Fundamentals of Computer Arch.

Lecture 1: Modern Microprocessor Design

Prof. Onur Mutlu
ETH Zürich

Spring 2025

19 February 2025

Brief Self Introduction

- Onur Mutlu
  - Full Professor @ ETH Zurich ITET (INFK), since Sept 2015
  - Strecker Professor @ Carnegie Mellon University ECE (CS), 2009-2016, 2016-…
  - Started the Comp Arch Research Group @ Microsoft Research, 2006-2009
  - Worked @ Google, VMware, Microsoft Research, Intel, AMD, Stanford
  - PhD in Computer Engineering from University of Texas at Austin in 2006
  - BS in Computer Engineering & Psychology from University of Michigan in 2000
  - https://people.inf.ethz.ch/omutlu/ omutlu@gmail.com

- Research and Teaching in:
  - **Computer architecture, systems, hardware security, bioinformatics**
  - Memory and storage systems
  - Robust & dependable hardware systems: security, safety, predictability, reliability
  - Hardware/software cooperation
  - New computing paradigms; architectures with emerging technologies/devices
  - Architectures for bioinformatics, genomics, health, medicine, AI/ML
  - …

My Co-Instructor

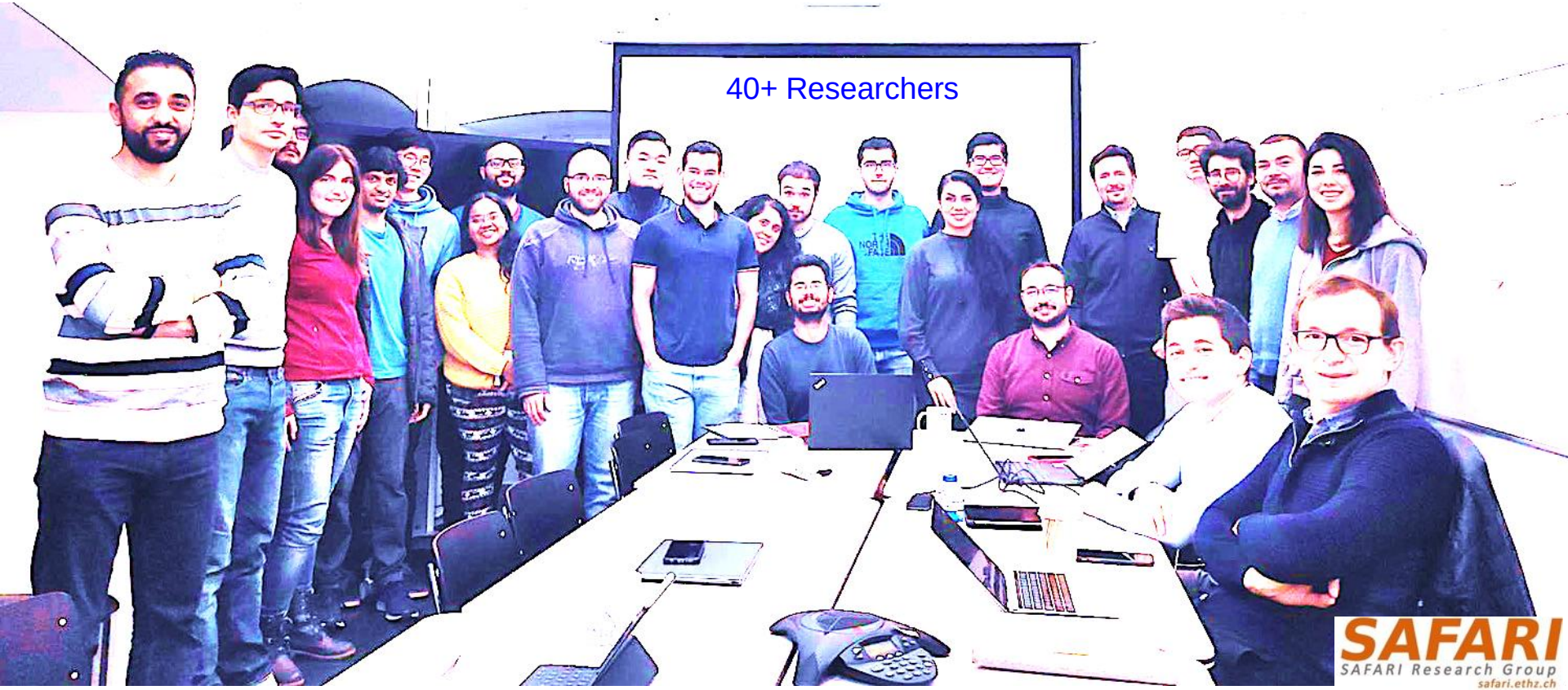
- Mohammad Sadrosadati
  - Senior Researcher and Lecturer @ SAFARI Research Group, ETHZ
  - PhD in Computer Engineering from Sharif University of Technology in 2019
  - MS in Computer Engineering from Sharif University of Technology in 2014
  - BS in Computer Engineering from Sharif University of Technology in 2012
  - mohammad.sadrosadati@safari.ethz.ch

- Research & Teaching Areas
  - Computer Architecture
  - Memory/Storage Systems
  - Near-Data Processing
  - Heterogeneous System Architecture
  - Bioinformatics
  - Interconnection Network

SAFARI Research Group

Computer architecture, HW/SW, systems, bioinformatics, security, memory

https://safari.ethz.ch/safari-newsletter-april-2020/



Think BIG, Aim HIGH!

SAFARI

https://safari.ethz.ch

SAFARI Newsletter January 2021 Edition

- https://safari.ethz.ch/safari-newsletter-january-2021/

SAFARI
SAFARI Research Group

Newsletter
January 2021

Think Big, Aim High, and
Have a Wonderful 2021!

Dear SAFARI friends,

Happy New Year! We are excited to share our group highlights with you in this second edition of the SAFARI newsletter (You can find the first edition from April 2020 here). 2020 has

SAFARI Newsletter July 2024 Edition

- **https://safari.ethz.ch/safari-newsletter-july-2024/**

SAFARI Introduction & Research

Computer architecture, HW/SW, systems, bioinformatics, security, memory

SAFARI
SAFARI Research Group
safari.ethz.ch

Think BIG, Aim HIGH!

SAFARI

https://www.youtube.com/watch?v=mV2OuB2djEs

SAFARI PhD and Post-Doc Alumni

- https://safari.ethz.ch/safari-alumni/

- C. Firtina (ETH Zurich)
- L. Breitwieser (CERN), Best artifact award at PPoPP 2023
- A. Giray Yaglikci (ETH Zurich), PACT 2023 SRC Winner, Intel Hardware Security Academic Award Finalist 2021, HOST PhD Competition Finalist 2024
- Hasan Hassan (Rivos), EDAA Outstanding Dissertation Award 2023; S&P 2020 Best Paper Award, 2020 Pwnie Award, IEEE Micro TP HM 2020
- Christina Giannoula (Univ. of Toronto), NTUA Best Dissertation Award 2023
- Minesh Patel (Rutgers, Asst. Prof.), DSN Carter Award Best Thesis 2022; ETH Medal 2023; MICRO'20 & DSN'20 Best Paper Awards; ISCA HoF 2021
- Damla Senol Cali (Bionano Genomics), SRC TECHCON 2019 Best Student Presentation Award; RECOMB-Seq 2018 Best Poster Award
- Nastaran Hajinazar (Intel)
- Gagandeep Singh (AMD/Xilinx), FPL 2020 Best Paper Award Finalist
- Amirali Boroumand (Stanford Univ → Google), SRC TECHCON 2018 Best Presentation Award
- Jeremie Kim (Apple), EDAA Outstanding Dissertation Award 2020; IEEE Micro Top Picks 2019; ISCA/MICRO HoF 2021
- Nandita Vijaykumar (Univ. of Toronto, Assistant Professor), ISCA Hall of Fame 2021
- Kevin Hsieh (Microsoft Research, Senior Researcher)
- Justin Meza (Facebook), HiPEAC 2015 Best Student Presentation Award; ICCD 2012 Best Paper Award
- Mohammed Alser (ETH Zurich), IEEE Turkey Best PhD Thesis Award 2018
- Yixin Luo (Google), HPCA 2015 Best Paper Session
- Kevin Chang (Facebook), SRC TECHCON 2016 Best Student Presentation Award
- Rachata Ausavarungnirun (KMUNTB, Assistant Professor), NOCS 2015 and NOCS 2012 Best Paper Award Finalist
- Gennady Pekhimenko (Univ. of Toronto, Assistant Professor), ISCA Hall of Fame 2021; ASPLOS 2015 SRC Winner
- Vivek Seshadri (Microsoft Research, Principal Researcher)
- Donghyuk Lee (NVIDIA Research, Senior Researcher), HPCA Hall of Fame 2018
- Yoongu Kim (Software Robotics → Google), IFIP JCL Award'24, TCAD'19 Top Pick Award; IEEE Micro Top Picks'10; HPCA'10 Best Paper Session
- Lavanya Subramanian (Intel Labs → Facebook)

- Samira Khan (Univ. of Virginia, Assistant Professor), HPCA 2014 Best Paper Session
- Saugata Ghose (Univ. of Illinois, Assistant Professor), DFRWS-EU 2017 Best Paper Award
- Jawad Haj-Yahya (Huawei Research Zurich, Principal Researcher)
- Lois Orosa (Galicia Supercomputing Center, Director)
- Jisung Park (POSTECH, Assistant Professor)
- Gagandeep Singh (AMD/Xilinx, Researcher)
- Juan Gomez-Luna (NVIDIA, Researcher), ISPASS 2023 Best Paper Session
- Mohammed Alser (Georgia State Univ. Assistant Professor), IEEE Turkey Best PhD Thesis Award 2018

An Interview on Computing Futures

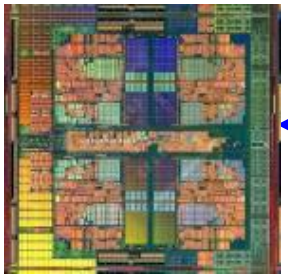
https://www.youtube.com/watch?v=8ffSEKZhmvo

Suggestions on Research, Education, PhD

https://www.youtube.com/watch?v=83tlorht7Mc&list=PL5Q2soXY2Zi8D_5MGV6EnXEJHnV2YFBJl&index=54

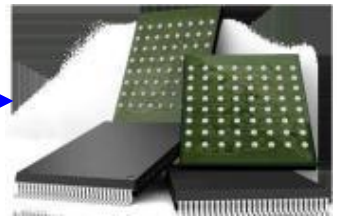
SAFARI Research Group: Current Mission

Computer architecture, HW/SW, systems, bioinformatics, security

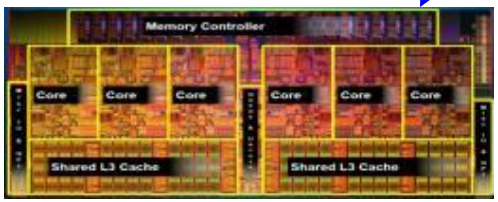


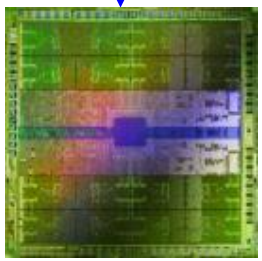
Heterogeneous Processors and Accelerators

Processing in Memories



Storage, Sensors, Memory, Interconnect with Processing Capabilities + New Paradigms for Computation





ML/AI, Vision, Genomics, Biological Signal Processing

https://safari.ethz.ch

SAFARI
SAFARI Research Group
safari.ethz.ch

Enable fundamentally better computers

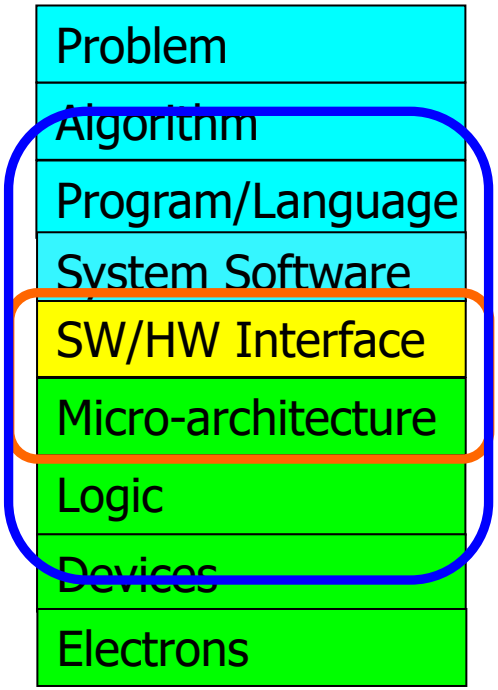
https://safari.ethz.ch/safari-newsletter-july-2024/

Major Current Research Topics

- Fundamentally Robust (Secure/Reliable/Safe) Architectures
- Fundamentally Energy-Efficient Architectures
  - Memory-centric (Data-centric) Architectures
- Fundamentally Low-Latency and Predictable Architectures
- Fundamentally Intelligent and Evolving Architectures
  - ML/AI-Assisted (Data-driven) and Data-aware Architectures
- Architectures for ML/AI, Genomics, Medicine, Health, …

The Transformation Hierarchy

Computer Architecture (expanded view)



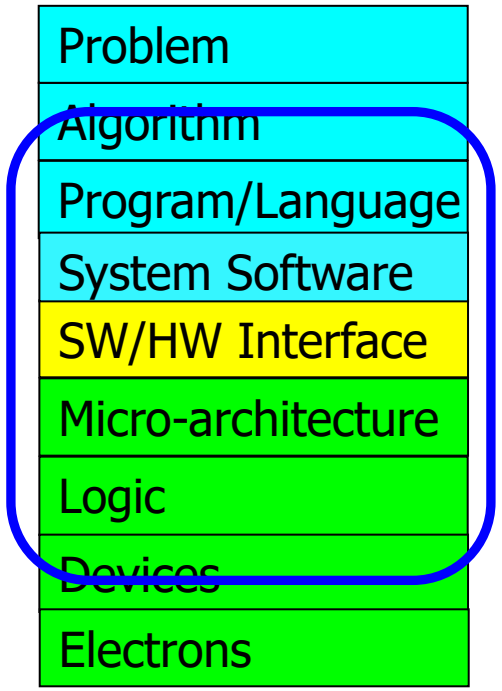
Computer Architecture (narrow view)

Approach: Cross Layer Design

To achieve the highest efficiency, performance, robustness:

we must take the expanded view

of computer architecture



Co-design across the hierarchy: Algorithms to devices

Specialize as much as possible within the design goals

Broad research
spanning applications, systems, software, logic, circuits
with architecture at the center

Principle: Teaching and Research

...

Teaching drives Research

Research drives Teaching

...

Accessing All Our Courses

SAFARI
Think Big, Aim High

Home | People | Courses | News | Research | Publications | Tools | Work with us | Contact us

researchers and practitioners, including leading companies. Many students and universities without access to state-of-the-art computer architecture classes benefit from our online classes (see our YouTube channels here).

Spring 2025:
Fundamentals of Computer Architecture
Digital Design and Computer Architecture
Seminar in Computer Architecture
SAFARI **Project & Seminars** courses

Fall 2024:
Computer Architecture
Seminar in Computer Architecture
SAFARI **Project & Seminars** courses

Spring 2024:
Digital Design and Computer Architecture
Seminar in Computer Architecture
SAFARI **Project & Seminars** courses

Fall 2023:
Computer Architecture
Seminar in Computer Architecture
SAFARI **Project & Seminars** courses

Spring 2023:
Digital Design and Computer Architecture
Seminar in Computer Architecture
SAFARI **Project & Seminars** courses

Onur Mutlu Lectures
47.1K subscribers

Subscribe to our newsletter

First name *

Last name *

Email *

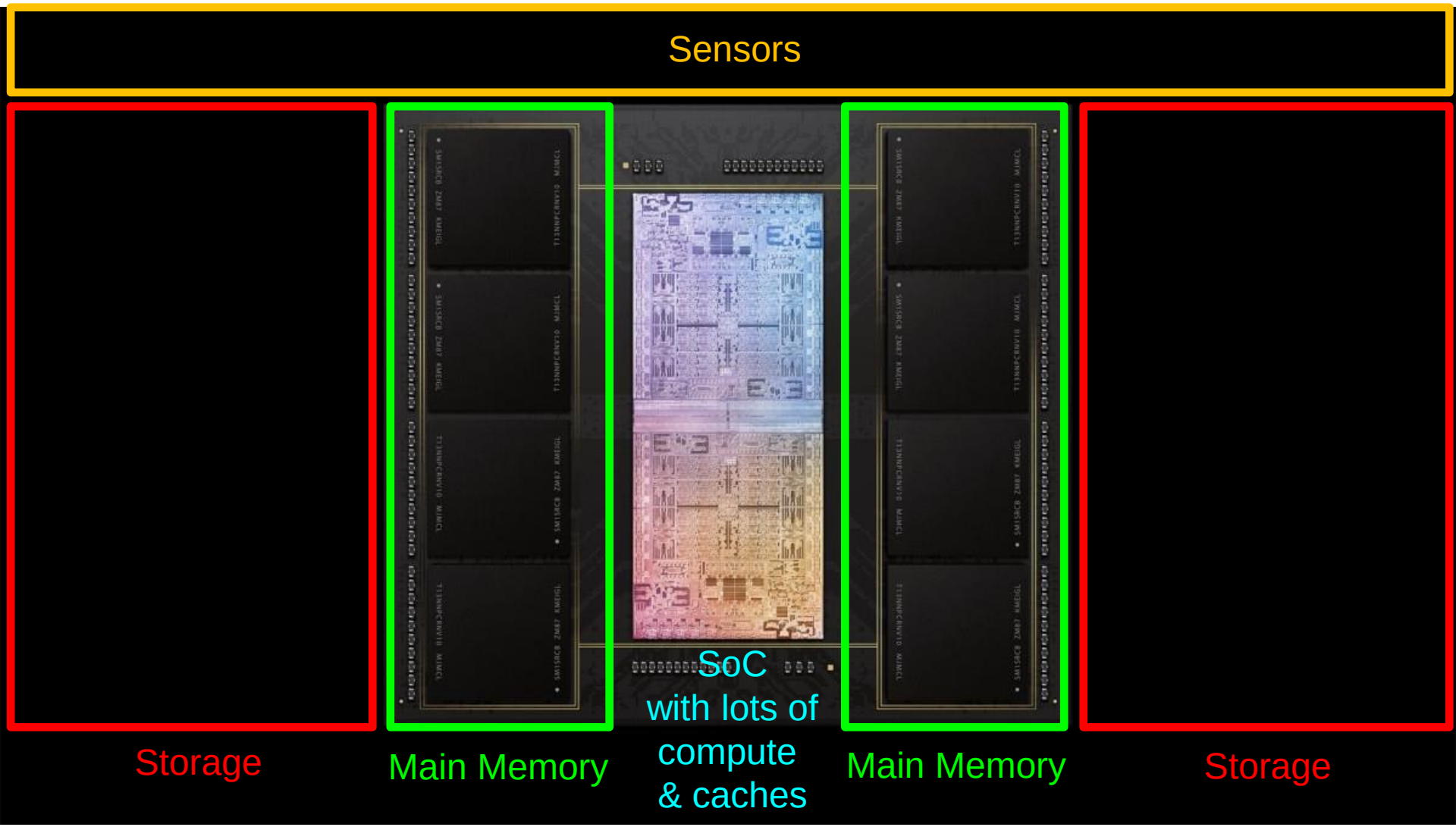
Subscribe!

https://safari.ethz.ch/courses

ETH zürich

Basic Goals & Structure of FOCA

We Will Study How Something Like This Works



Apple M1 Ultra System (2022)

https://www.gsmarena.com/apple_announces_m1_ultra_with_20core_cpu_and_64core_gpu-news-53481.php

Related Courses (I)

- DDCA (Digital Design & Computer Architecture; every Spring)
  - Bachelor's first year; mostly Computer Science students
  - Broader than FOCA, but some overlap in topics
  - Hands-on FPGA labs

- Computer Architecture (every Fall)
  - Master's level; mix of CS & ECE
  - More advanced topics than FOCA
  - Good to take after either DDCA or FOCA

- Seminar in Computer Architecture (every Fall and Spring)
  - Mostly Bachelor's students; mix of CS and ECE
  - Good to take after DDCA or FOCA or Computer Engineering

Related Courses (II)

- Computer Engineering (every Spring)
  - Bachelor's second year; ECE students
  - Prerequisite to FOCA
  - Covers OS and some Basic Comp Arch concepts

- FOCA
  - Bachelor's third year; ECE students
  - Covers Comp Arch concepts
    - Subset of DDCA + some advanced topics
    - Builds on Computer Engineering
  - Hands-on labs
  - Best to take if interested in hardware and computer design

Major High-Level Goals of This Course

- In Fundamentals of Computer Architecture

- Understand the basics
- Understand the principles (of design)
- Understand the precedents

- Based on such understanding:
  - learn a modern computing system works underneath
  - learn HW/SW interfaces of modern computing platforms
  - evaluate tradeoffs of different designs and ideas
  - implement and simulate various components
  - learn to systematically debug increasingly complex systems
  - Hopefully enable you to develop novel, out-of-the-box designs

FOCA Course Components

- **Lectures** (understanding concepts)
- **Readings** (reinforcing & going deeper)
- **Optional Homeworks** (problem solving preparation)
- **Labs** (hands-on experience in some concepts)
- **Exam** (test of understanding)
- **Extra Credit Assignments** (fundamental and simple)

- My advice: Focus on learning & scholarship & understanding

https://safari.ethz.ch/foca/spring2025/

Learning & Exam

- We will enable you to learn + prepare you for the exam

- My suggestions:
  - focus on understanding, learning, mastering the material
    - lectures, readings, labs, HWs all enable this and prepare you
  - reinforce problem solving skills with homeworks
  - do **not** worry about the exam while listening to lectures

- We will release a lot of material to help you with the exam
  - Problem solving sessions
  - Exam guidance

Summary

- Learning is for life (never ends)
- Exam study is until you pass (ends, hopefully August 2025)

Focus on

learning and scholarship

How to Approach This Course

Learning experience

Long-term tradeoff analysis

Critical thinking & decision making

How to Approach This Course

Find and choose
the learning style
that works best for you

Why Study Computer Architecture?

What is Computer Architecture?

- 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.

Why Study Computer Architecture?

