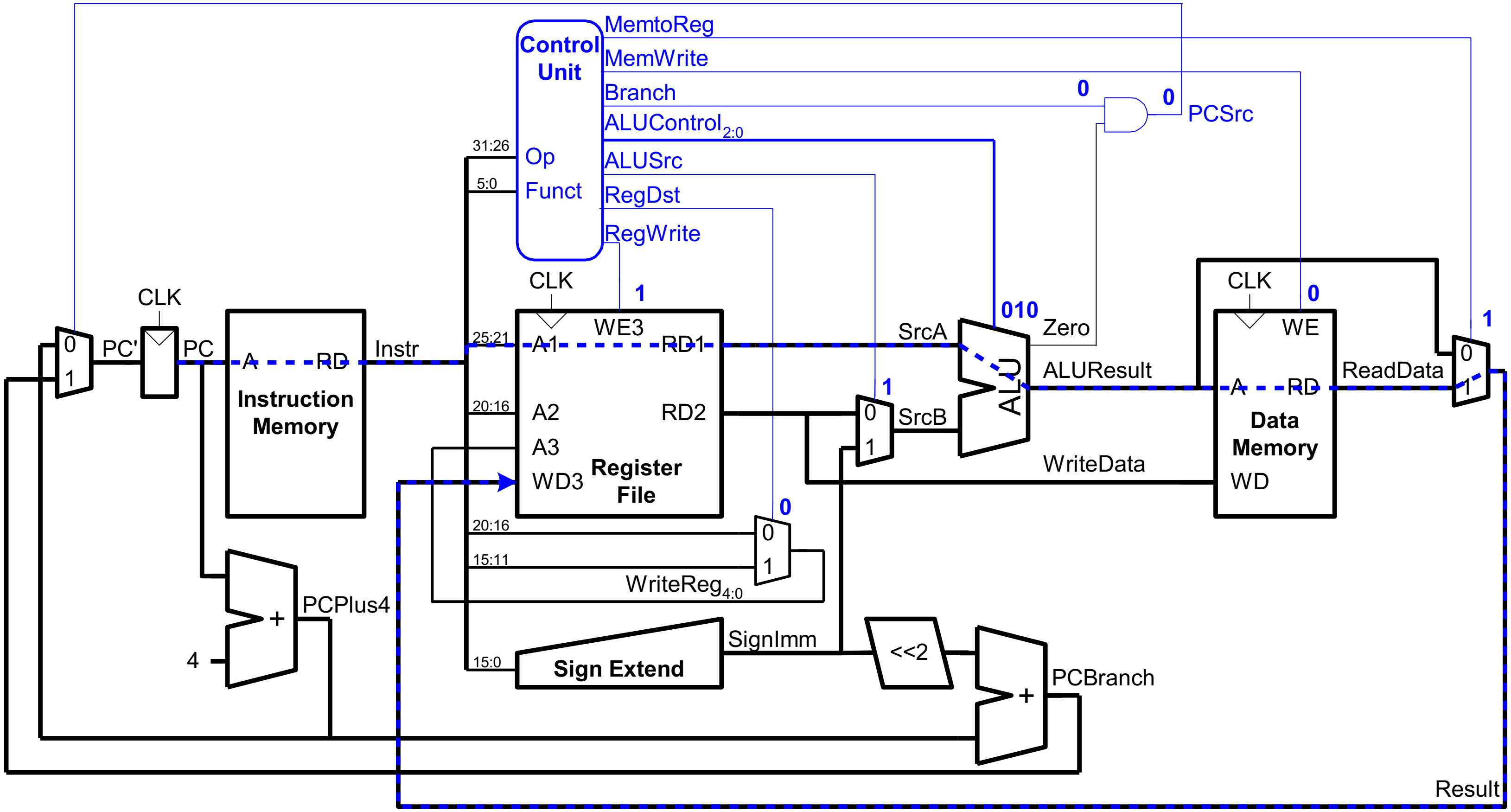
How can I Make the Program Run Faster?

N x CPI x (1/f)

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining

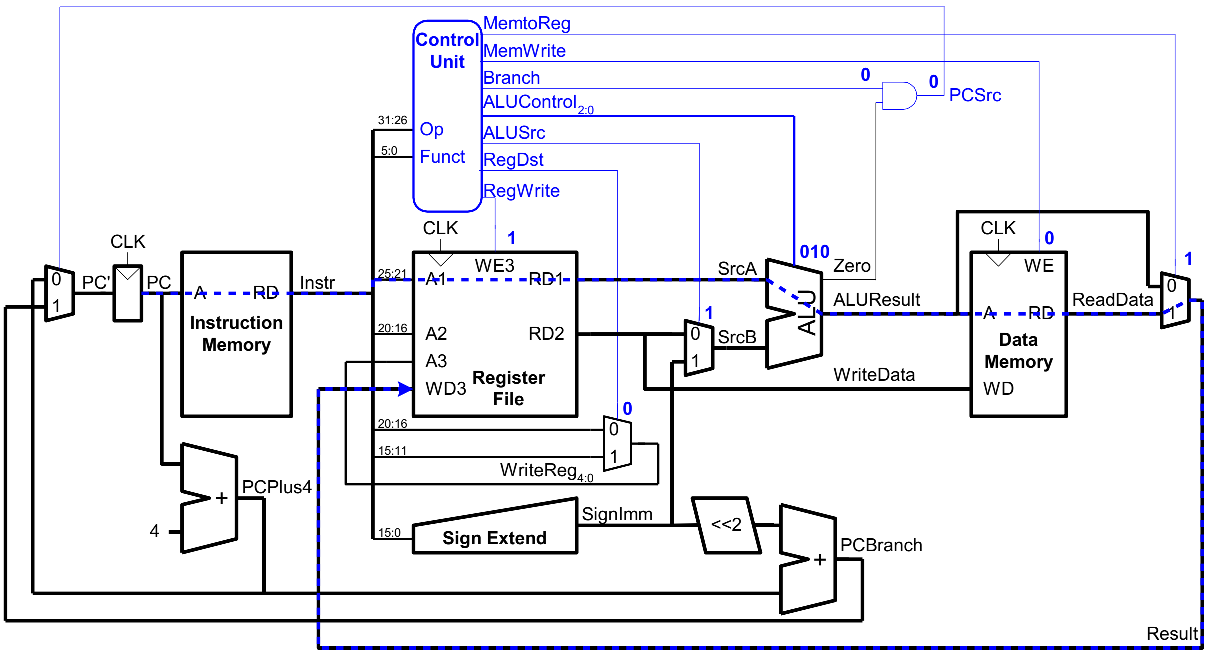
Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



Single-Cycle Performance

- **Single-cycle critical path:**
  - **$T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$**
- **In most implementations, limiting paths are:**
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$



Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$T_c =$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$\begin{aligned} T_c &= t_{pcq\_PC} + 2t_{\mathrm{mem}} + t_{RF\mathrm{read}} + t_{\mathrm{mux}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}} \\ &= [30 + 2(250) + 150 + 25 + 200 + 20]\ \mathrm{ps} \\ &= 925\ \mathrm{ps} \end{aligned}$$

Single-Cycle Performance Example

- **Example:**

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Single-Cycle Performance Example

- **Example:**

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Execution Time = # instructions x CPI x TC

$= (100 \times 10^{9})(1)(925 \times 10^{-12}\ \mathrm{s})$

= 92.5 seconds