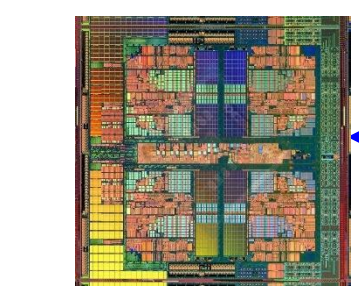
- Enable better systems: make computers faster, cheaper, smaller, more reliable, …
  - By exploiting advances and changes in underlying technology/circuits
- Enable new applications
  - Life-like 3D visualization 20 years ago? Virtual reality?
  - Self-driving cars?
  - Personalized genomics? Personalized medicine?
- Enable better solutions to problems
  - Software innovation is built on trends and changes in computer architecture
    - > 50% performance improvement per year has enabled this innovation
- Understand why computers work the way they do

Computer Architecture Today (I)

- Today is a very exciting time to study computer architecture
- Industry is in a large paradigm shift (to novel architectures) – many different potential system designs possible
- Many difficult problems *motivating* and *caused by* the shift
  - Huge hunger for data and new data-intensive applications
  - Power/energy/thermal constraints
  - Complexity of design
  - Difficulties in technology scaling
  - Memory bottleneck
  - Reliability problems
  - Programmability problems
  - Security and privacy issues
- No clear, definitive answers to these problems

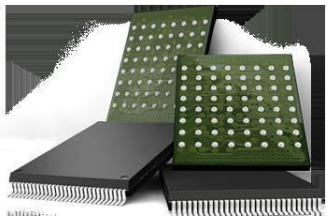
Computer Architecture Today (II)

- Computing landscape is very different from 10-20 years ago
- Applications and technology both demand novel architectures

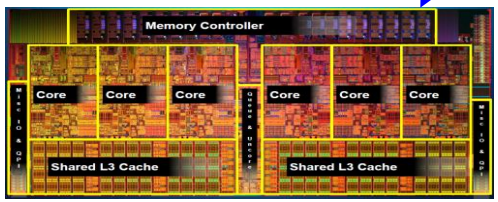


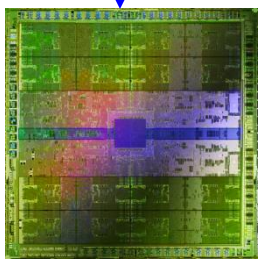
Heterogeneous Processors and Accelerators

Hybrid Main Memory



Persistent Memory/Storage

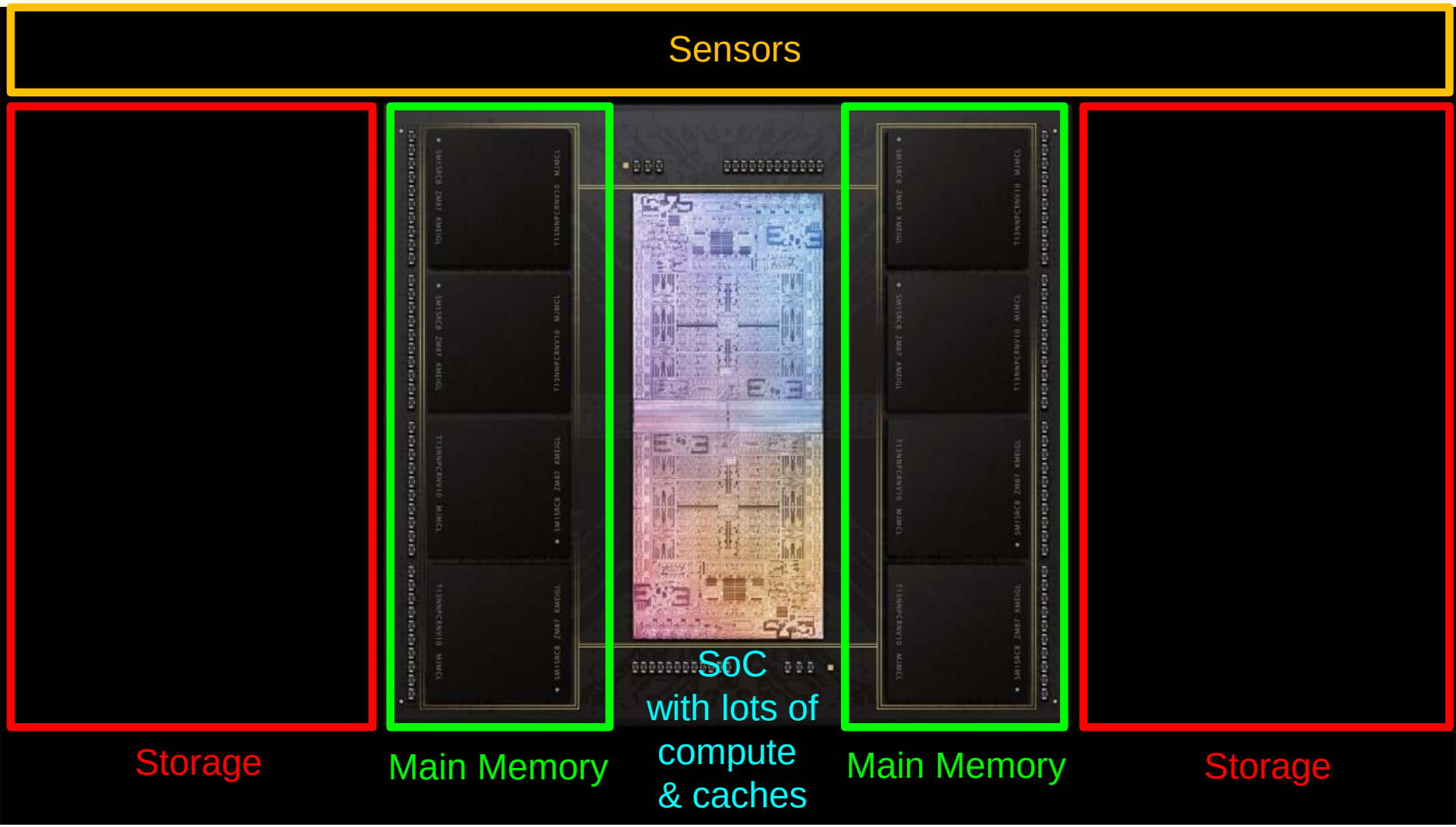




General Purpose GPUs

Every component and its interfaces, as well as entire system designs are being re-examined

An Example System in Your Pocket



Apple M1 Ultra System (2022)

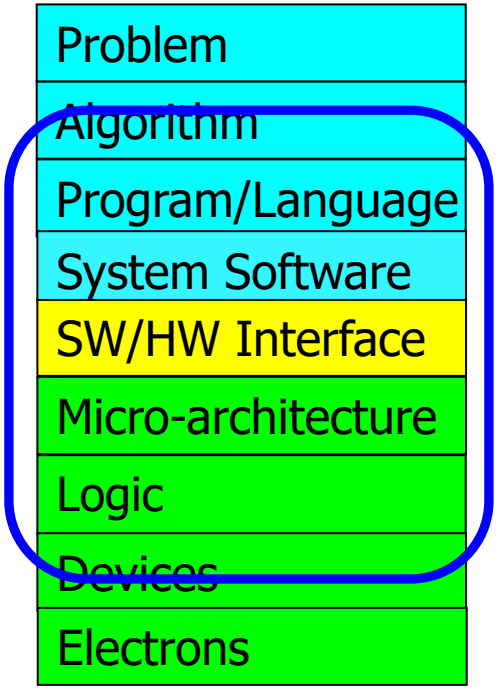
https://www.gsmarena.com/apple_announces_m1_ultra_with_20core_cpu_and_64core_gpu-news-53481.php

Axiom

To achieve the highest energy efficiency and performance:

we must take the expanded view

of computer architecture



Co-design across the hierarchy: Algorithms to devices

Specialize as much as possible within the design goals

Historical: Opportunities at the Bottom

There's Plenty of Room at the Bottom

From Wikipedia, the free encyclopedia

"**There's Plenty of Room at the Bottom: An Invitation to Enter a New Field of Physics**" was a lecture given by physicist Richard Feynman at the annual American Physical Society meeting at Caltech on December 29, 1959.[1] Feynman considered the possibility of direct manipulation of individual atoms as a more powerful form of synthetic chemistry than those used at the time. Although versions of the talk were reprinted in a few popular magazines, it went largely unnoticed and did not inspire the conceptual beginnings of the field. Beginning in the 1980s, nanotechnology advocates cited it to establish the scientific credibility of their work.

https://en.wikipedia.org/wiki/There%27s_Plenty_of_Room_at_the_Bottom

Historical: Opportunities at the Bottom (II)

There's Plenty of Room at the Bottom

From Wikipedia, the free encyclopedia

Feynman considered some ramifications of a general ability to manipulate matter on an atomic scale. He was particularly interested in the possibilities of denser computer circuitry, and microscopes that could see things much smaller than is possible with scanning electron microscopes. These ideas were later realized by the use of the scanning tunneling microscope, the atomic force microscope and other examples of scanning probe microscopy and storage systems such as Millipede, created by researchers at IBM.

Feynman also suggested that it should be possible, in principle, to make nanoscale machines that "arrange the atoms the way we want", and do chemical synthesis by mechanical manipulation.

He also presented the possibility of "swallowing the doctor", an idea that he credited in the essay to his friend and graduate student Albert Hibbs. This concept involved building a tiny, swallowable surgical robot.

https://en.wikipedia.org/wiki/There%27s_Plenty_of_Room_at_the_Bottom

Historical: Opportunities at the Top

REVIEW

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson[1], Neil C. Thompson[1,2,*], Joel S. Emer[1,3], Bradley C. Kuszmaul[1,†], Butler W. Lampson[1,4], ...
+ See all authors and affiliations

Science 05 Jun 2020:
Vol. 368, Issue 6495, eaam9744
DOI: 10.1126/science.aam9744

Much of the improvement in computer performance comes from decades of miniaturization of computer components, a trend that was foreseen by the Nobel Prize–winning physicist Richard Feynman in his 1959 address, "There's Plenty of Room at the Bottom," to the American Physical Society. In 1975, Intel founder Gordon Moore predicted the regularity of this miniaturization trend, now called Moore's law, which, until recently, doubled the number of transistors on computer chips every 2 years.

Unfortunately, semiconductor miniaturization is running out of steam as a viable way to grow computer performance—there isn't much more room at the "Bottom." If growth in computing power stalls, practically all industries will face challenges to their productivity. Nevertheless, opportunities for growth in computing performance will still be available, especially at the "Top" of the computing-technology stack: software, algorithms, and hardware architecture.

https://www.science.org/doi/10.1126/science.aam9744

Axiom, Revisited

There is plenty of room both at the top and at the bottom

but **much more so**

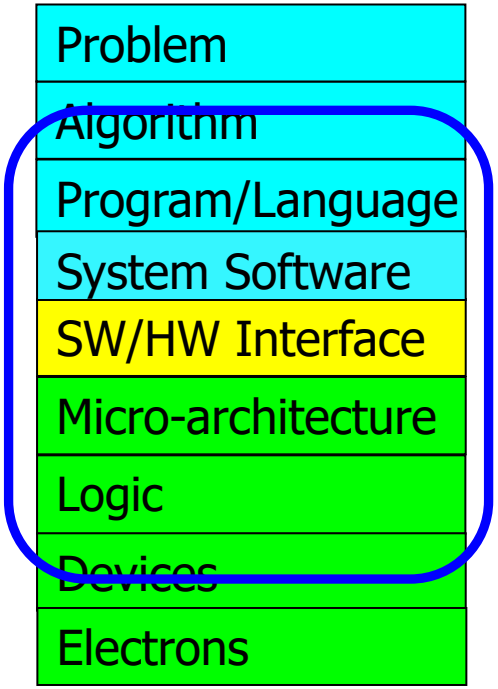
when you

communicate well between and optimize across

the top and the bottom

Hence the Expanded View

Computer Architecture (expanded view)



Course Info and Logistics

Brief Self Introduction

- Onur Mutlu
  - Full Professor @ ETH Zurich ITET (INFK), since Sept 2015
  - Strecker Professor @ Carnegie Mellon University ECE (CS), 2009-2016, 2016-…
  - Started the Comp Arch Research Group @ Microsoft Research, 2006-2009
  - Worked @ Google, VMware, Microsoft Research, Intel, AMD
  - PhD in Computer Engineering from University of Texas at Austin in 2006
  - BS in Computer Engineering & Psychology from University of Michigan in 2000
  - https://people.inf.ethz.ch/omutlu/ omutlu@gmail.com

- Research and Teaching in:
  - **Computer architecture, systems, hardware security, bioinformatics**
  - Memory and storage systems
  - Robust & dependable hardware systems: security, safety, predictability, reliability
  - Hardware/software cooperation
  - New computing paradigms; architectures with emerging technologies/devices
  - Architectures for bioinformatics, genomics, health, medicine, AI/ML
  - …

My Co-Instructor

- Mohammad Sadrosadati
  - Senior Researcher and Lecturer @ SAFARI Research Group, ETHZ
  - PhD in Computer Engineering from Sharif University of Technology in 2019
  - MS in Computer Engineering from Sharif University of Technology in 2014
  - BS in Computer Engineering from Sharif University of Technology in 2012
  - mohammad.sadrosadati@safari.ethz.ch

- Research & Teaching Areas
  - Computer Architecture
  - Memory/Storage Systems
  - Near-Data Processing
  - Heterogeneous System Architecture
  - Bioinformatics
  - Interconnection Network

Course Info: Lecturers & PhD Assistants

- Head Assistants
  - Dr. Konstantina Koliogeorgi
- Assistants and Guest Lecturers
  - Dr. Giray Yaglikci
  - Dr. Can Firtina
  - Ataberk Olgun
  - Geraldo De Oliveira Junior
  - Rahul Bera
  - Konstantinos Kanellopoulos
  - Nika Mansouri Ghiasi
  - Rakesh Nadig
  - Nisa Bostancı
  - İsmail Emir Yüksel
  - Haocong Luo
  - Andreas Kakolyris
  - Mayank Kabra
  - Jikun Wang
  - Susana Rebolledo Ruiz
  - Harshita Gupta

If You Need Help

- Post your question on Moodle Q&A Forum
  - https://moodle-app2.let.ethz.ch/course/view.php?id=19395
  - We will create a forum on Moodle for each activity
  - **Preferred** for **technical** questions

- Write an e-mail to:
  - mohammad.sadrosadati@safari.ethz.ch
  - omutlu@ethz.ch

- Come to office hours
  - We will provide office locations & Zoom links
  - TBD

Where to Get Up-to-date Course Info?

- Website:
  - **https://safari.ethz.ch/foca/spring2025/**
  - Lecture slides and videos
  - Readings
  - Lab information
  - Course schedule, handouts, FAQs
  - Software
  - Any other useful information for the course
  - Check frequently for announcements and due dates
  - This is your single point of access to all resources
- Your ETH Email
- Lecturers and Teaching Assistants

Lecture and Lab Times and Policies

- Lectures:
  - Wednesdays, 14:00-16:00, HG D5.1
  - YouTube livestream playlist: https://www.youtube.com/playlist?list=PL5Q2soXY2Zi_ZMtqz1r-GHm-zzuE1QfIg
  - Zoom link provided via Moodle
  - Attendance is for your benefit and is therefore important
  - Some days, we may have guest lectures and exercise sessions
- Lab sessions:
  - Thursdays, 16:00-17:00, NO E39
  - You should definitely attend the lab sessions
    - Info about each lab and its evaluation
  - Labs will start on March 6th
  - Lab information and handouts are here:
    - https://safari.ethz.ch/foca/spring2025/doku.php?id=labs

Final Exam

- 180-minute written exam
  - Find examination rules in Course Catalogue
    - https://www.vvz.ethz.ch/Vorlesungsverzeichnis/lerneinheit.view?semkez=2025S&ansicht=KATALOGDATEN&lerneinheitId=191658&lang=en
  - Some exam questions may be similar to those in Optional HWs
    - Optional HWs are not graded, but highly recommended to solve

How Will You Be Evaluated?

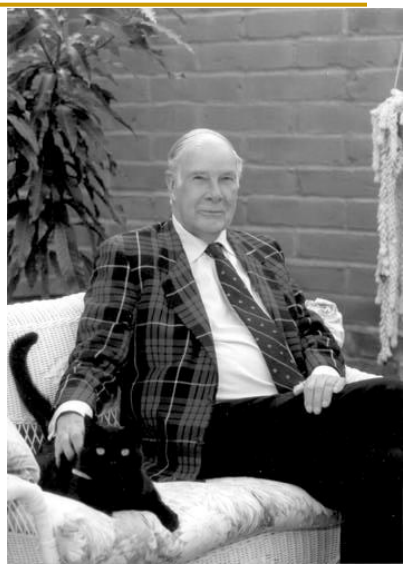
- Lab assignments: 50%
- Final exam (180 minutes): 50%

- Extra credit possibilities in HWs, Labs, Exam

Modern Microprocessor Design

Levels of Transformation

"The purpose of computing is [to gain] insight" (*Richard Hamming*)
We gain and generate insight by solving problems
How do we ensure problems are solved by electrons?

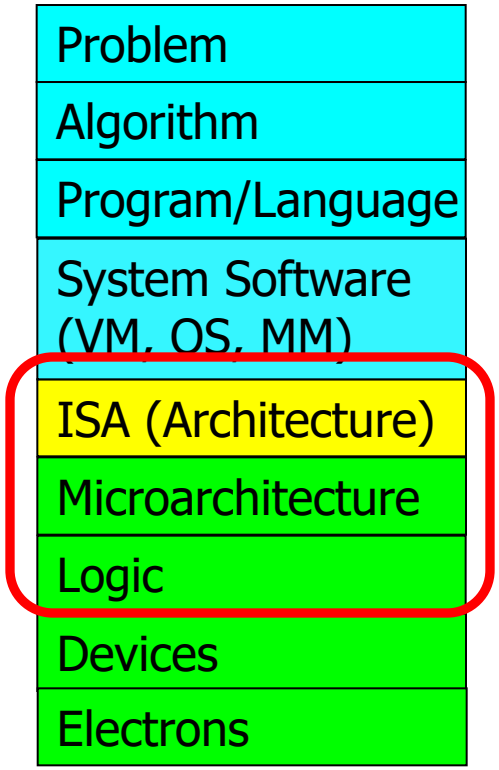


Algorithm

Step-by-step procedure that is **guaranteed to terminate** where **each step is precisely stated** and **can be carried out by a computer**

- **Finiteness**
- **Definiteness**
- **Effective computability**

Many algorithms for the same problem



ISA (Instruction Set Architecture)

Interface/contract between SW and HW.

What the programmer assumes hardware will satisfy.

Microarchitecture
An implementation of the ISA

Digital logic circuits
Building blocks of micro-arch (e.g., gates)

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

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

Now That We Know What an ISA Is…

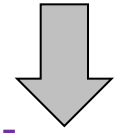
- How do we implement it?

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

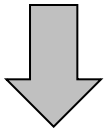
How Does a Machine Process Instructions?

- 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

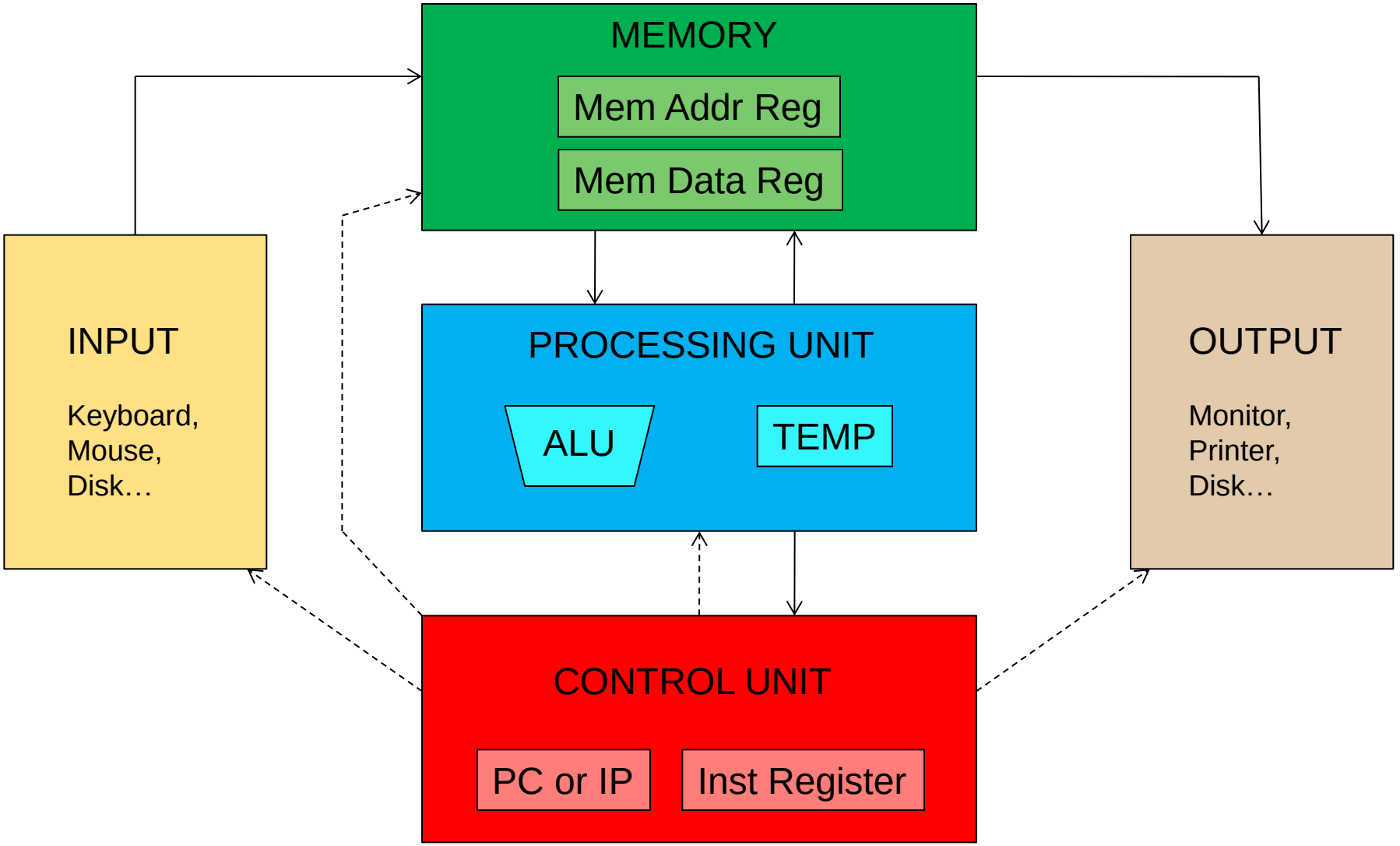
- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The Von Neumann Model/Architecture

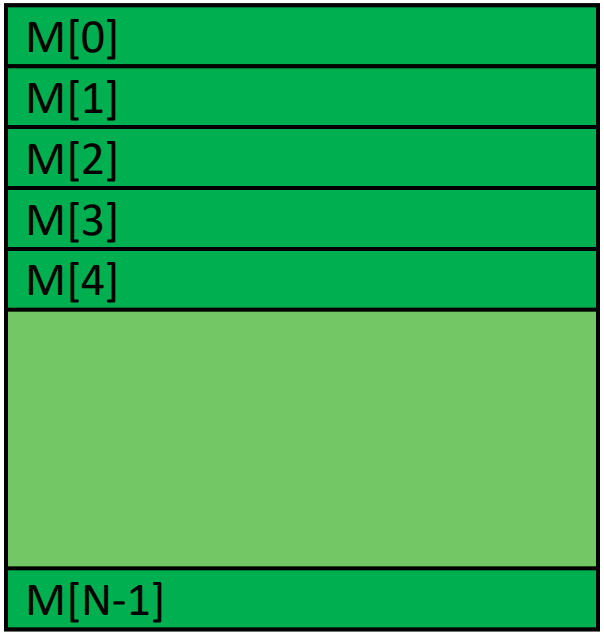
Stored program

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

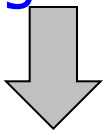
The "Process Instruction" Step

- 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')

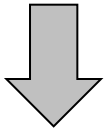
A Very Basic Instruction Processing Engine

- 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



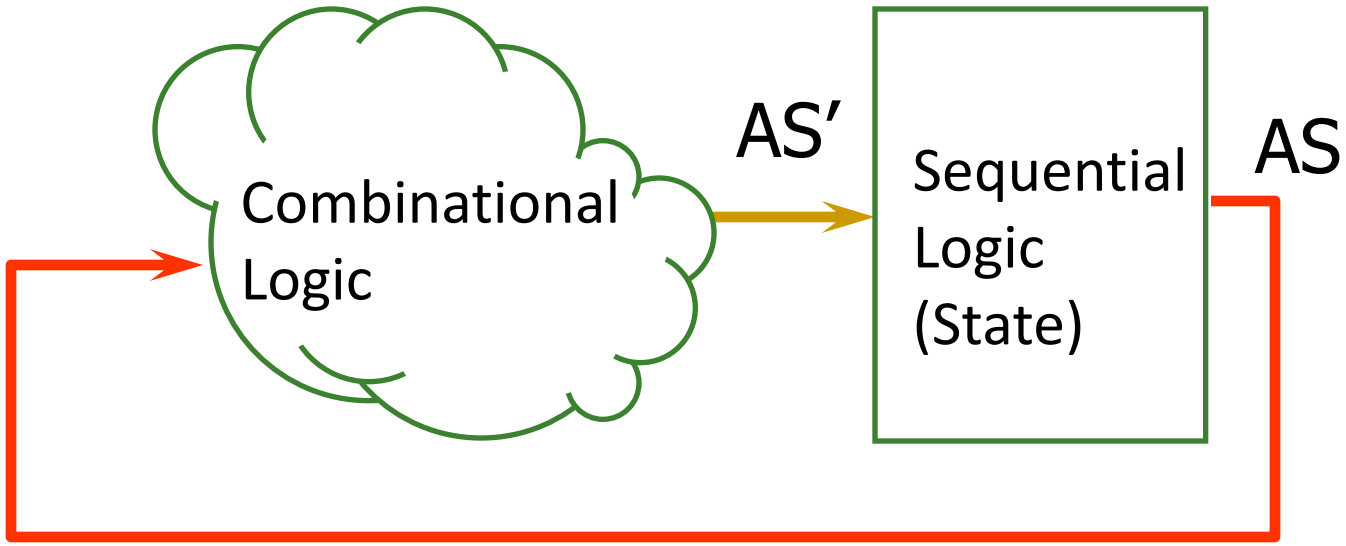
Process instruction in **one clock cycle**



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

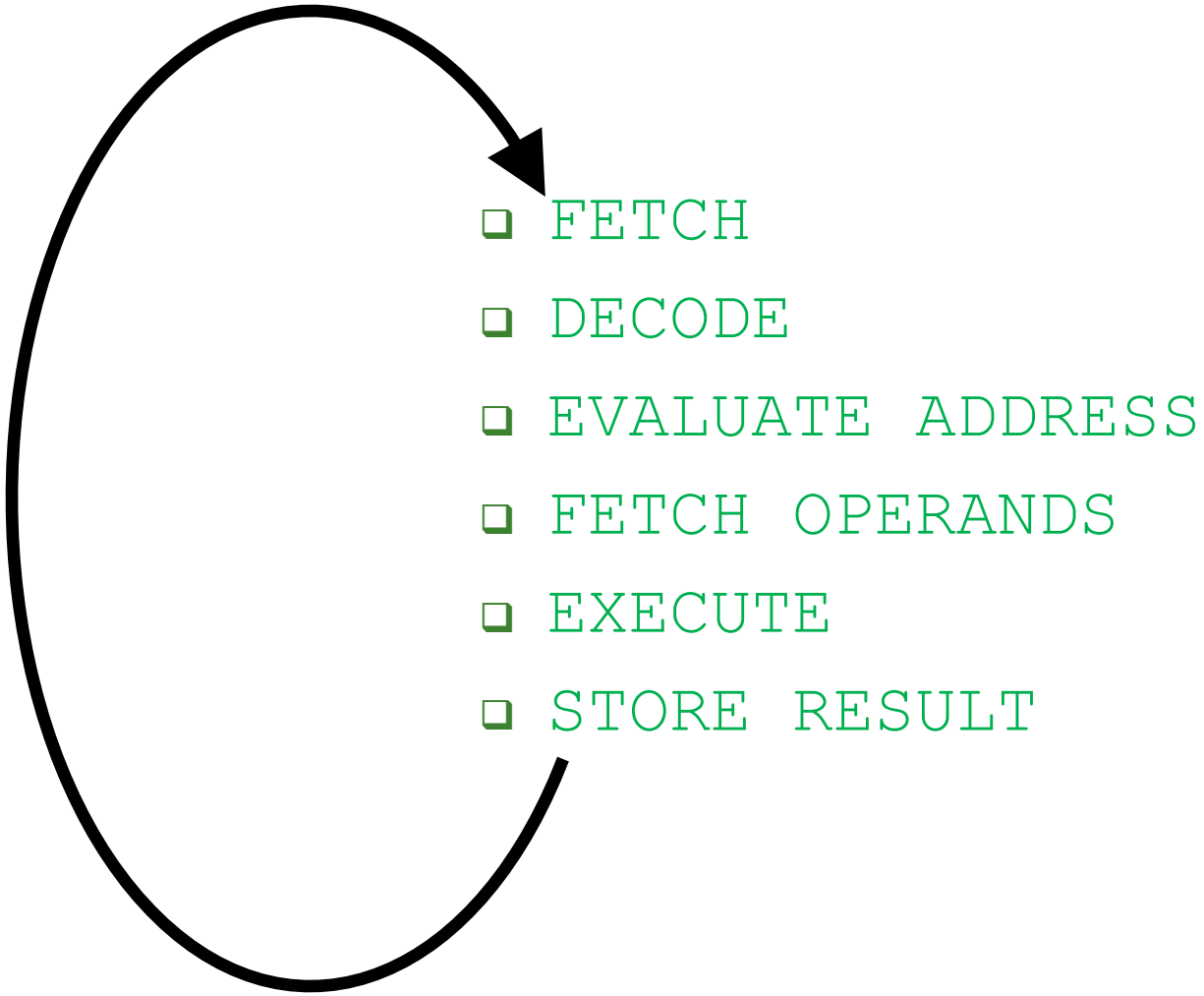
Instruction Processing "Cycle"

- 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"



Instruction Processing "Cycle" vs. Machine Clock Cycle

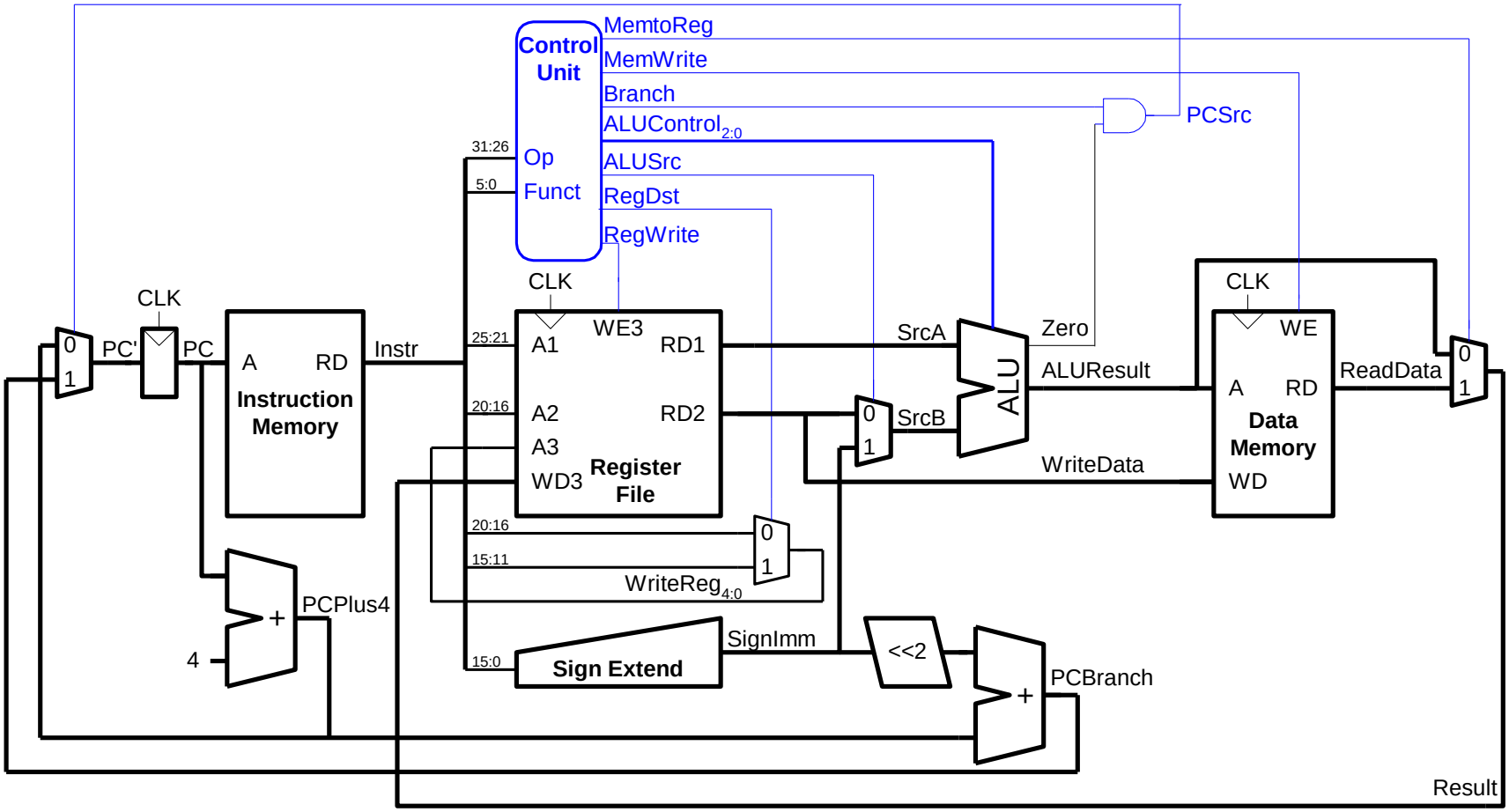
- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock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



A Single-Cycle Microarchitecture: Analysis

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

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

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

What is the Slowest Instruction to Process?

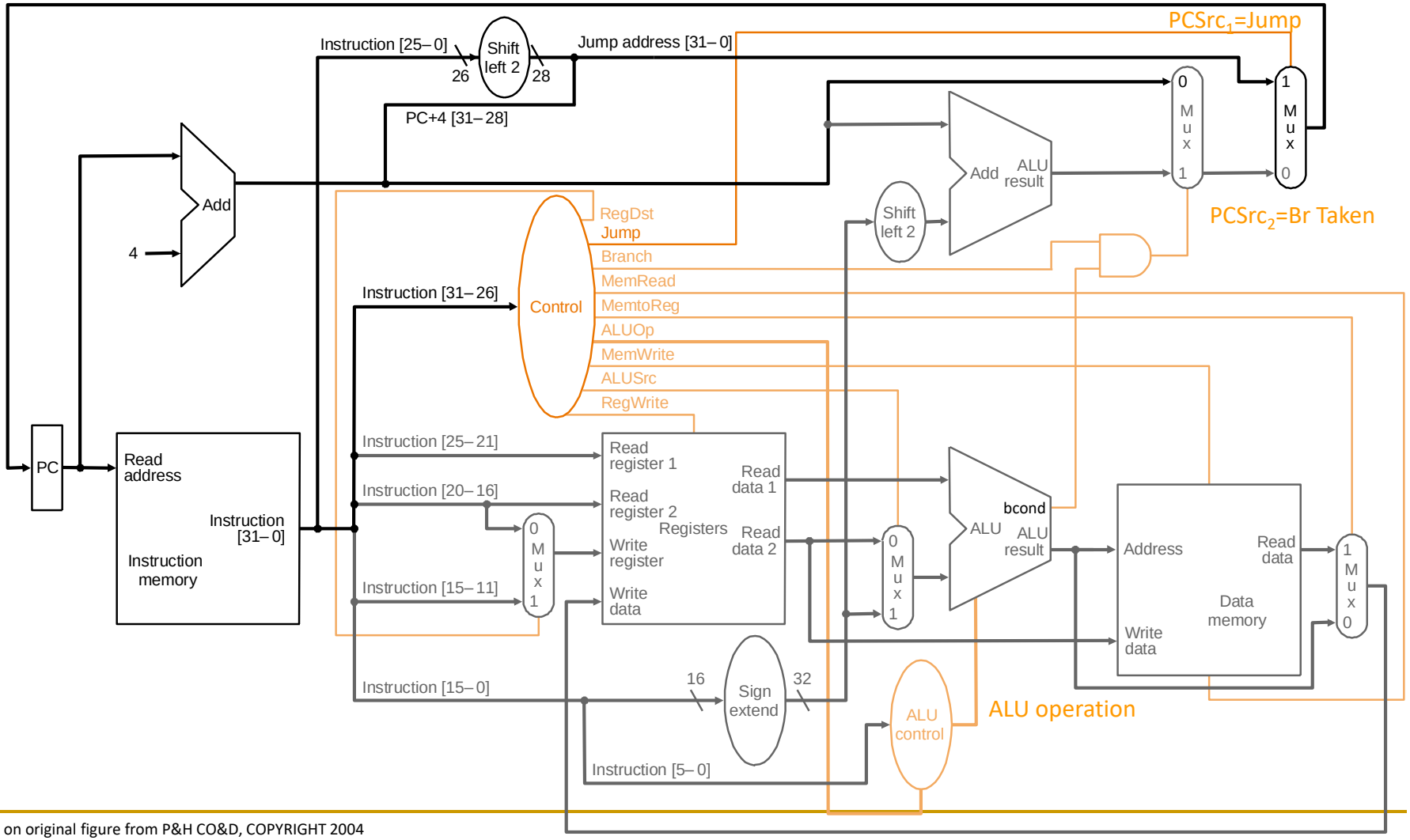
- Let's go back to the basics

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

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

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

Let's Find the Critical Path

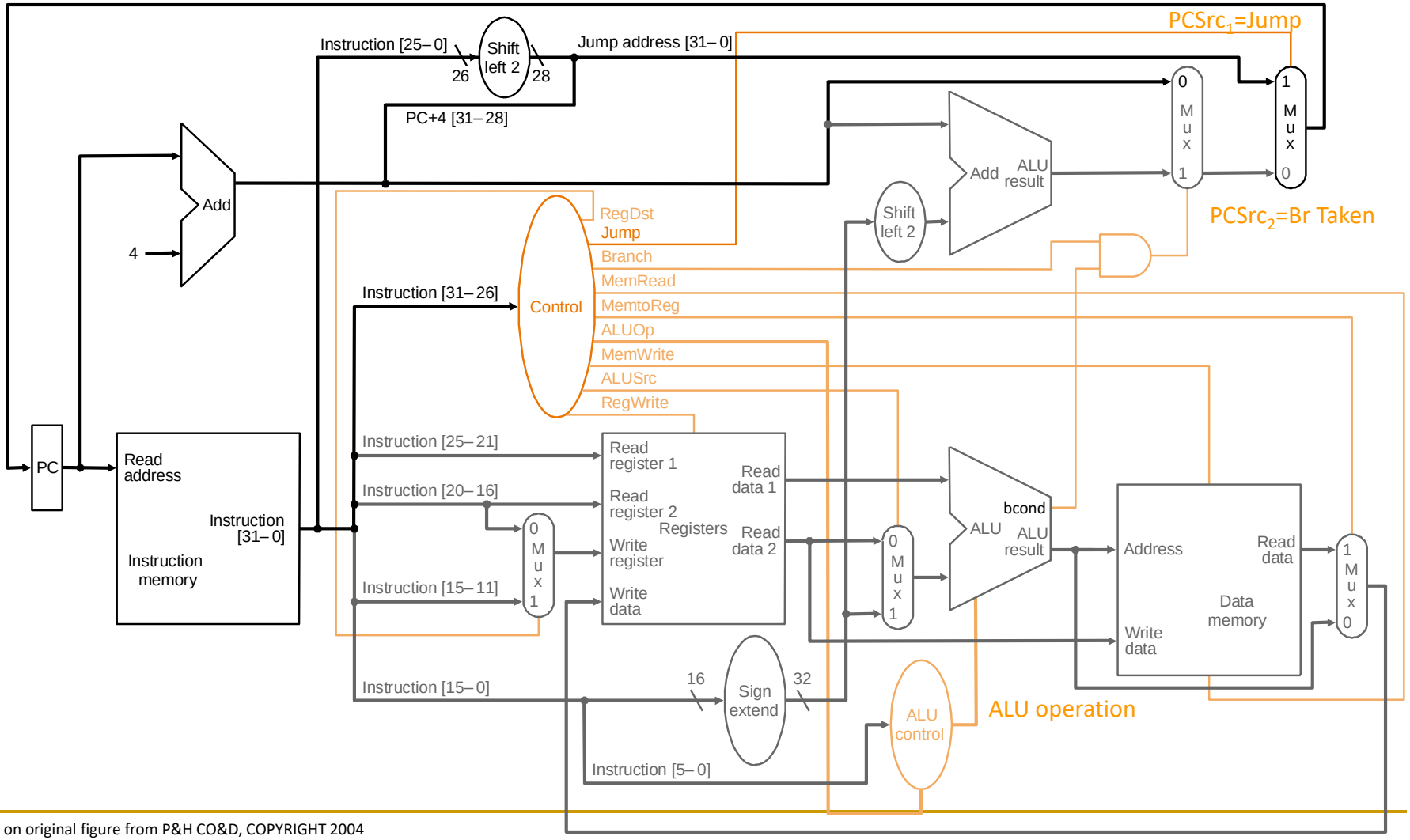


Example Single-Cycle Datapath Analysis

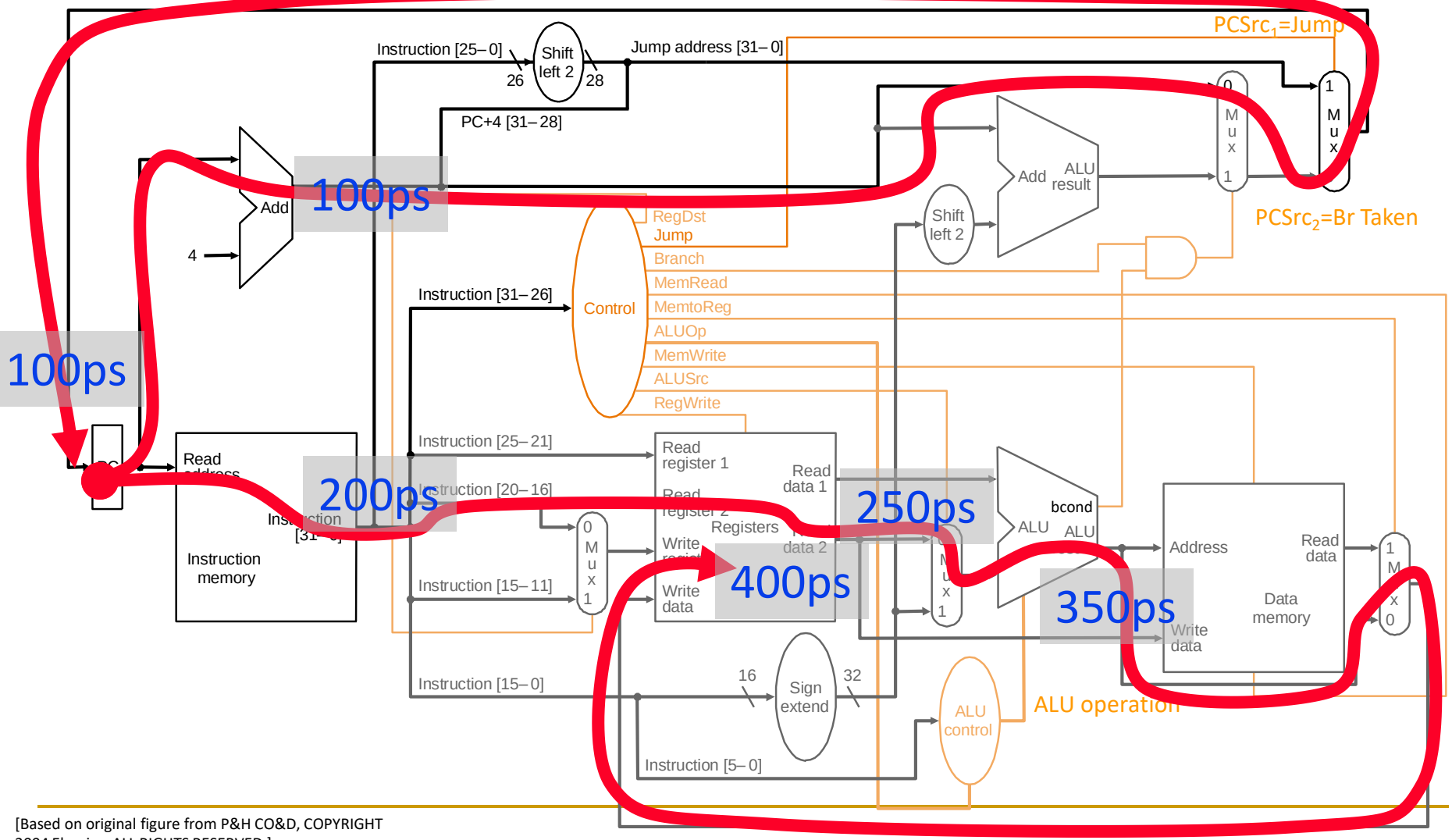
- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other logic or wire delay: 0 ps

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

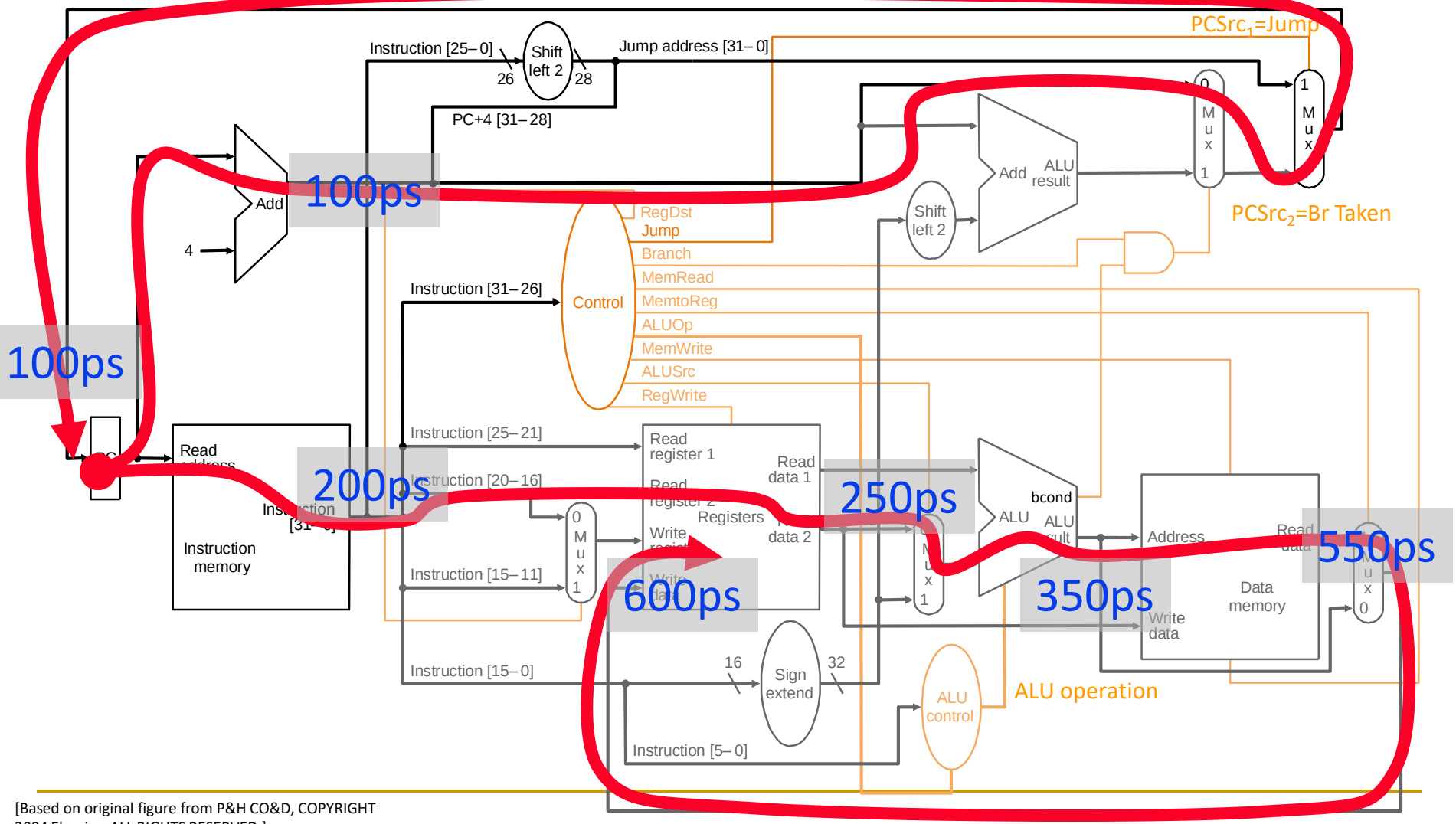
Let's Find the Critical Path



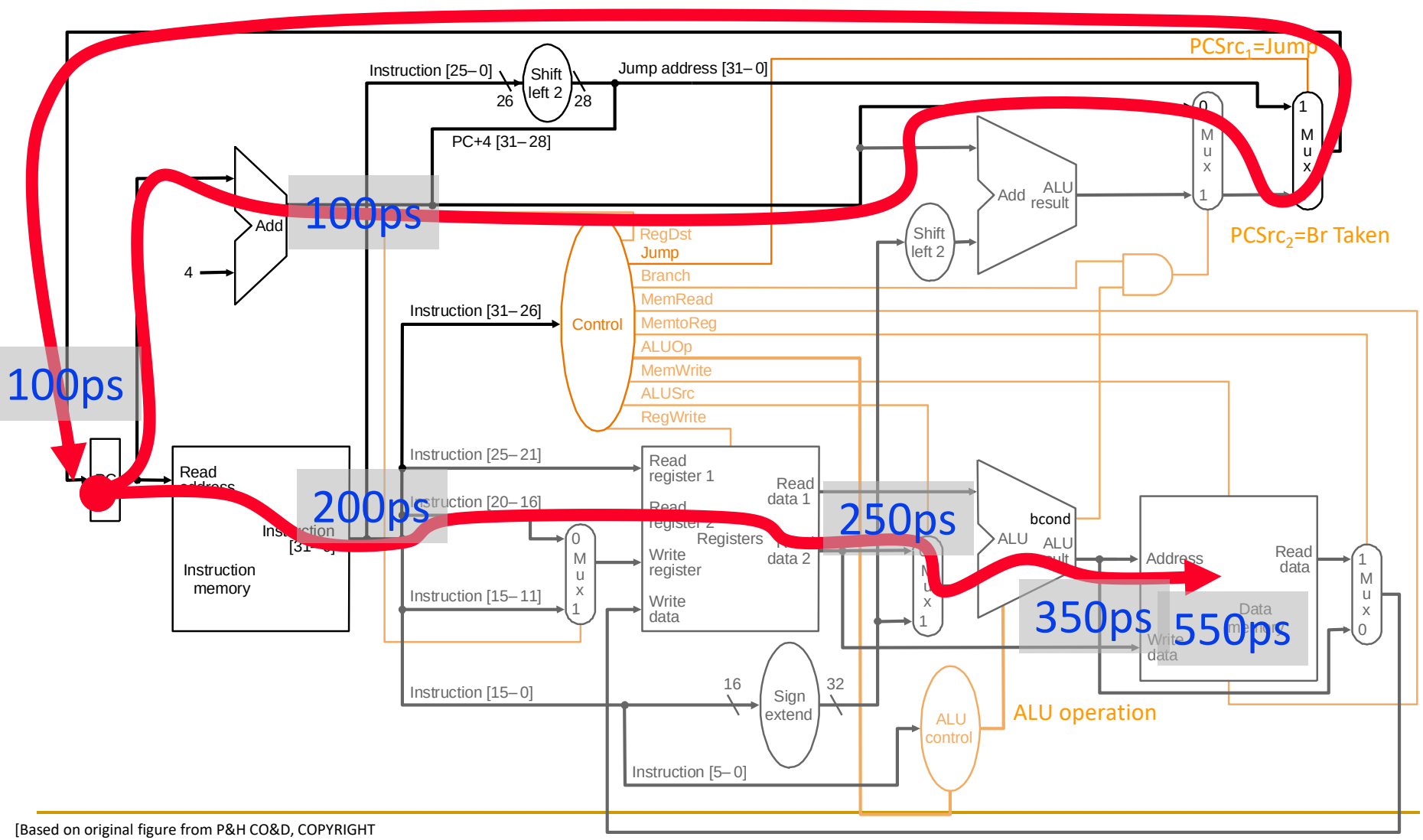
R-Type and I-Type ALU



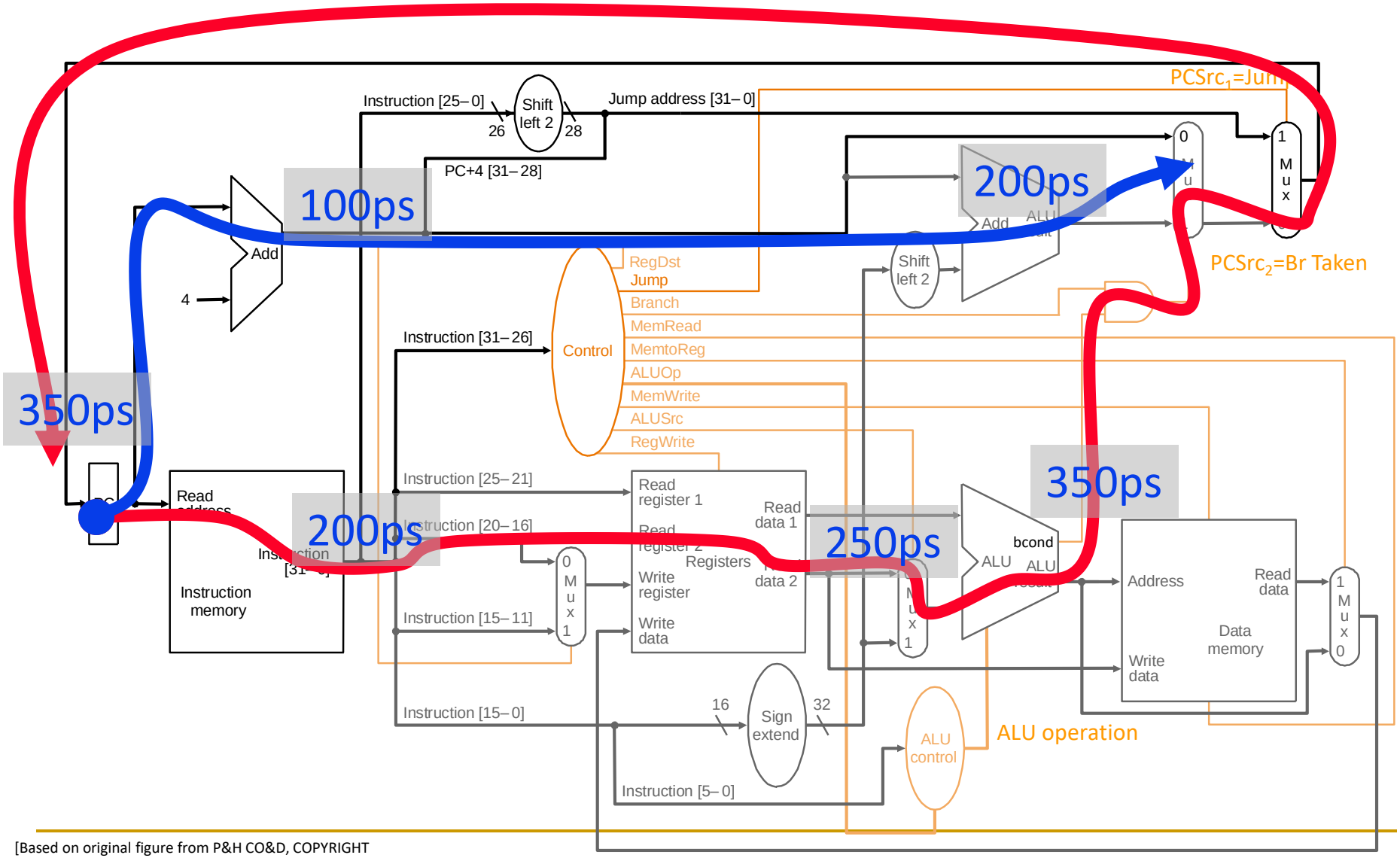
LW



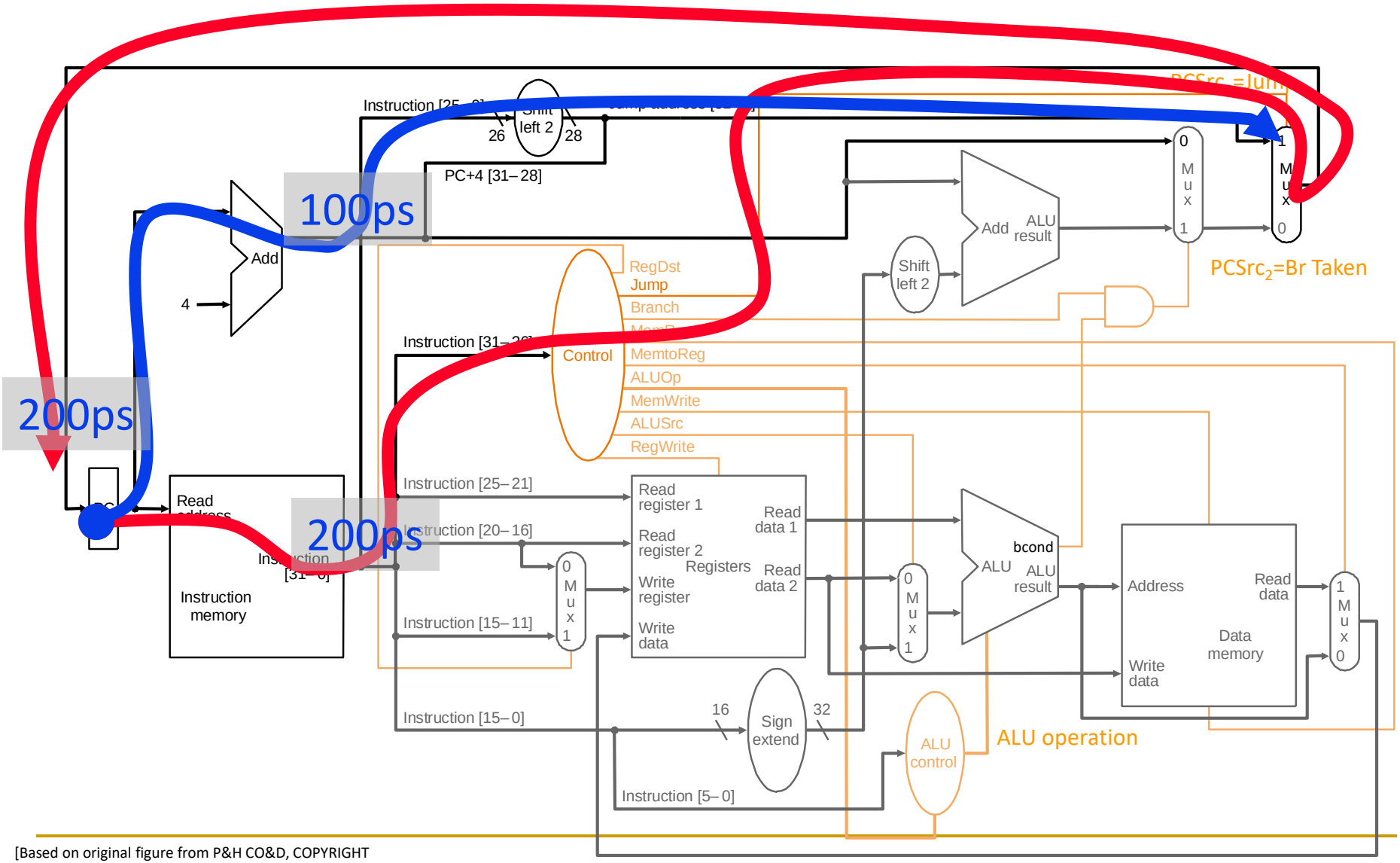
SW



Branch Taken



Jump



Example Single-Cycle Datapath Analysis

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

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

(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?

Can We Do Better?

Multi-Cycle Microarchitectures

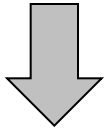
Multi-Cycle Microarchitectures

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

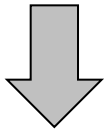
- Idea
  - Determine clock cycle time independently of instruction processing time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

Multi-Cycle Microarchitecture

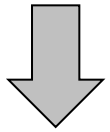
AS = Architectural (programmer visible) state
at the beginning of an instruction



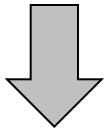
Step 1: Process part of instruction in one clock cycle



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



...



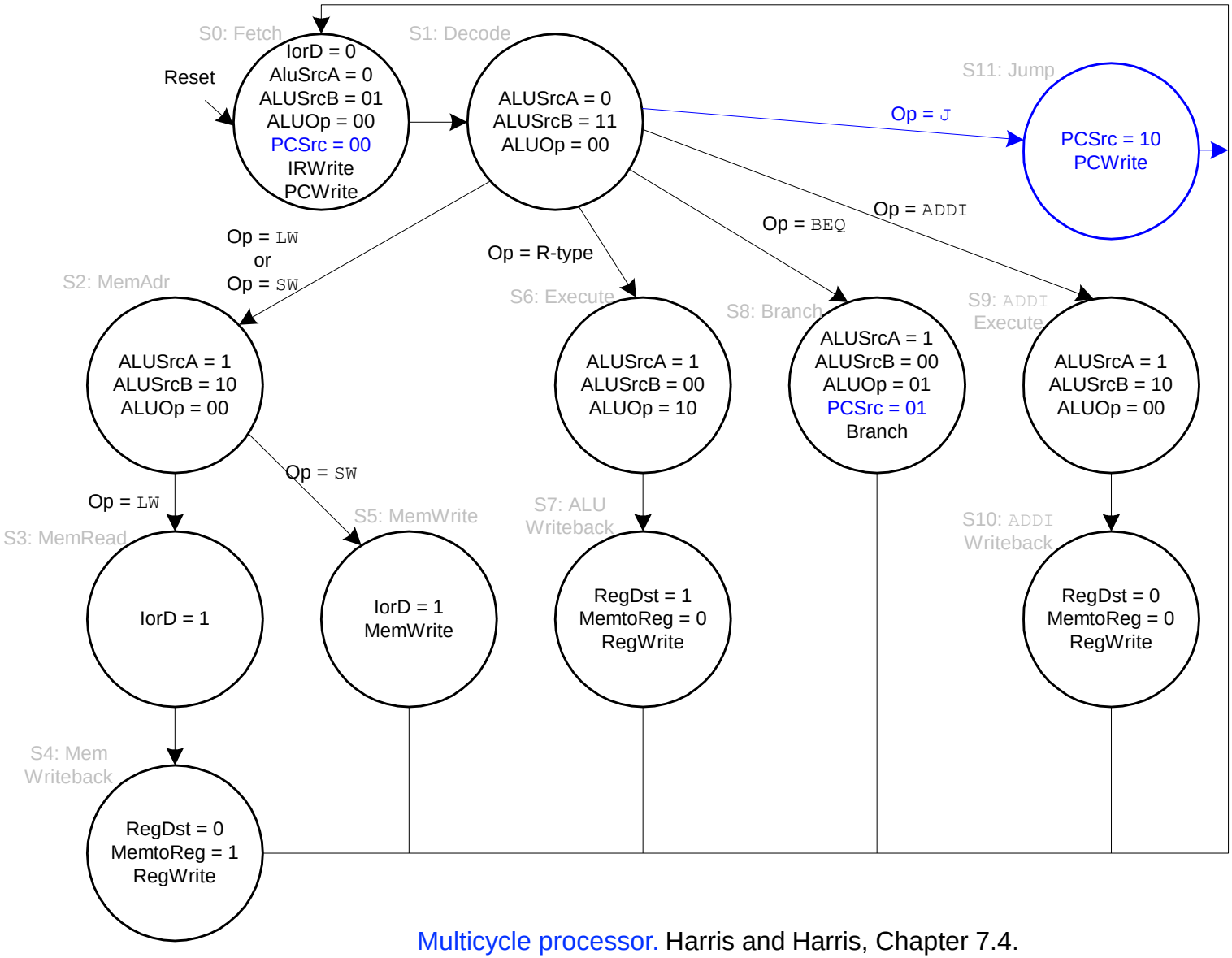
AS' = Architectural (programmer visible) state
at the end of a clock cycle

Multi-Cycle Microarchitectures

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

Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.

An Example Multi-Cycle FSM

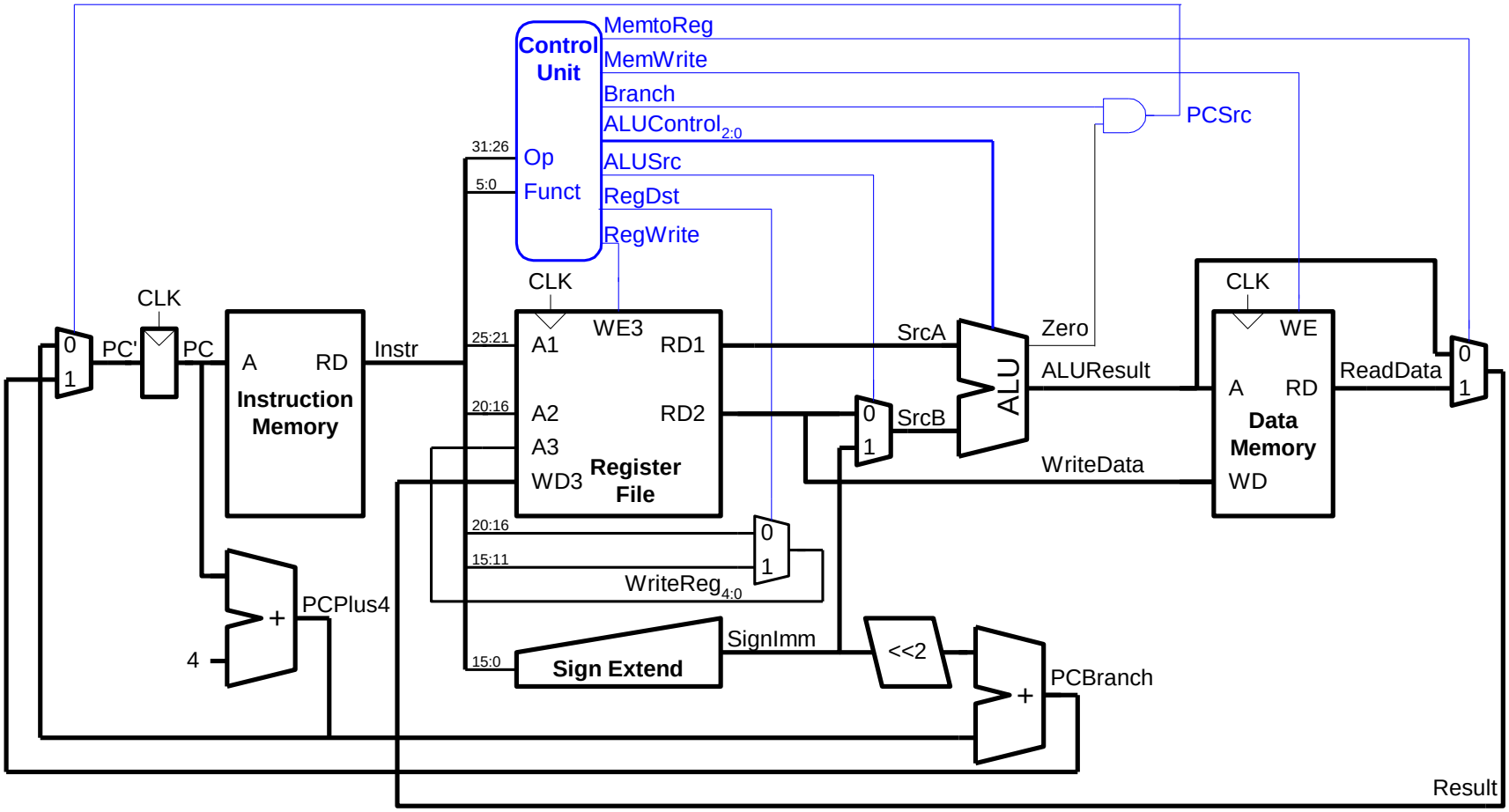


Multicycle processor. Harris and Harris, Chapter 7.4.

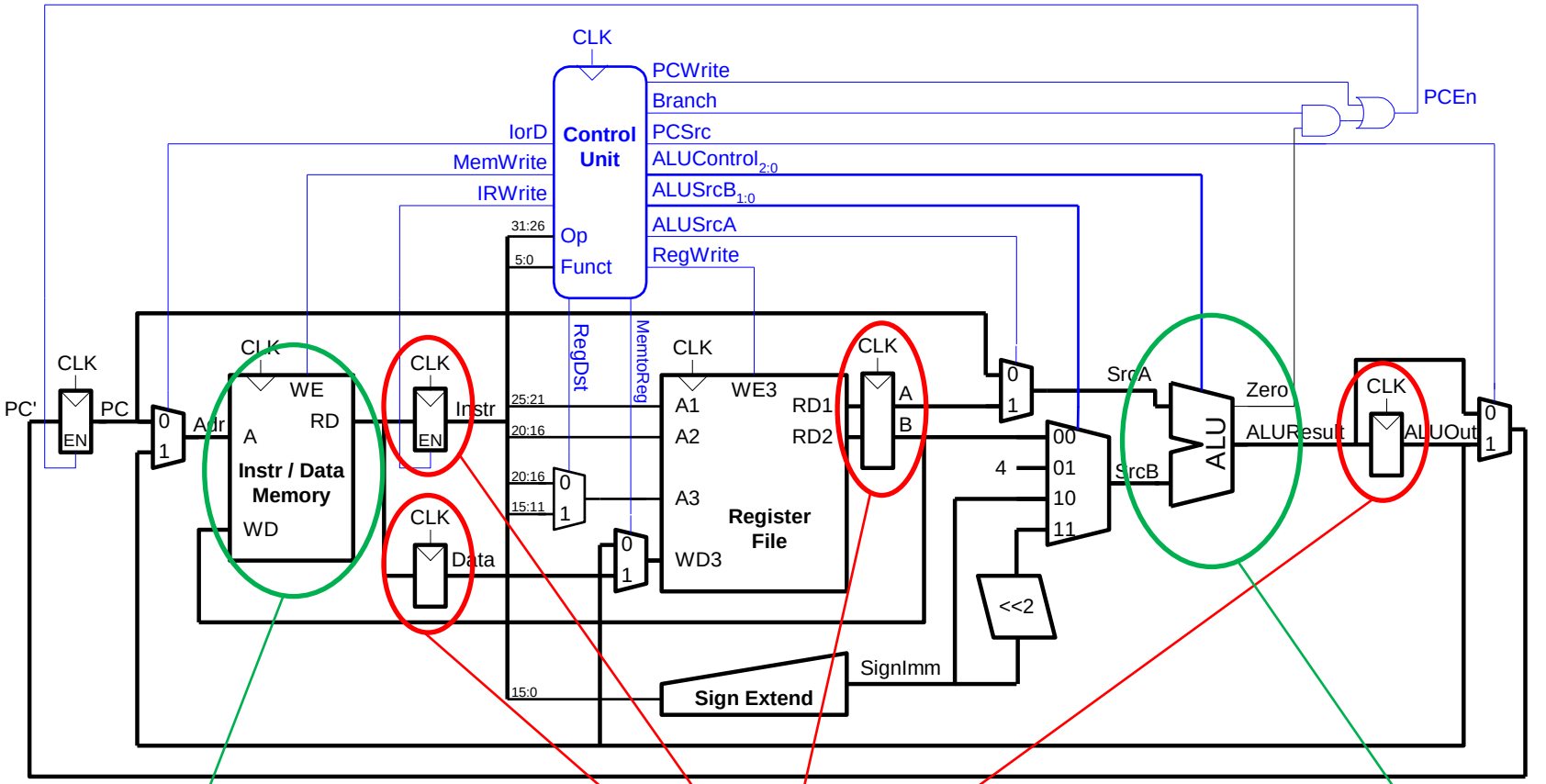
A Basic Multi-Cycle Microarchitecture

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

Recall: Single-Cycle Microarchitecture



Complete Multi-Cycle Microarchitecture



Only one memory

Extra registers not needed in a single-cycle design

Only one ALU/adder

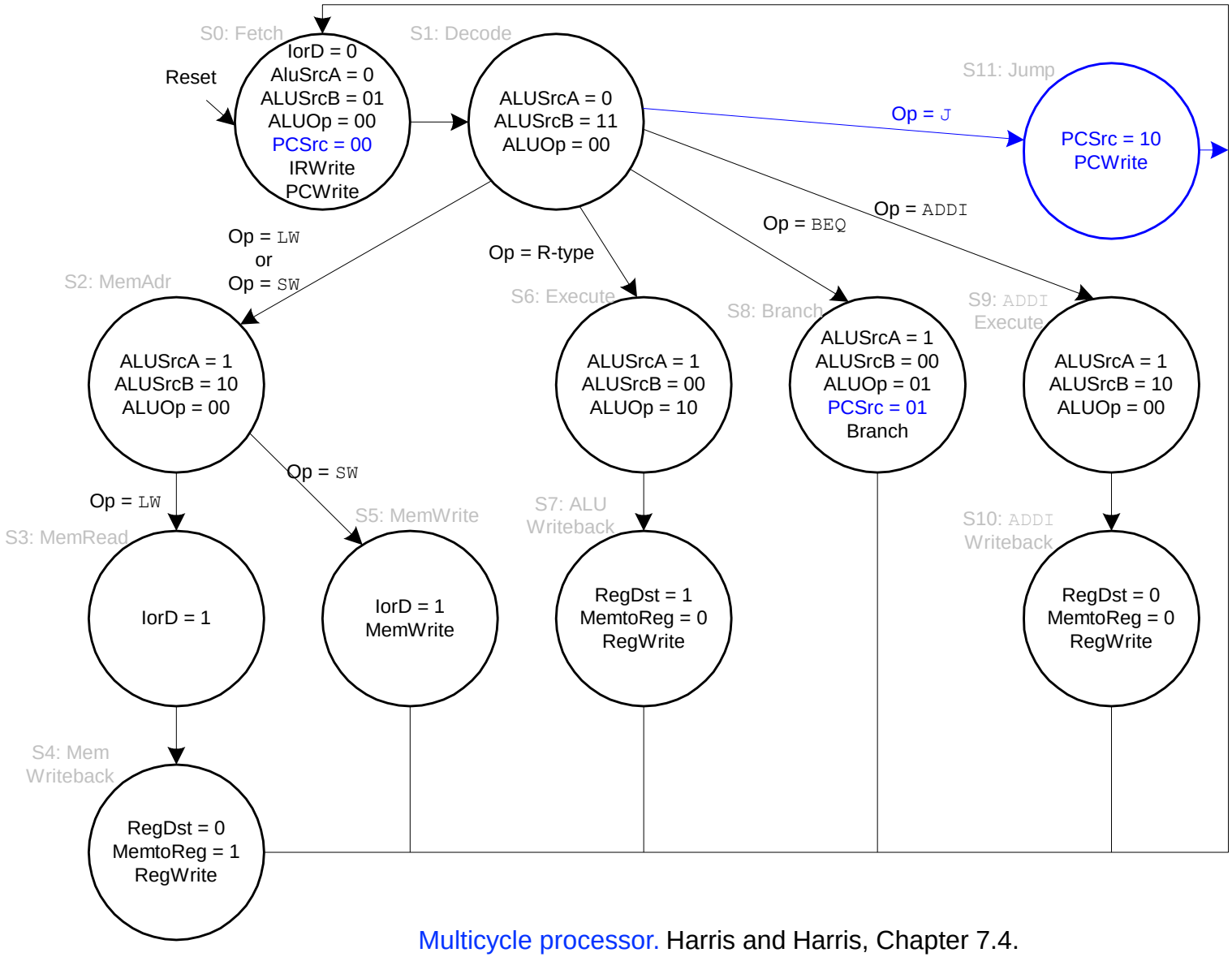
Why Do We Want Multi-Cycle?

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

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

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

Flexibility of Multi-Cycle Designs



What if memory takes 500 cycles?

Multicycle processor. Harris and Harris, Chapter 7.4.

Benefits of Multi-Cycle Design

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

Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
  - Hardware overhead for microarchitectural registers
  - Register setup/hold overhead (i.e., sequencing overhead) is paid multiple times for an instruction

- Limited concurrency
  - Only a small part of the machine is used at any point in time

Remember: 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

Can We Do Better?

Can We Do Better?

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

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

Can We Use the Idle Hardware to Improve Concurrency?

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

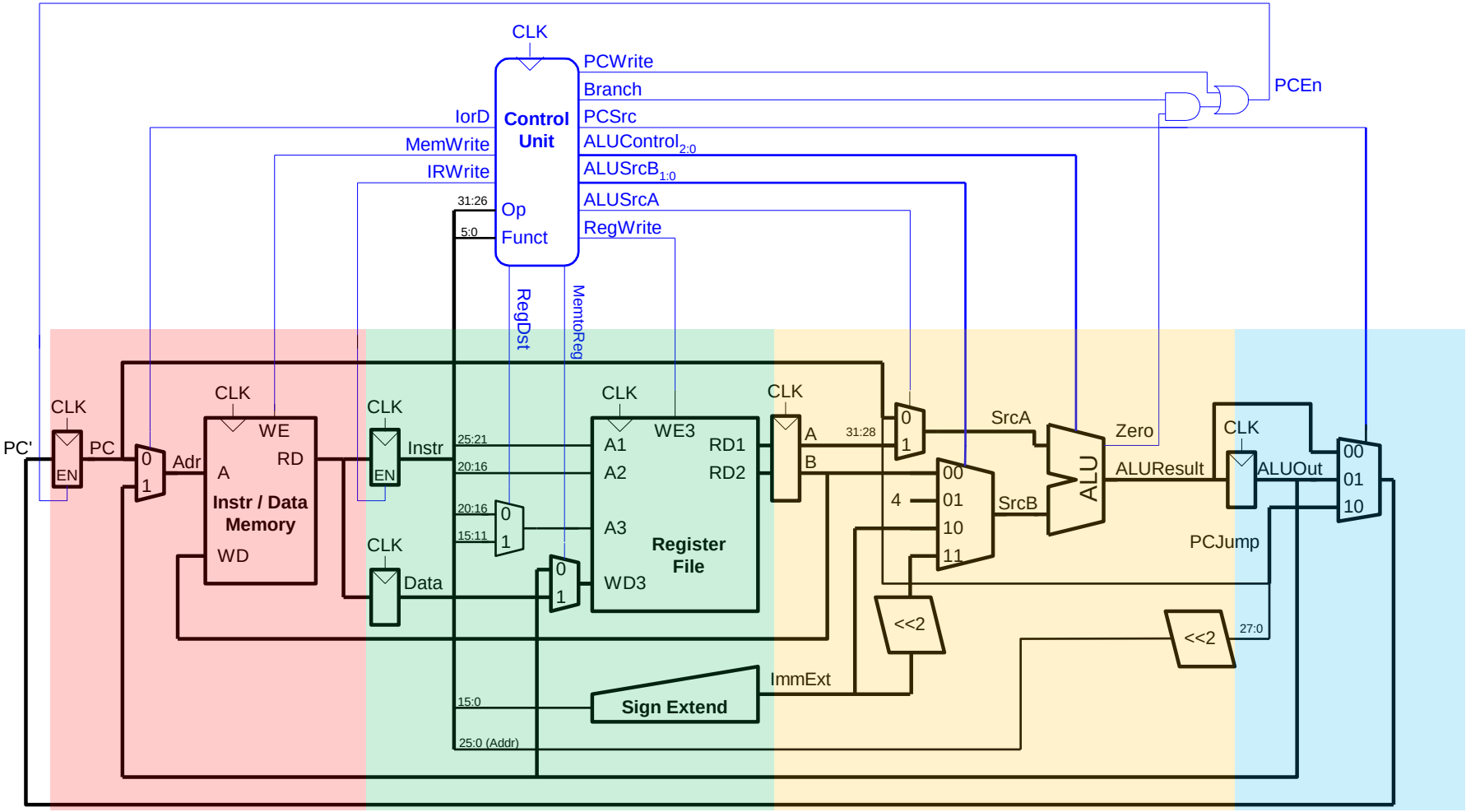
- Idea: When an instruction is using some resources in its processing phase, process **other instructions** on **idle resources** not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction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

Can Have Different Instructions in Different Stages

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

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

Can Have Different Instructions in Different Stages



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

Pipelining

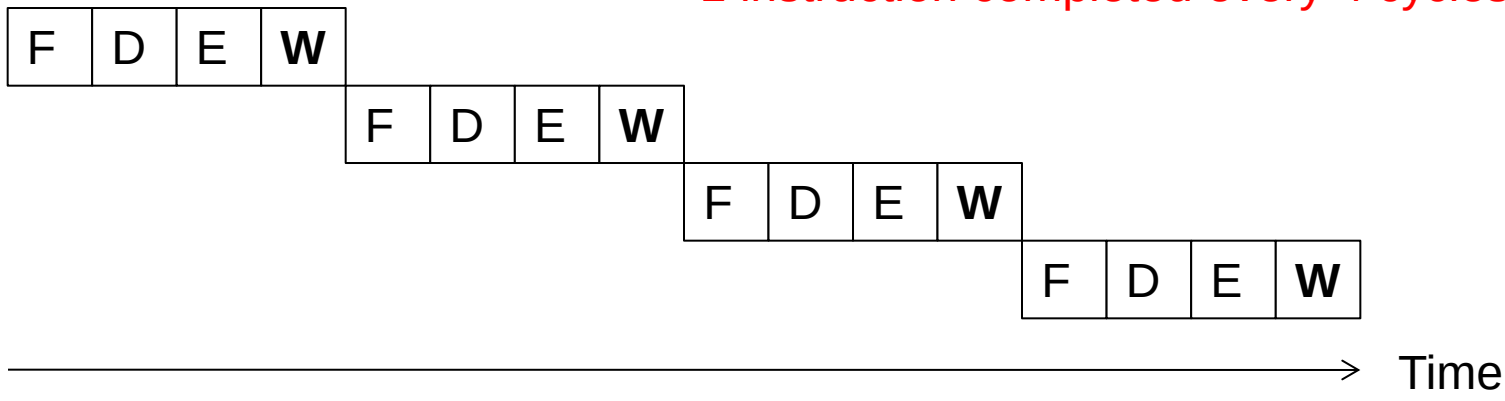
Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside(s): Start thinking about this…

Example: Execution of Four Independent ADDs

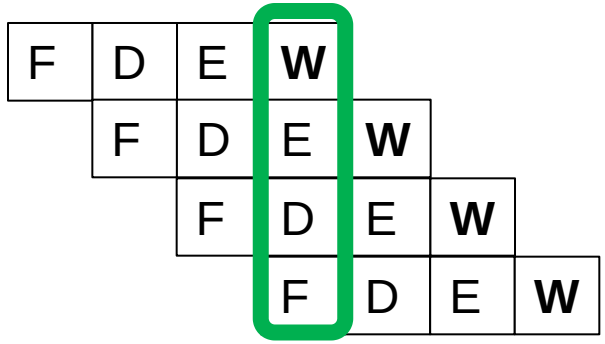
- Multi-cycle: 4 cycles per instruction

1 instruction completed every 4 cycles



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

1 instruction completed every 1 cycle



Is life always this beautiful?

An Old Pipelined Computer: IBM Stretch

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

V • T • E

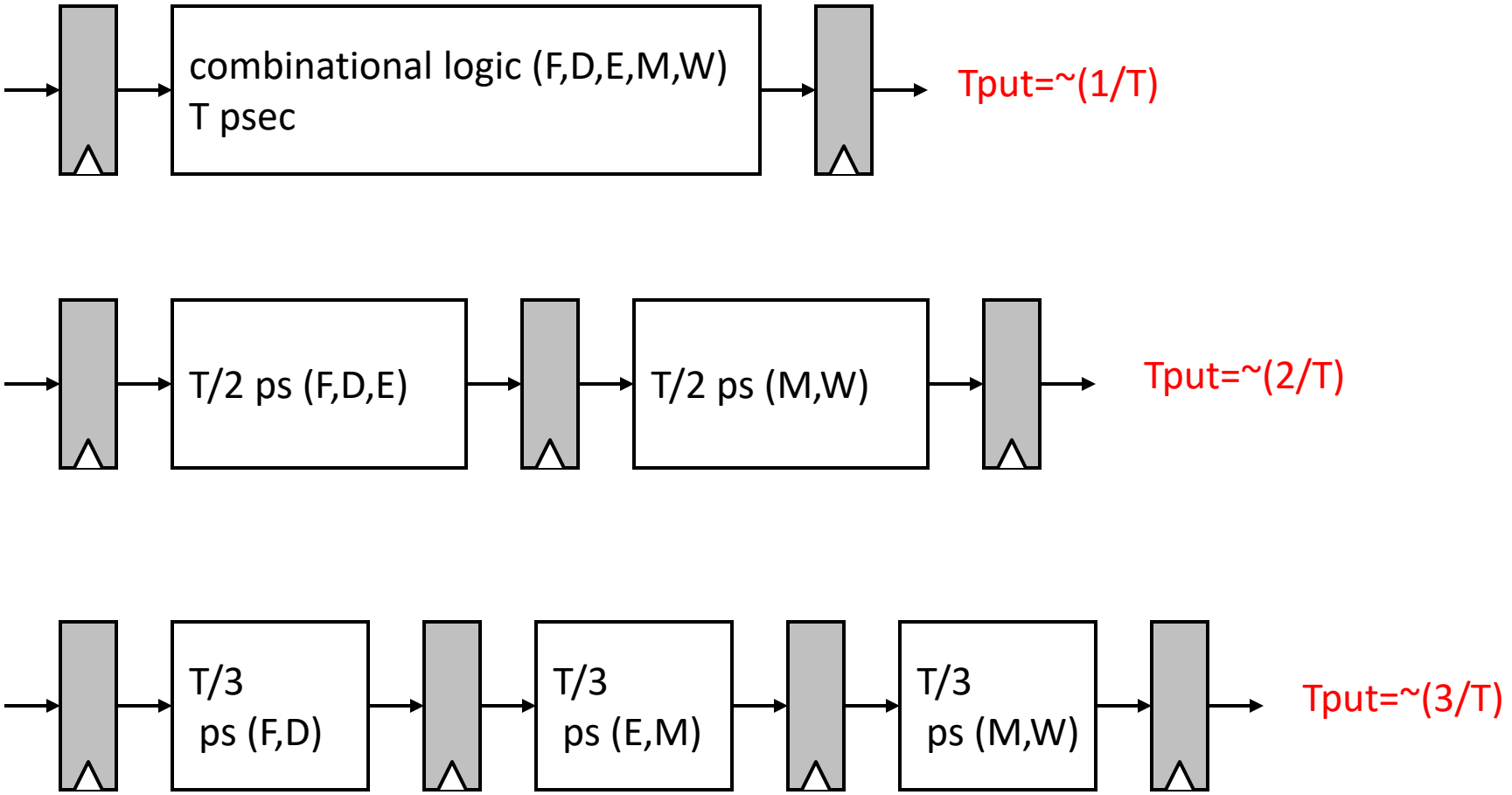
https://en.wikipedia.org/wiki/IBM_7030_Stretch

An Ideal Pipeline

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

Ideal Pipelining

Tput = Throughput



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

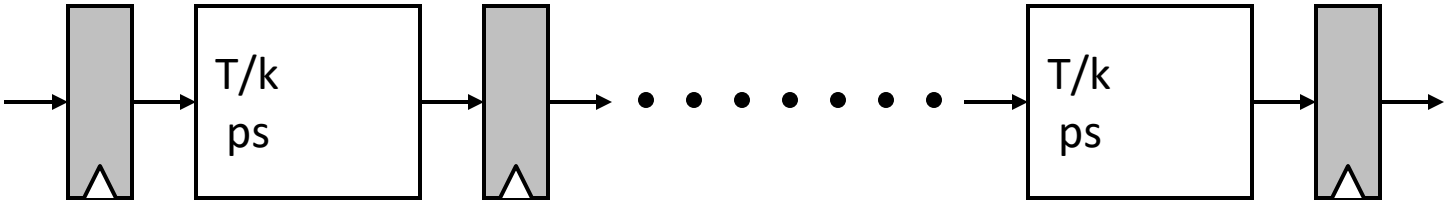
  Tput = 1 / (T+S) where S = register (sequential logic) delay

- k-stage pipelined version

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

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

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



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

More Realistic Pipeline: Cost

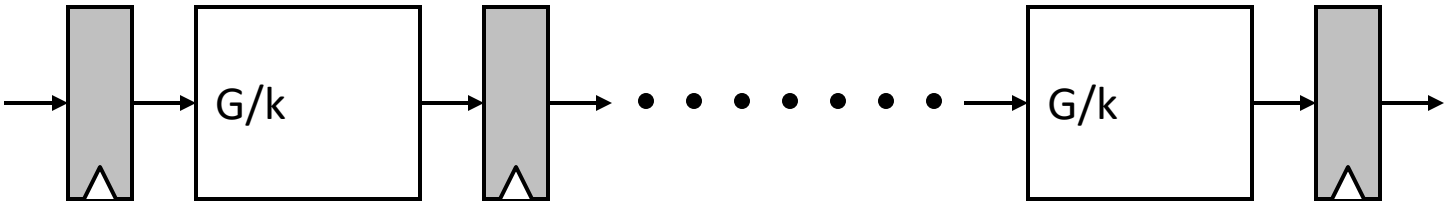
- Nonpipelined version with combinational cost G

  Cost = G+R where R = register cost

- k-stage pipelined version

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

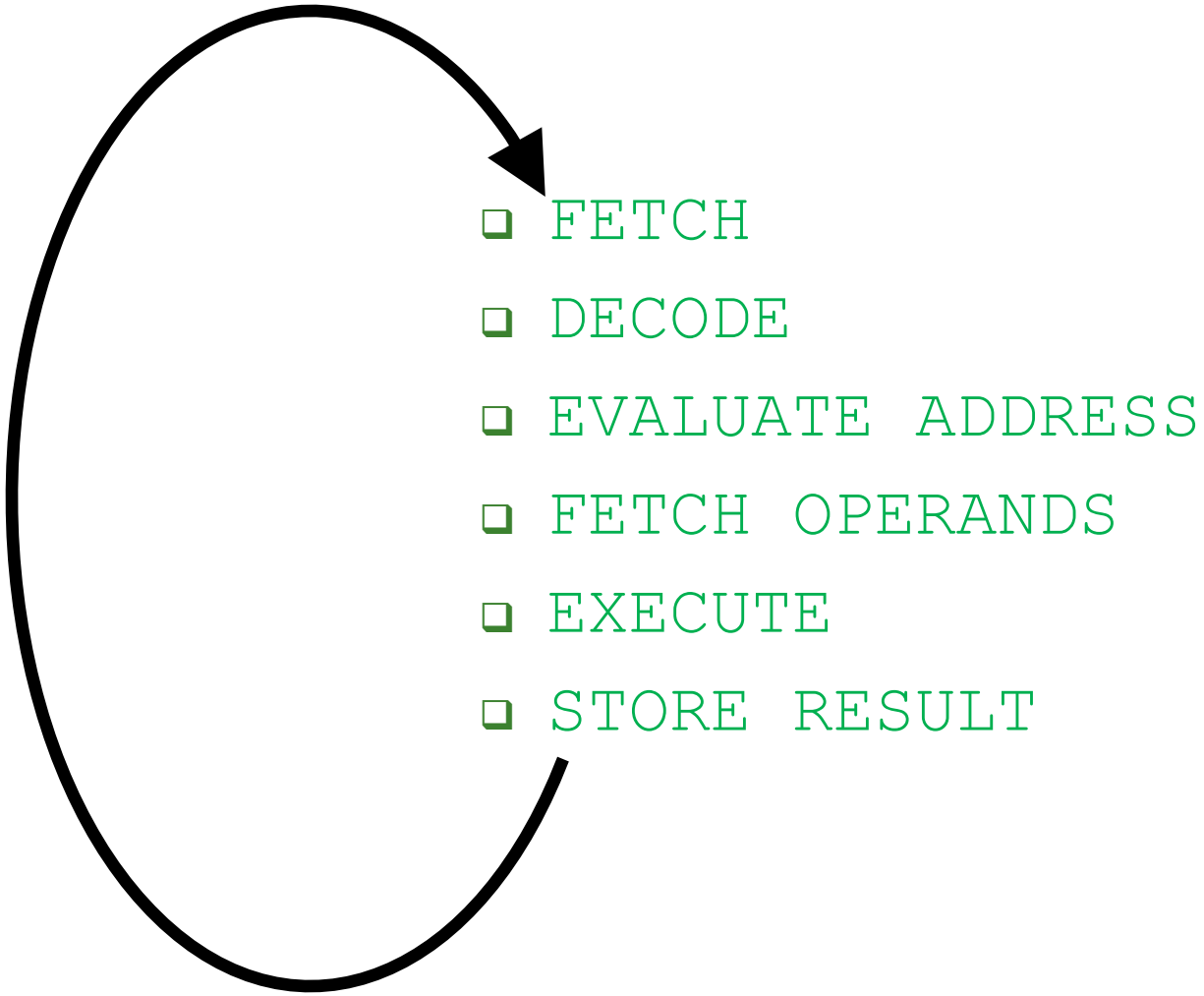
Registers increase hardware cost



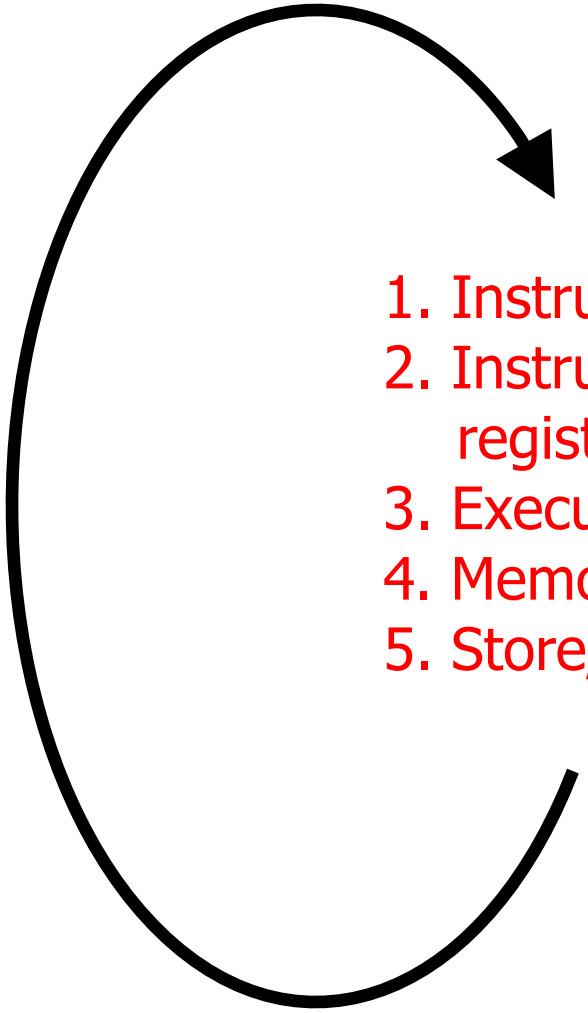
This picture ignores resource and register replication that is likely needed (G/k and Rk)

Pipelining Instruction Processing

Remember: The Instruction Processing Cycle

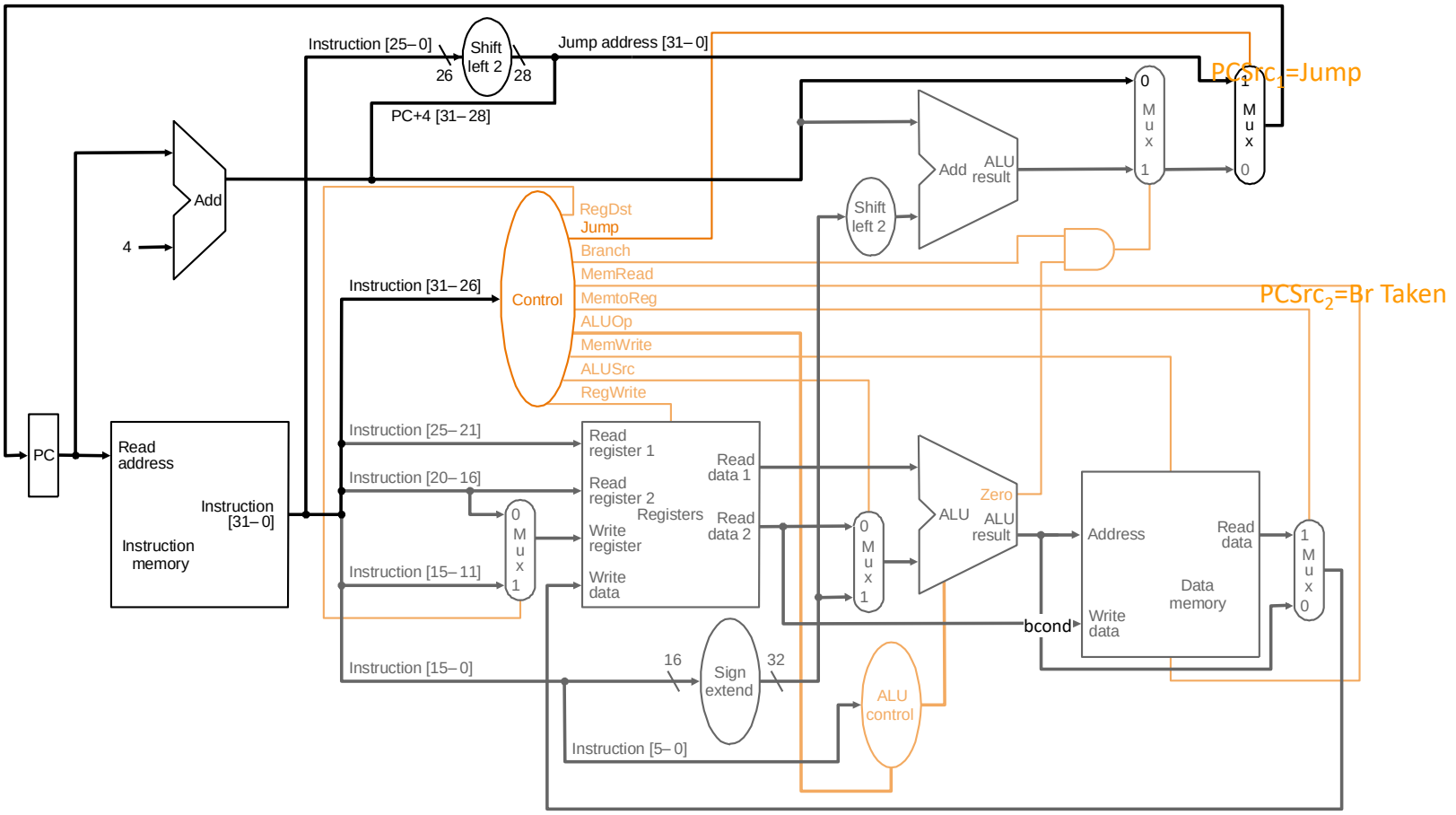


Remember: The Instruction Processing Cycle



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)

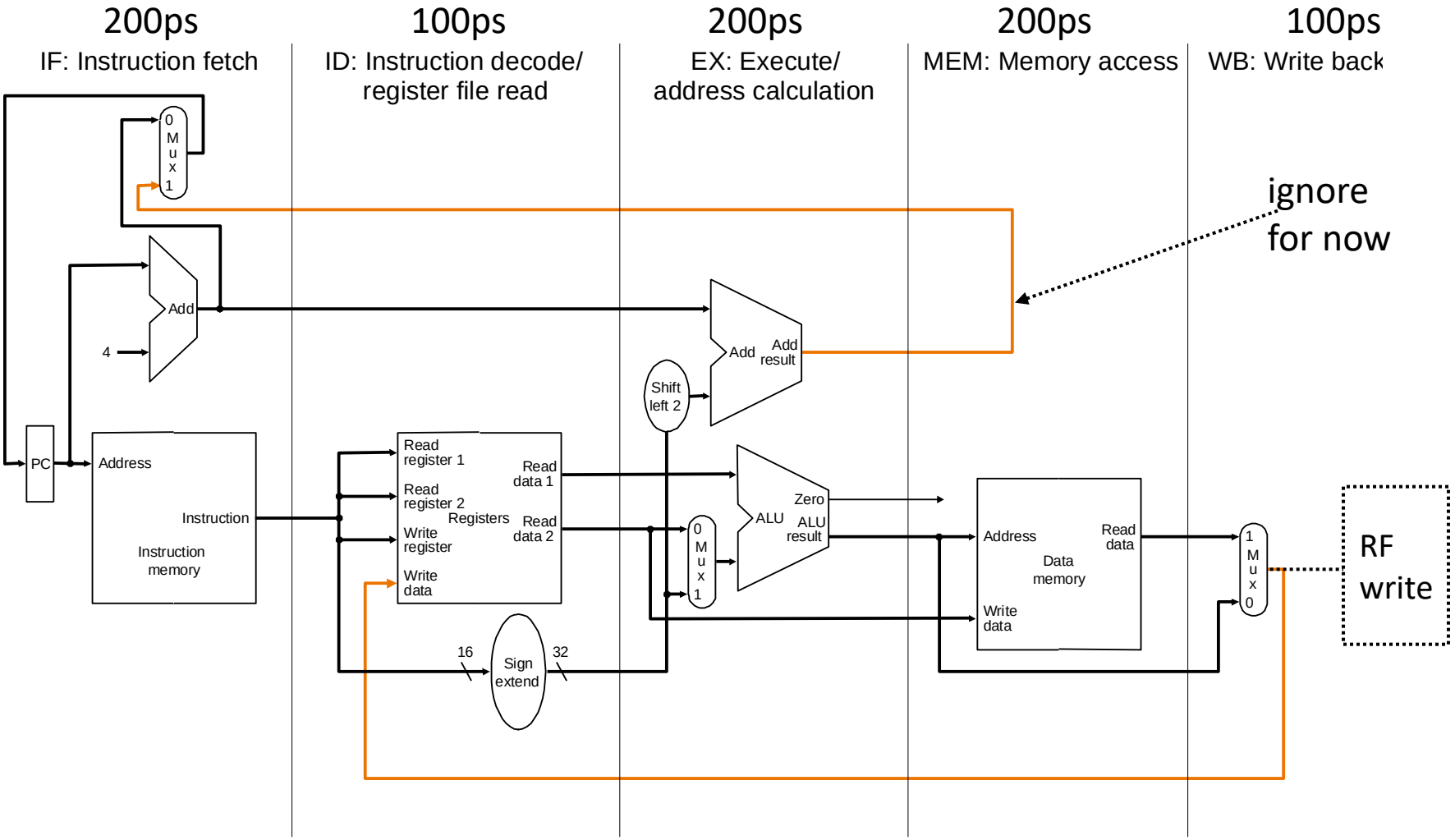
Remember the Single-Cycle Microarchitecture



ALU operation

Tput=~(1/T)

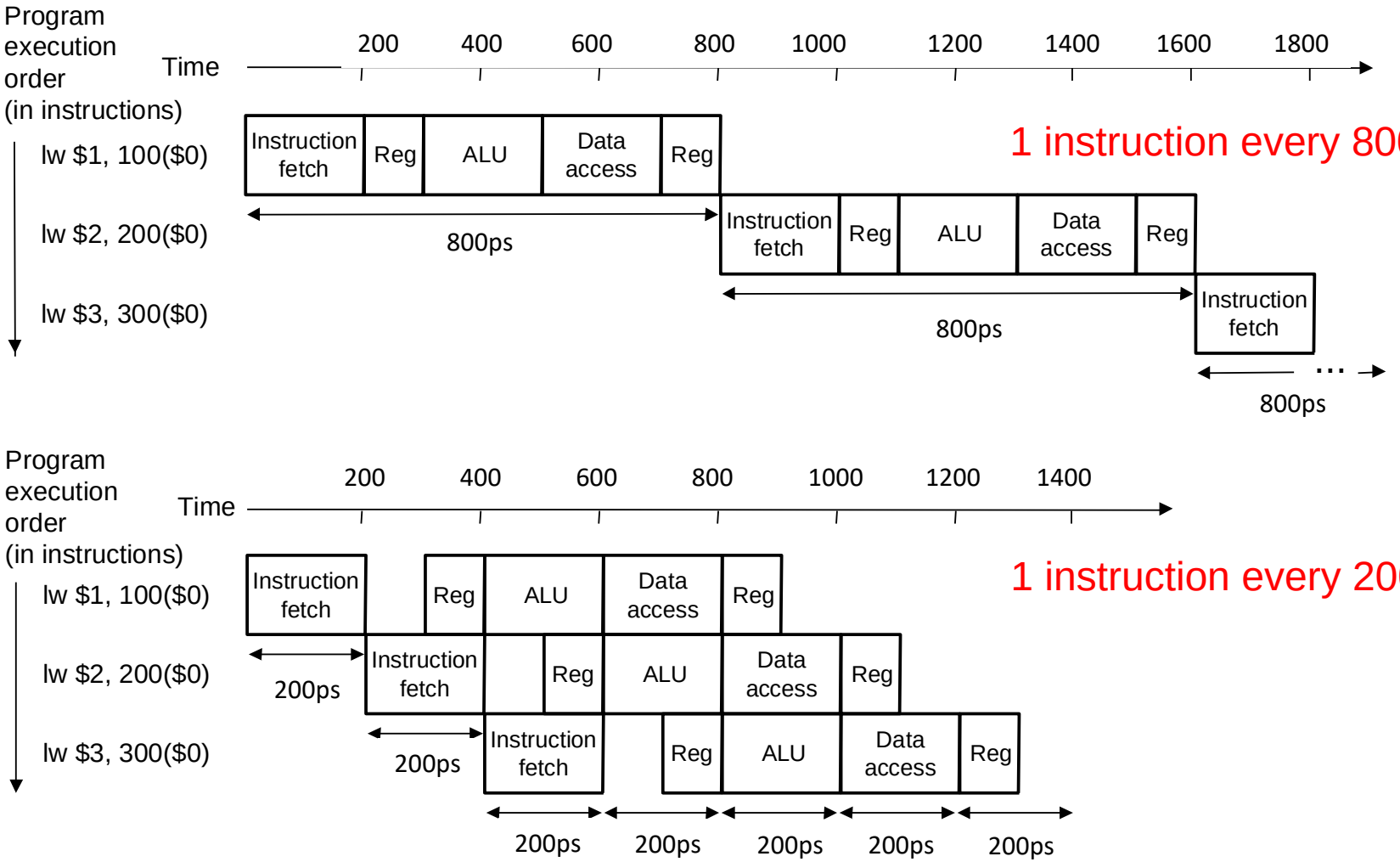
Dividing the Single-Cycle Uarch Into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

Instruction Pipeline Throughput

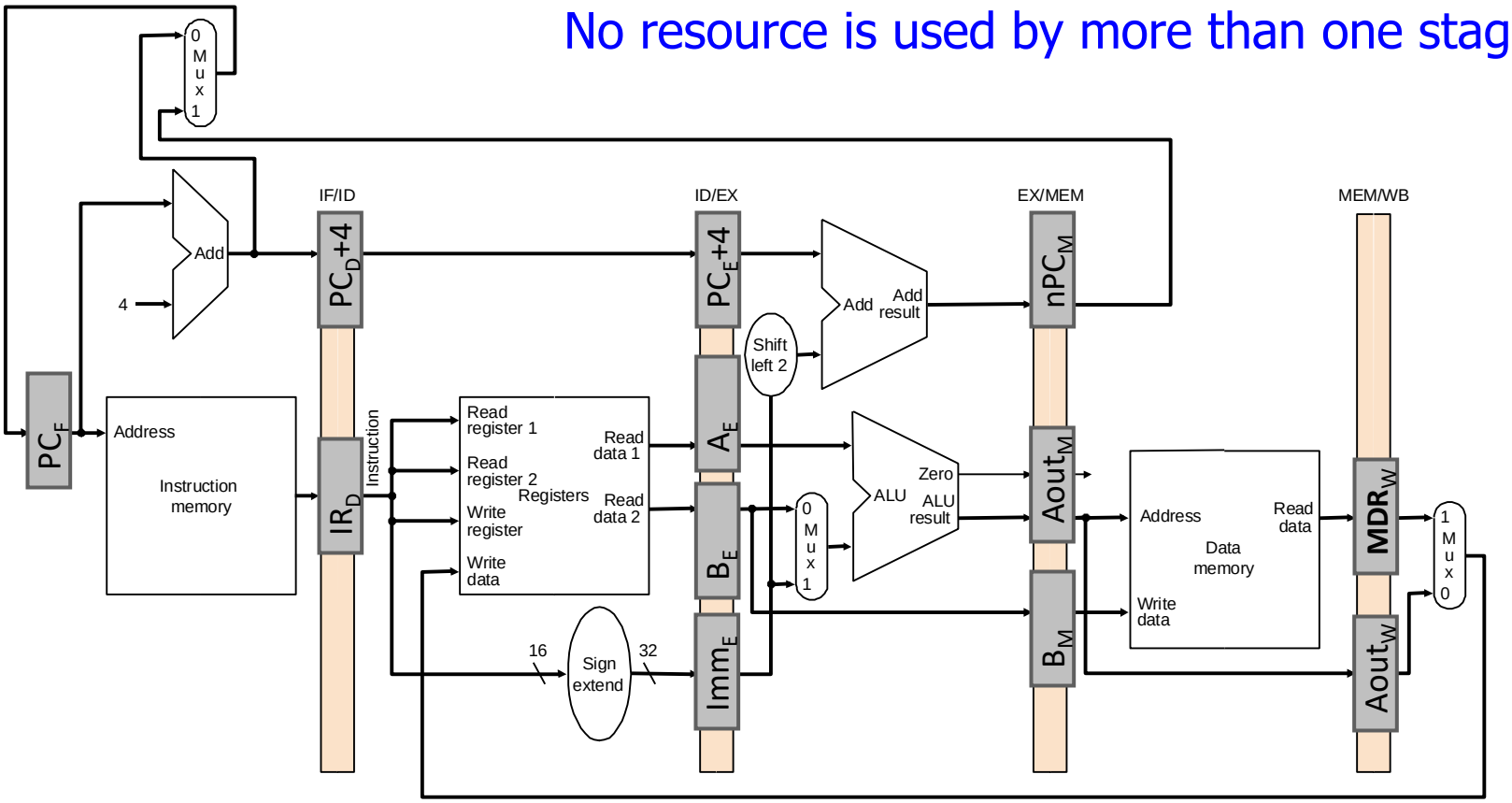


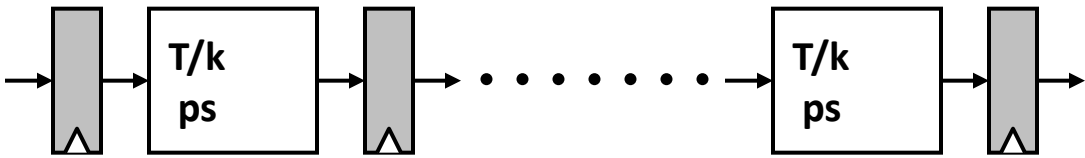
5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

Half of the clock cycle is wasted in two stages (ID/RF and WB)

Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than one stage

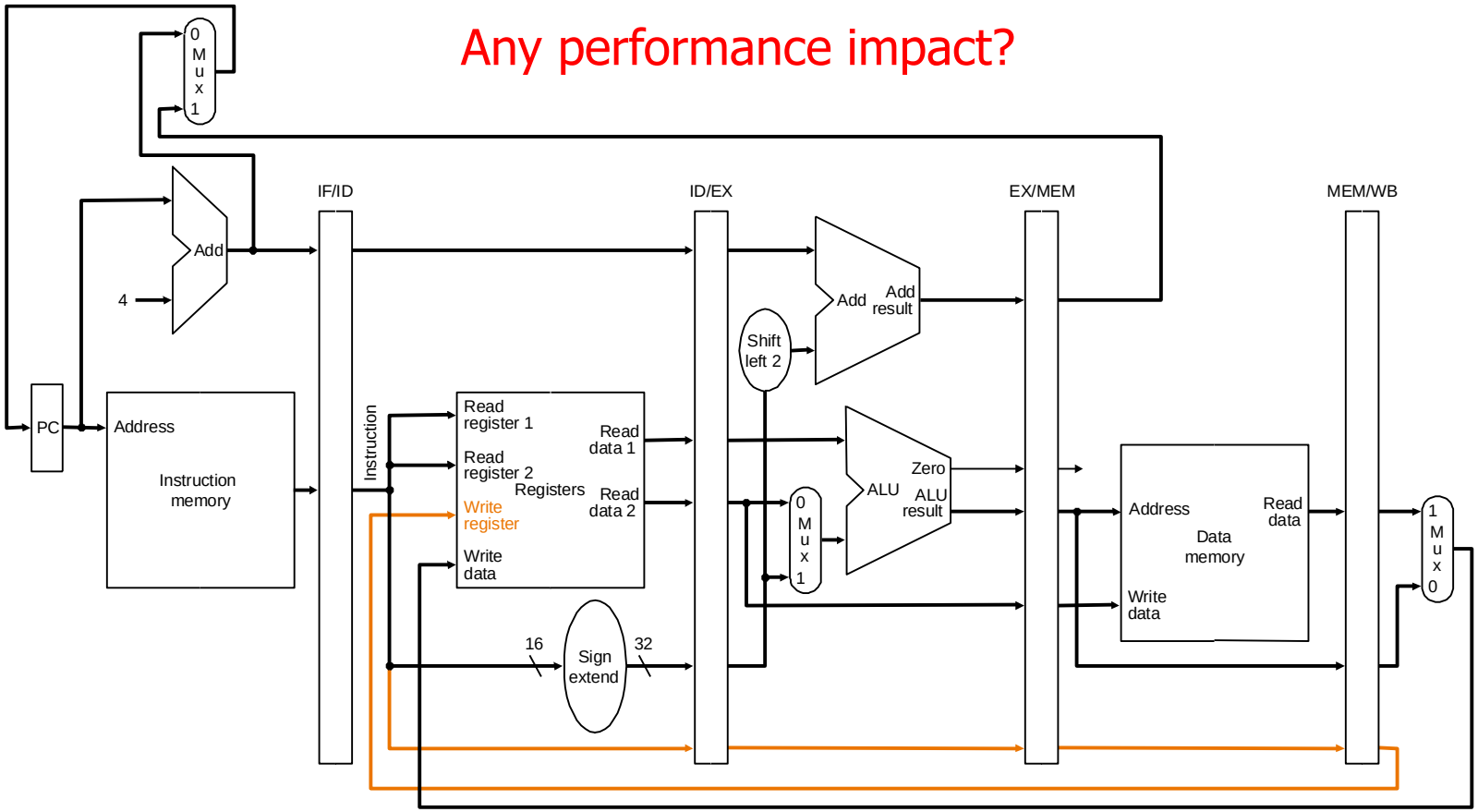




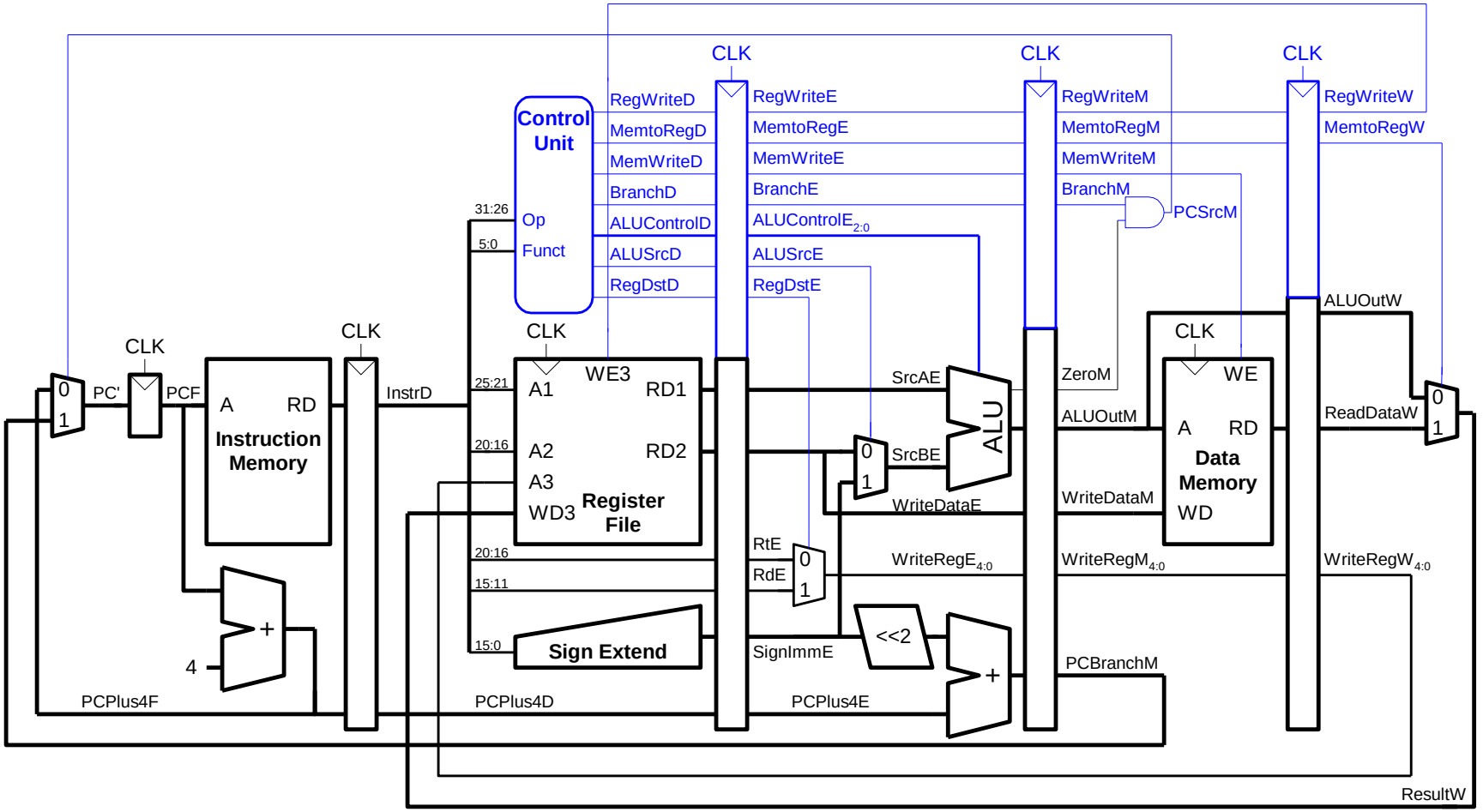
Pipelined Operation Example

All instruction classes must follow the same path and timing through the pipeline stages.

Any performance impact?



Pipelined Control – An example



- **Same control unit as single-cycle processor**
  Control delayed to proper pipeline stage

Remember: An Ideal Pipeline

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

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!
  - ⇒ different instructions → not all need the same stages
    Forcing different instructions to go through the same pipe stages
    → external fragmentation (some pipe stages idle for some instructions)
- Uniform suboperations ... NOT!
  - ⇒ different pipeline stages → not the same latency
    Need to force each stage to be controlled by the same clock
    → internal fragmentation (some pipe stages are fast but still have to take the same clock cycle time)
- Independent operations ... NOT!
  - ⇒ instructions are not independent of each other
    Need to detect and resolve inter-instruction dependences to ensure the pipeline provides correct results
    → pipeline stalls (pipeline is not always moving)

We Covered Until Here in Lecture

Fundamentals of Computer Arch.

Lecture 1: Modern Microprocessor Design

Prof. Onur Mutlu
ETH Zürich

Spring 2025

19 February 2025

Further Slides for Your Own Study (May Be Covered in Future Lectures)

Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing *stalls*

Causes of Pipeline *Stalls*

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

Dependences and Their Types

- Also called "dependency" or *less desirably* "hazard"
- Dependences dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately

Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?

Data Dependences

- Data dependence types
  - Flow dependence (true data dependence – read after write)
  - Anti dependence (write after read)
  - Output dependence (write after write)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2

$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR)

Output dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

Data Dependence Handling

Readings for Next Few Lectures

- H&H, Chapter 7.5-7.9

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination only in last stage and in program order
- Flow dependences are more interesting & challenging
- Six fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Detect and move it out of the way for independent instructions
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

Recall: Data Dependence Types

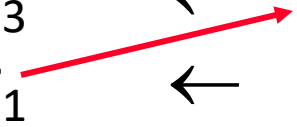
Flow dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

Anti dependence



$r_3 \leftarrow r_1$ op r_2

$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR)

Output dependence

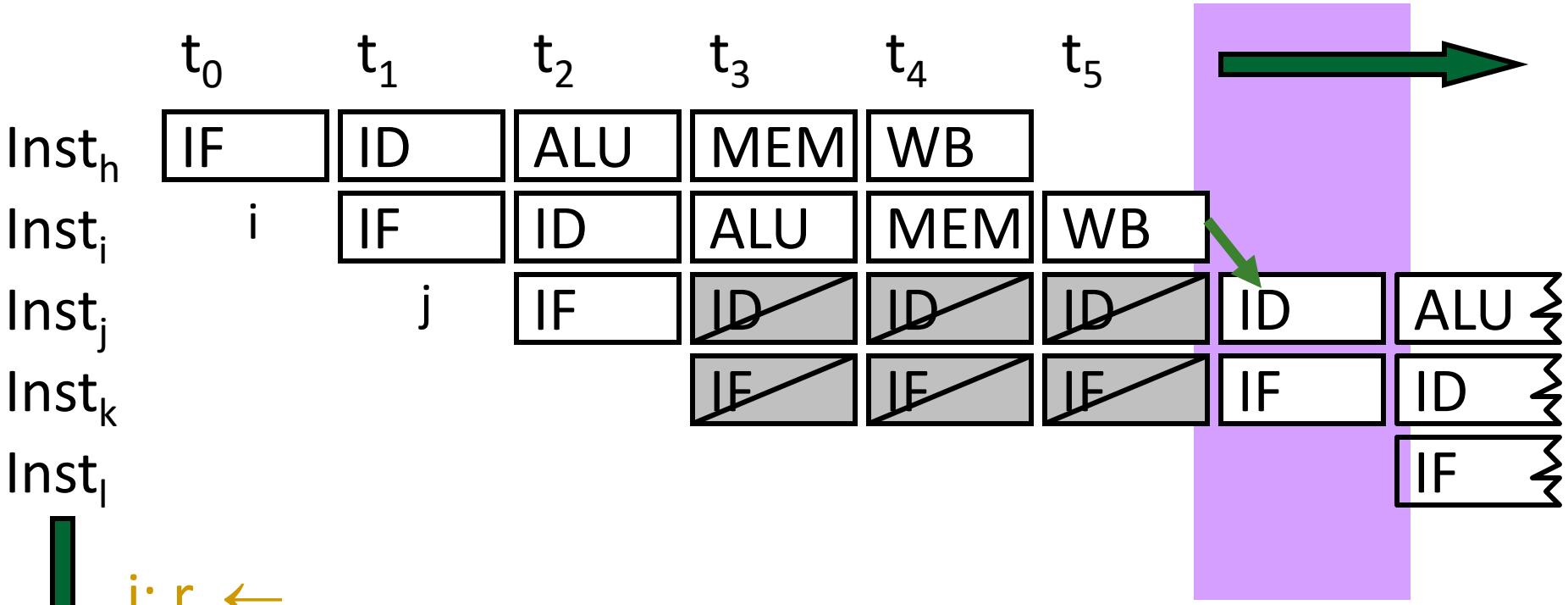
$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

Pipeline Stall: Resolving Data Dependence



Stall = make the dependent instruction **wait** until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages

Interlocking

- Interlocking: Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software based interlocking

  vs.

- Hardware based interlocking

- MIPS acronym?

Approach to Data Dependence Detection

- Combinational dependence check logic
  - Special logic checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall… no flow dependence

- Advantage:
  - No need to stall on anti and output dependences

- Disadvantage:
  - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

Once You Detect the Dependence in Hardware

- What do you do afterwards?

- Observation: Dependence between two instructions is detected before the communicated data value becomes available

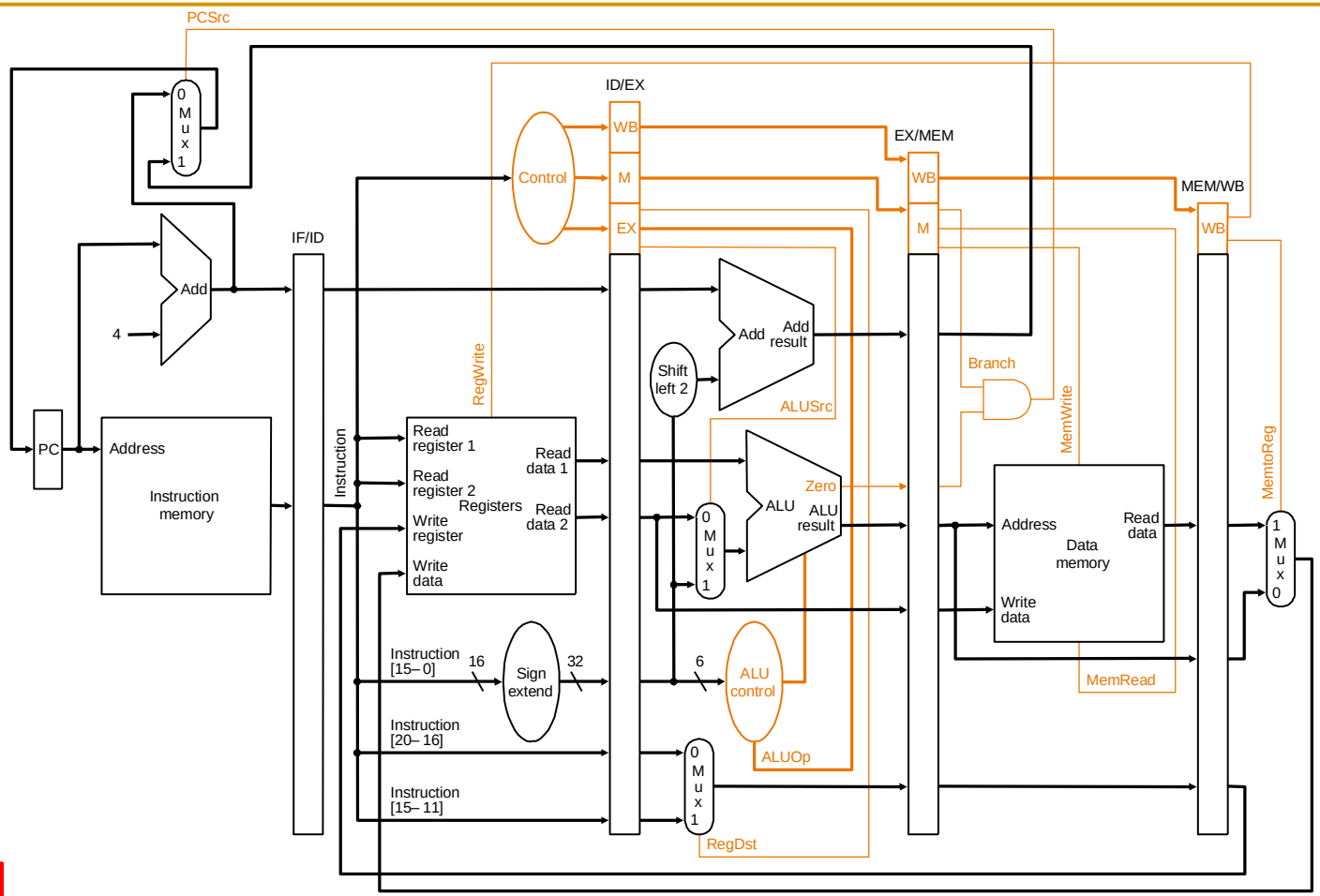
- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: …

Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → **less stalling**

Data Dependence Handling: Concepts and Implementation

How to Implement Stalling



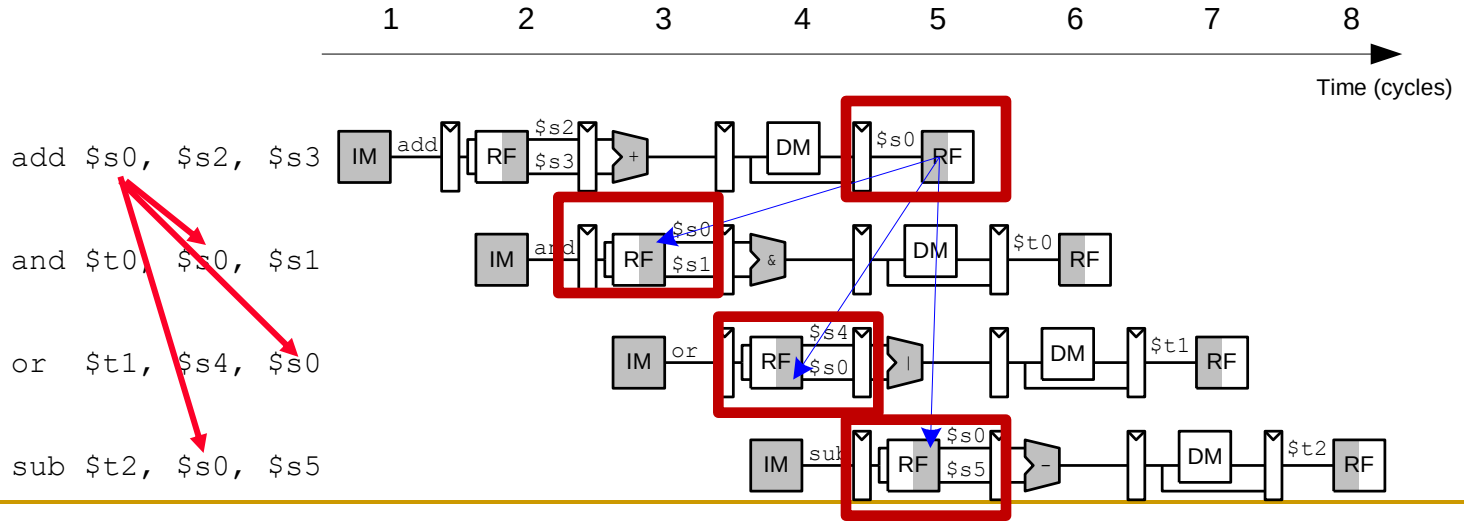
- **Stall**
  - disable **PC** and **IF/ID** latching; ensure stalled instruction stays in its stage
  - Insert **"invalid"** instructions/nops into the stage following the stalled one (called "**bubbles**")

RAW Data Dependence Example

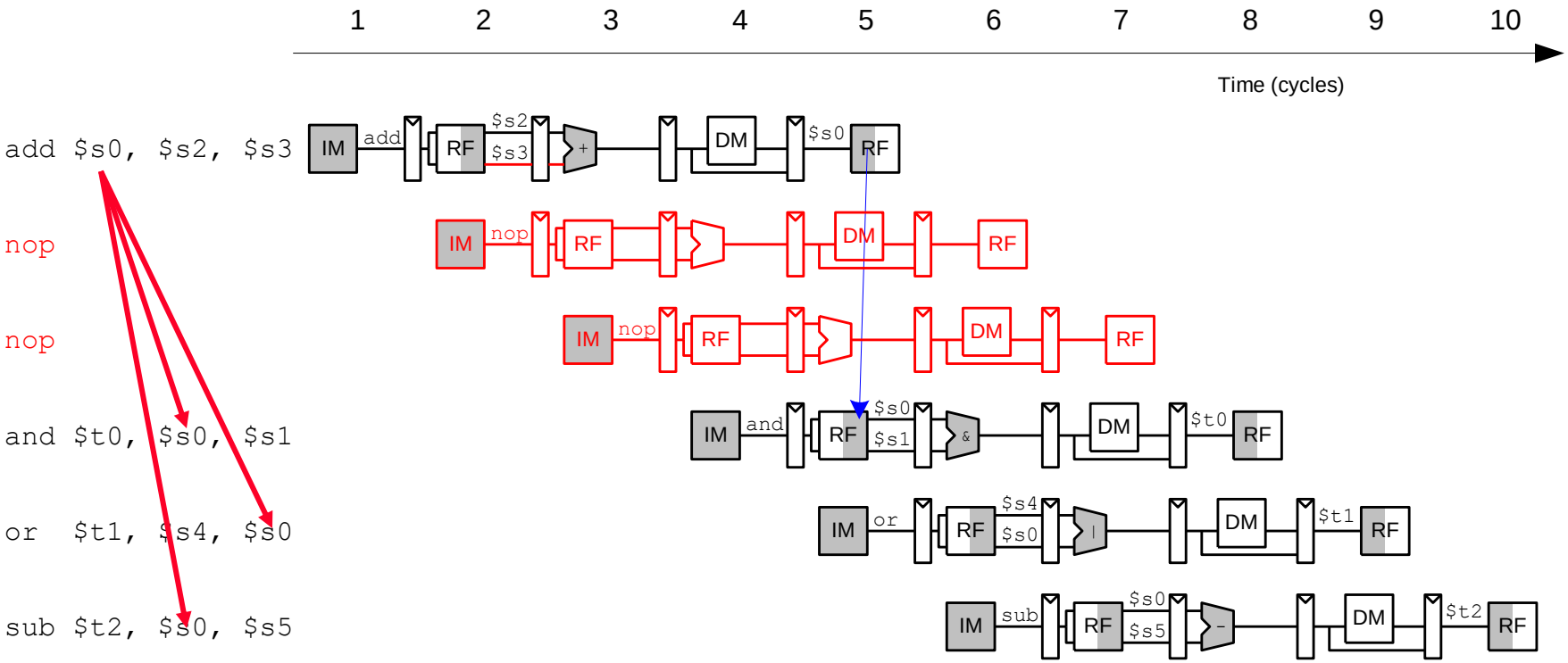
- One instruction writes a register ($s0) and next instructions read this register => read after write (RAW) dependence.

Wrong results happen only if the pipeline handles data dependences incorrectly!

  - subsequent instructions read the correct value of $s0



Compile-Time Detection and Elimination

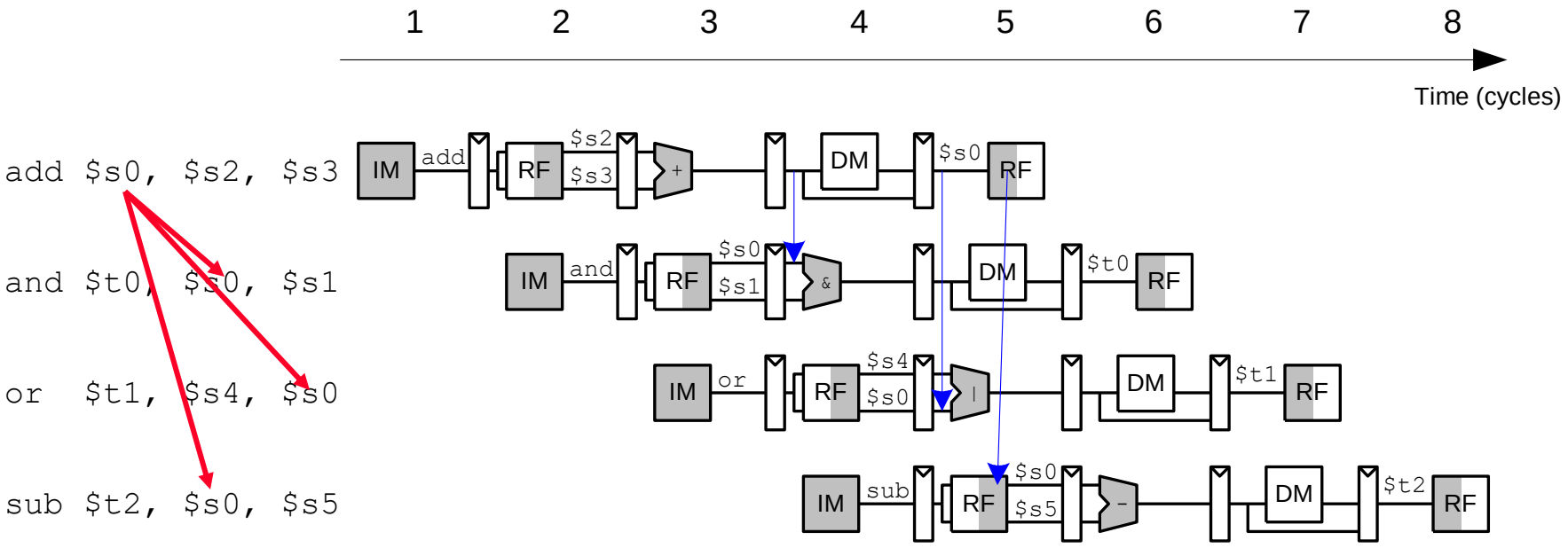


- Insert enough independent instructions for the required result to be ready by the time it is needed by a dependent one
  - Reorder/reschedule/insert instructions at the compiler level

Data Forwarding

- Also called Data Bypassing
- Forward the result value to the dependent instruction as soon as the value is available
- Basic idea comes from "dataflow architectures"
  - Data value is supplied to dependent instruction as soon as it is available
  - Instruction executes when all its operands are available
- Data forwarding brings a pipeline closer to data flow execution principles

Data Forwarding: Locations in Datapath



From latched output of ALU to input of ALU

From WB to input of ALU

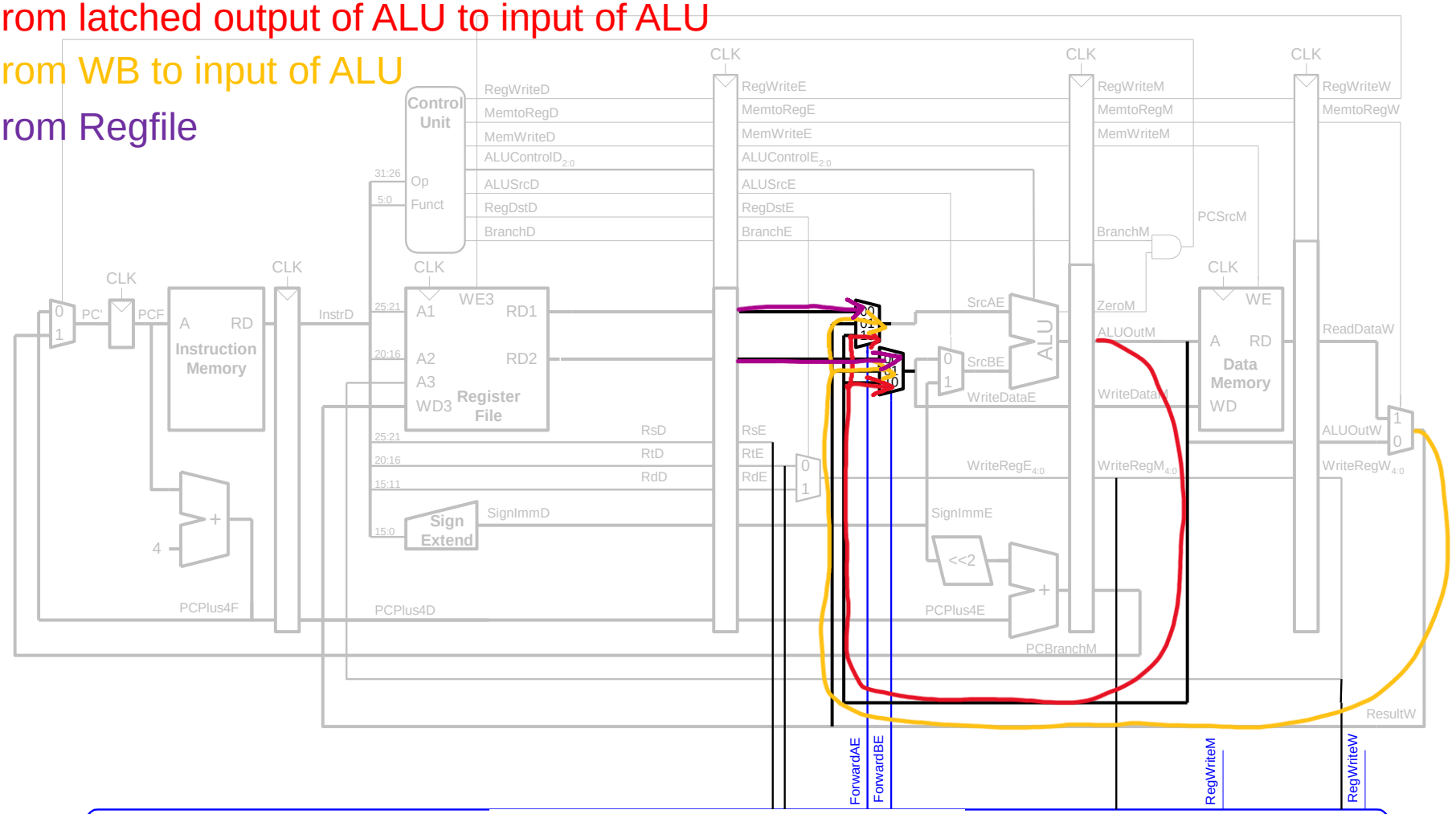
From WB to RF (internal in Register File)

Data Forwarding: Datapath & Control

From latched output of ALU to input of ALU

From WB to input of ALU

From Regfile



Dependence Detection Logic

Data Forwarding: Implementation

- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage

- When should we forward from either Memory or Writeback stage?
  - If that stage will write to a destination register and the destination register matches the source register
  - If both the Memory & Writeback stages contain matching destination registers, Memory stage has priority to forward its data, because it contains the *more recently executed* instruction

Data Forwarding (in Pseudocode)

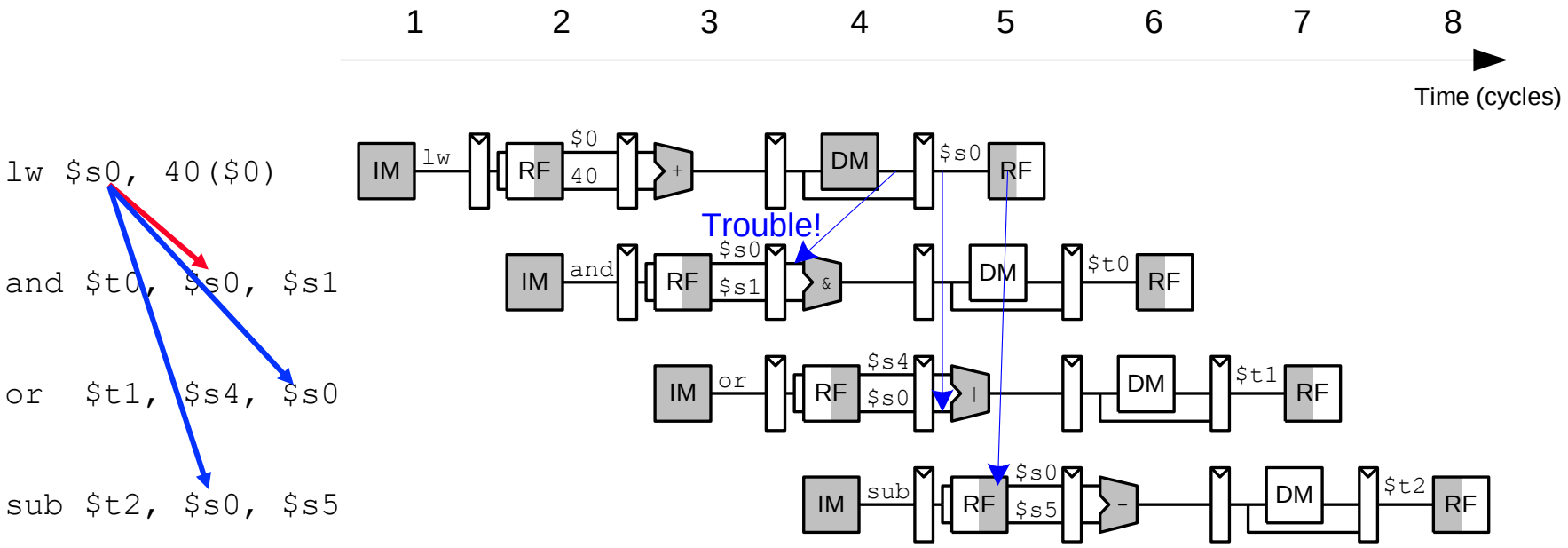
- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage

- Forwarding logic for *ForwardAE (pseudo code)*:

```
if  ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
      ForwardAE = 10  # forward from Memory stage
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
      ForwardAE = 01  # forward from Writeback stage
else
      ForwardAE = 00  # no forwarding
```

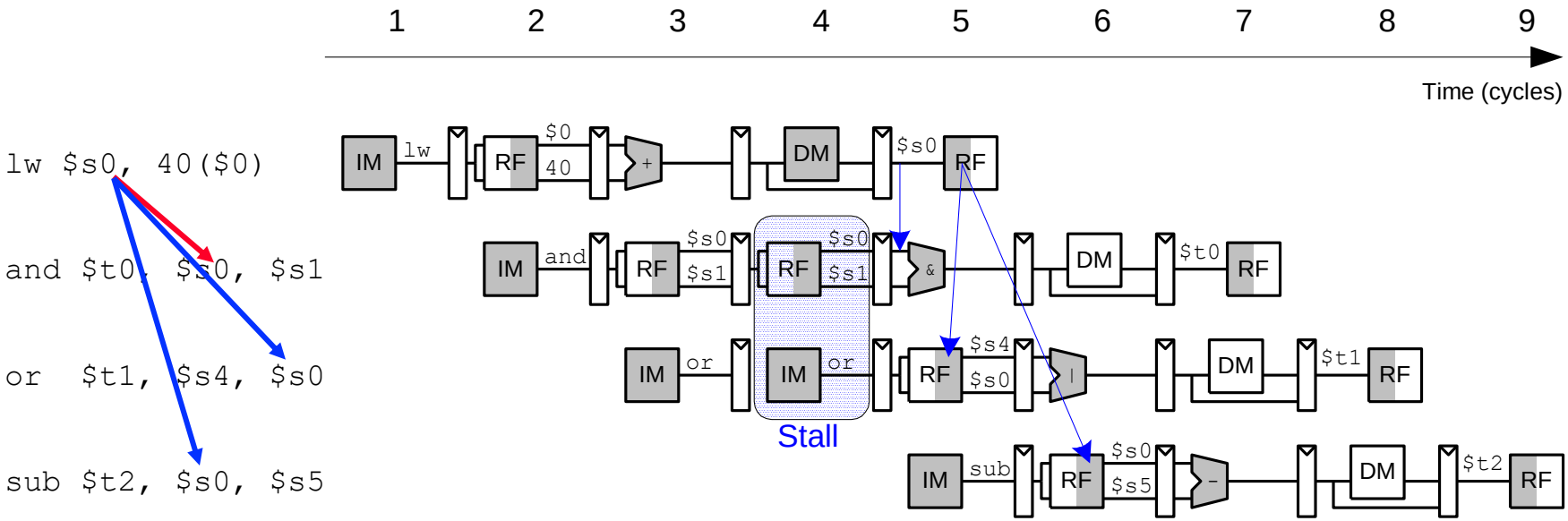
- Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

Data Forwarding Is **Not** Always Possible

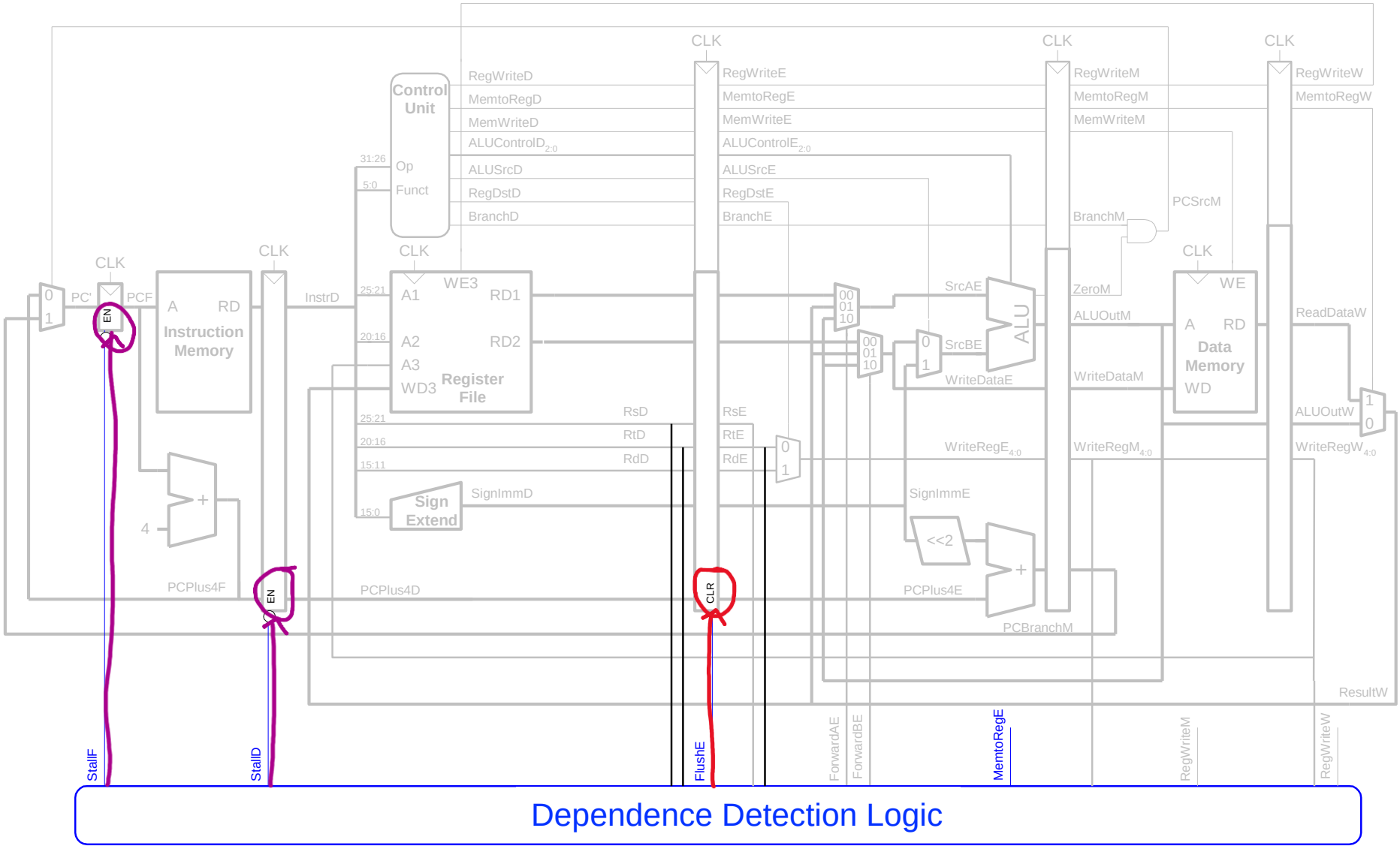


- Forwarding is usually sufficient to resolve RAW data dependences
- Unfortunately, there are cases when forwarding is **not possible**
  - due to pipeline design and instruction latencies
  - The `lw` instruction does not finish reading data until the end of Memory stage
    → its result cannot be forwarded to the Execute stage of the next instruction
    unless we want a long critical path → breaks critical path design principle

Stalling Necessary for MEM-EX Dependence



Stalling and Dependence Detection Hardware



Hardware Needed for Stalling

- Stalls are supported by adding
  - enable inputs (EN) to the Fetch and Decode pipeline registers
  - synchronous reset/clear (CLR) input to the Execute pipeline register
    - or an INV bit associated with each pipeline register, indicating that contents are INValid

- When a lw stall occurs
  - Keep the values in the Decode and Fetch stage pipeline registers
    - StallD and StallF are asserted
  - Clear the contents of the Execute stage register, introducing a bubble
    - FlushE is also asserted

A Special Case of Data Dependence

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

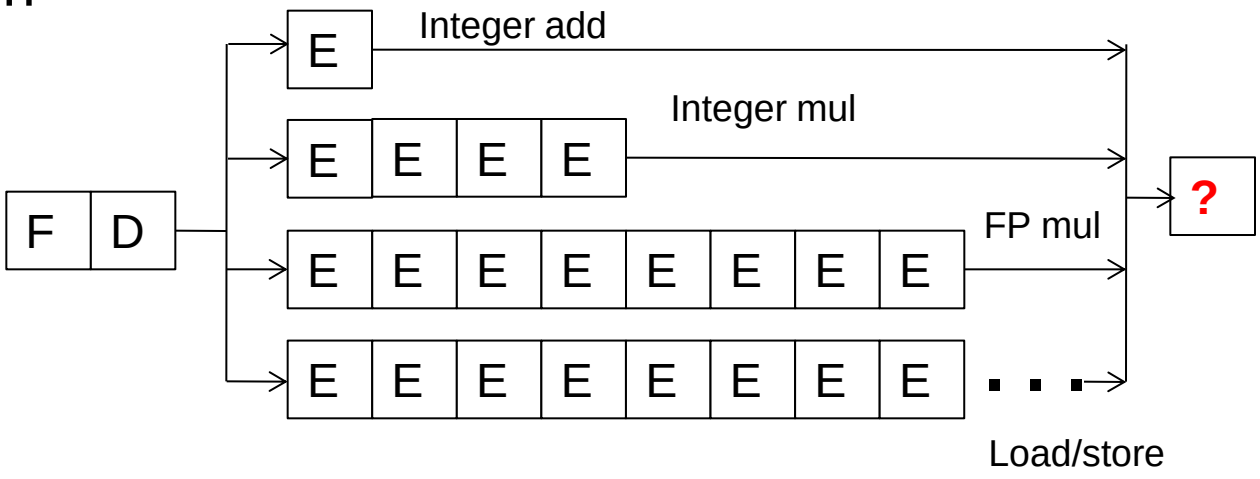
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?

Pipelining and Precise Exceptions: Preserving Sequential Semantics

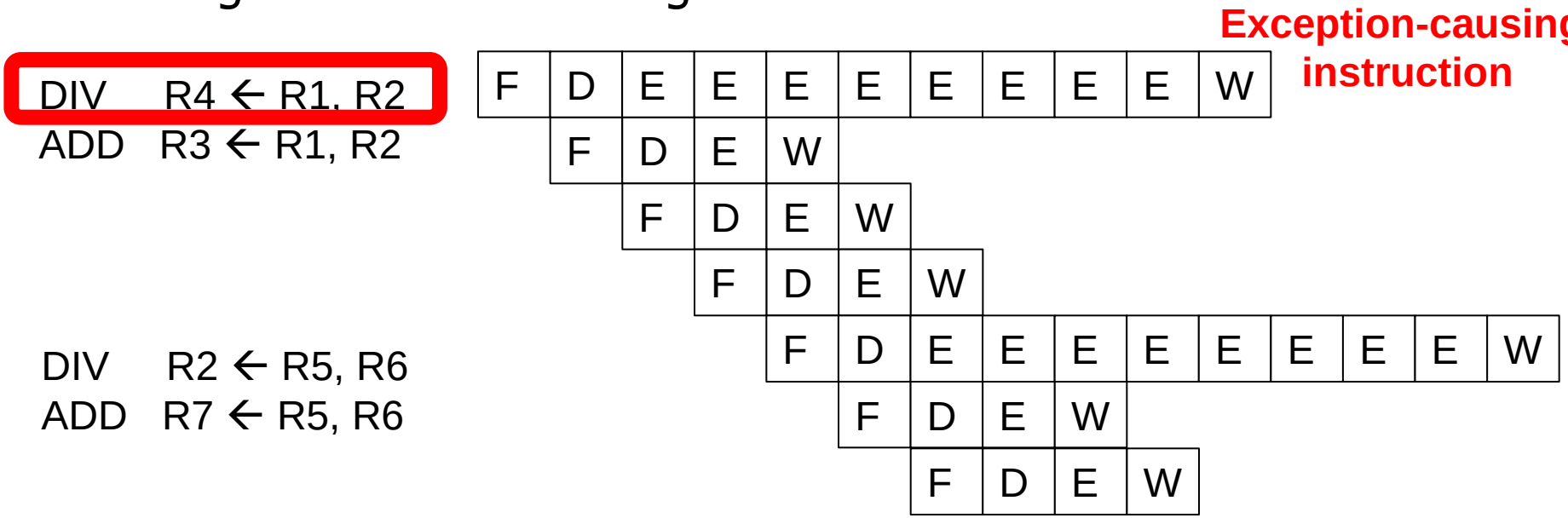
Multi-Cycle Execution

- Not all instructions take the same amount of time in the "execute stage" of the pipeline
- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution



Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus Integer DIVide

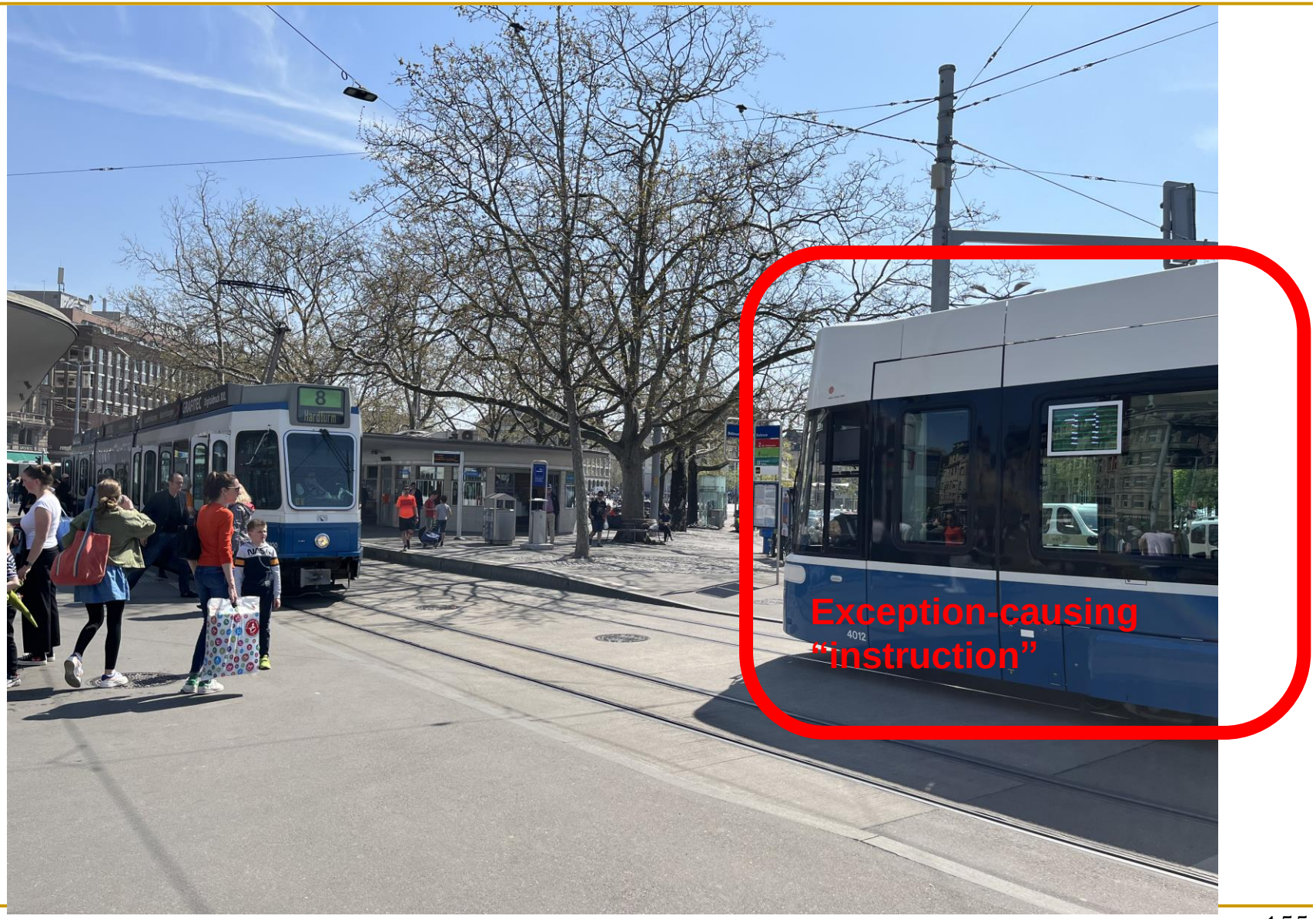


  - What is wrong with this picture in a Von Neumann architecture?
    - Sequential semantics of the ISA NOT preserved!
    - What if DIV incurs an exception? (e.g., DIV by zero)

An Example Exception

Time: 12:55

An Example Exception



Time: 12:57

An Example Exception

Time: 12:58

An Example Exception

Time: 13:00

Another View

Exception Handled & Resolved…

Time: 13:06

Exceptions and Interrupts

- "Unplanned" changes or interruptions in program execution

- Due to **internal** problems in execution of the program
  → Exceptions

- Due to **external** events that need to be handled by the processor
  → Interrupts

- Both exceptions and interrupts require
  - stopping of the current program
  - saving the architectural state
  - handling the exception/interrupt → switch to handler
  - (if possible and makes sense) returning back to program execution

Exceptions and Interrupts: Examples

- Exception examples
  - Divide by zero
  - Overflow
  - Undefined opcode
  - General protection (or access protection)
  - Page fault
  - …
- Interrupt examples
  - I/O device needing service (e.g., keyboard input, video input)
  - (Periodic) system timer expiration
  - Power failure
  - Machine check
  - …

Exceptions vs. Interrupts

- Cause
  - Exceptions: internal to the running thread
  - Interrupts: external to the running thread
- When to Handle
  - Exceptions: when detected (and known to be non-speculative)
  - Interrupts: when convenient
    - Except for very high priority ones
      - Power failure
      - Machine check (error)
- Priority: process (exception), depends (interrupt)
- Handling Context: process (exception), system (interrupt)

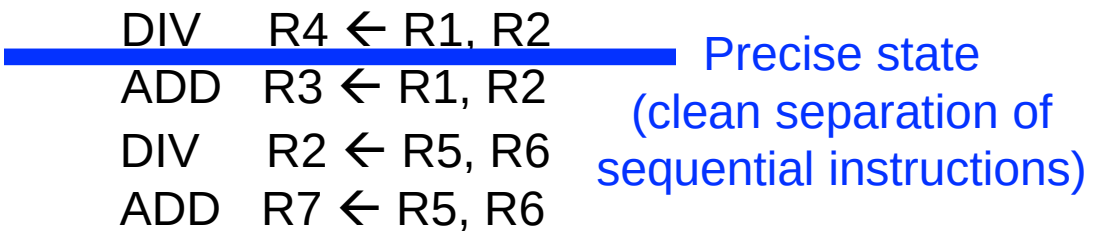
Precise Exceptions/Interrupts

- The architectural state should be consistent (precise) when the exception/interrupt is ready to be handled

1. All previous instructions should be completely retired

2. No later instruction should be retired

Retire = commit = finish execution and update arch. state



Checking for and Handling Exceptions in Pipelining

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic
  - Ensures architectural state is precise (register file, PC, memory)
  - Flushes all younger instructions in the pipeline
  - Saves PC and registers (as specified by the ISA)
  - Redirects the fetch engine to the appropriate exception handling routine

Aside: From the x86-64 ISA Manual

6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT *n* instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor's interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

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

Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. Dataflow
- Aids software debugging
- Enables (easy) recovery from exceptions
- Enables (easily) restartable processes
- Enables traps into software (e.g., software implemented opcodes)

Ensuring Precise Exceptions

- Easy to do in single-cycle and multi-cycle machines

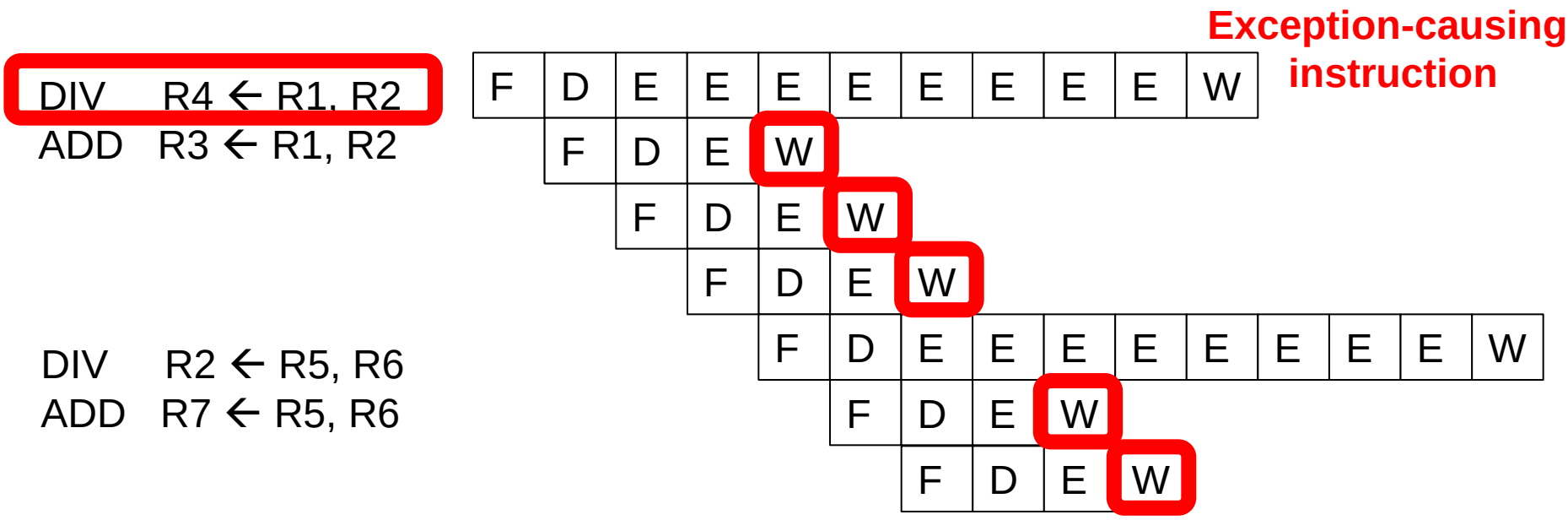
- Single-cycle
  - Instruction boundary == Cycle boundary
  - An instruction is guaranteed to be finished in one cycle
    → no possibility of violating sequential execution semantics

- Multi-cycle
  - Add special states in the control FSM that lead to the exception or interrupt handlers
  - Switch to the handler only at a precise state
    → before fetching the next instruction

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture

Multi-Cycle Execute: More Complications

- Instructions can take different number of cycles in EXECUTE stage → **This complicates exception/interrupt handling**

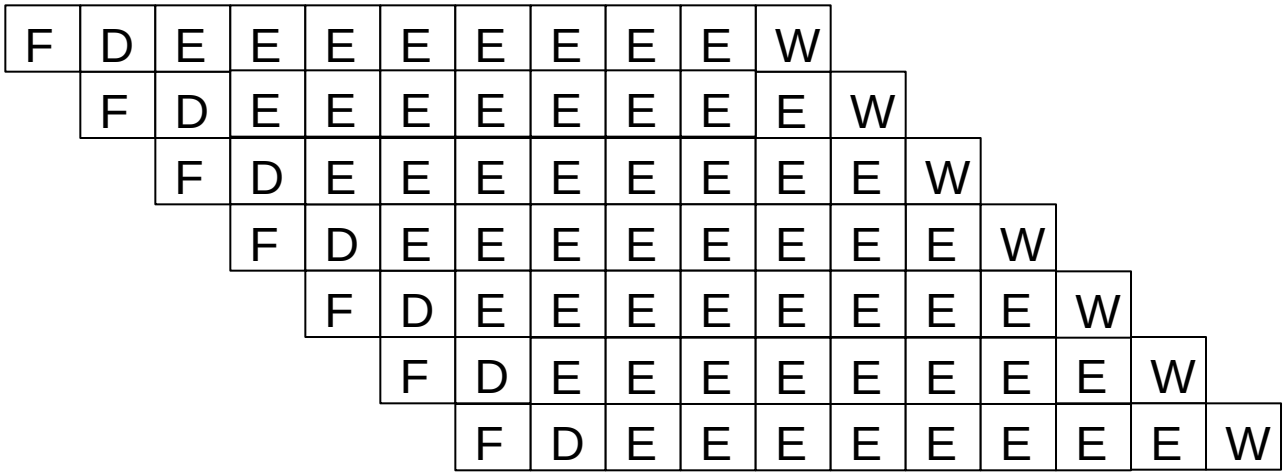


  - What is wrong with this picture in a Von Neumann architecture?
    - Sequential semantics of the ISA NOT preserved!
    - What if DIV incurs an exception? (e.g., DIV by zero)

Ensuring Precise Exceptions in Pipelining

- Idea: Make each operation take the same amount of time

DIV R3 ← R1, R2
ADD R4 ← R1, R2



- Downside
  - Worst-case instruction latency determines all instructions' latency
    - What about memory operations?
    - Each functional unit takes worst-case number of cycles?

Solutions: Supporting Precise Exceptions

- How do we support precise exceptions in the presence of instructions completing out of program order?

- Reorder buffer

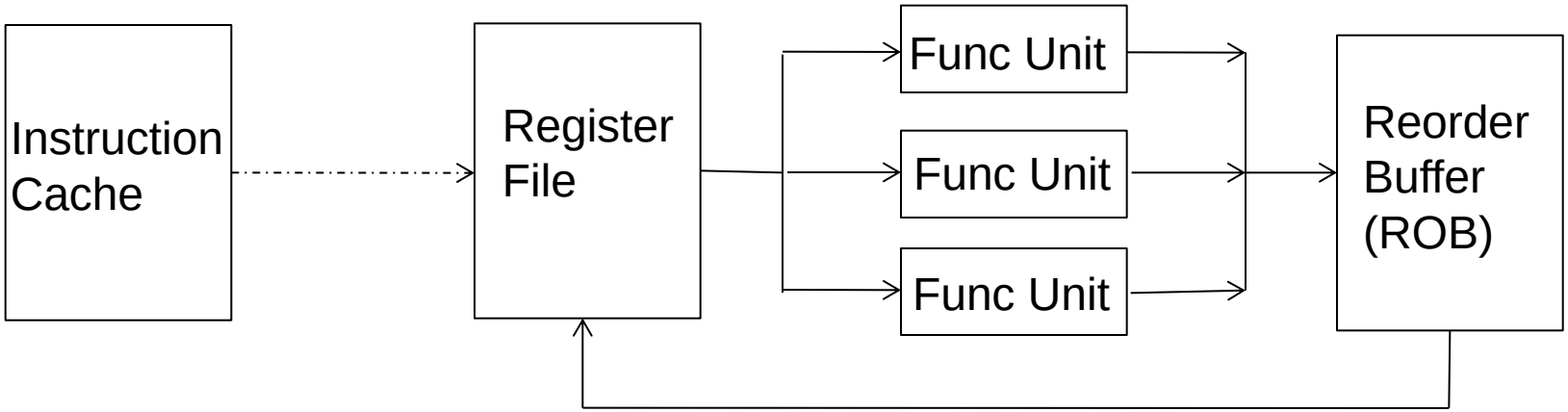
- History buffer
- Future register file
- Checkpointing

We will not cover these
See suggested lecture video from Spring 2015
Also see backup slides

- Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 and ISCA 1985.

Solution I: Reorder Buffer (ROB)

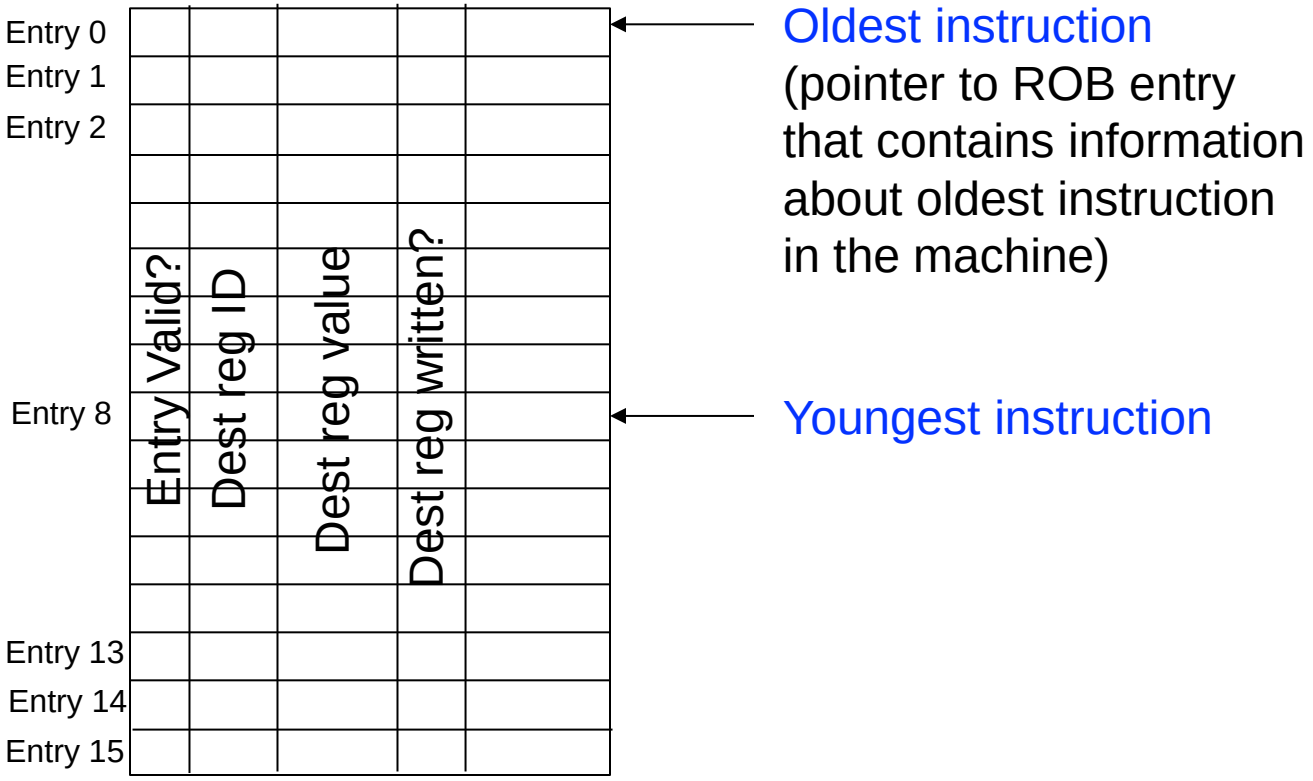
- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is **decoded**, it reserves the next-sequential entry in a special buffer called the Reorder Buffer (ROB)
- When instruction **completes**, it writes result into ROB entry
- When instruction **oldest in ROB** and it has completed without exceptions, its result moved to reg. file or memory



ROB is implemented as a circular queue in hardware

Reorder Buffer

- A hardware structure that keeps information about all instructions that are **decoded** but **not yet retired**/committed



ROB is implemented as a circular queue in hardware

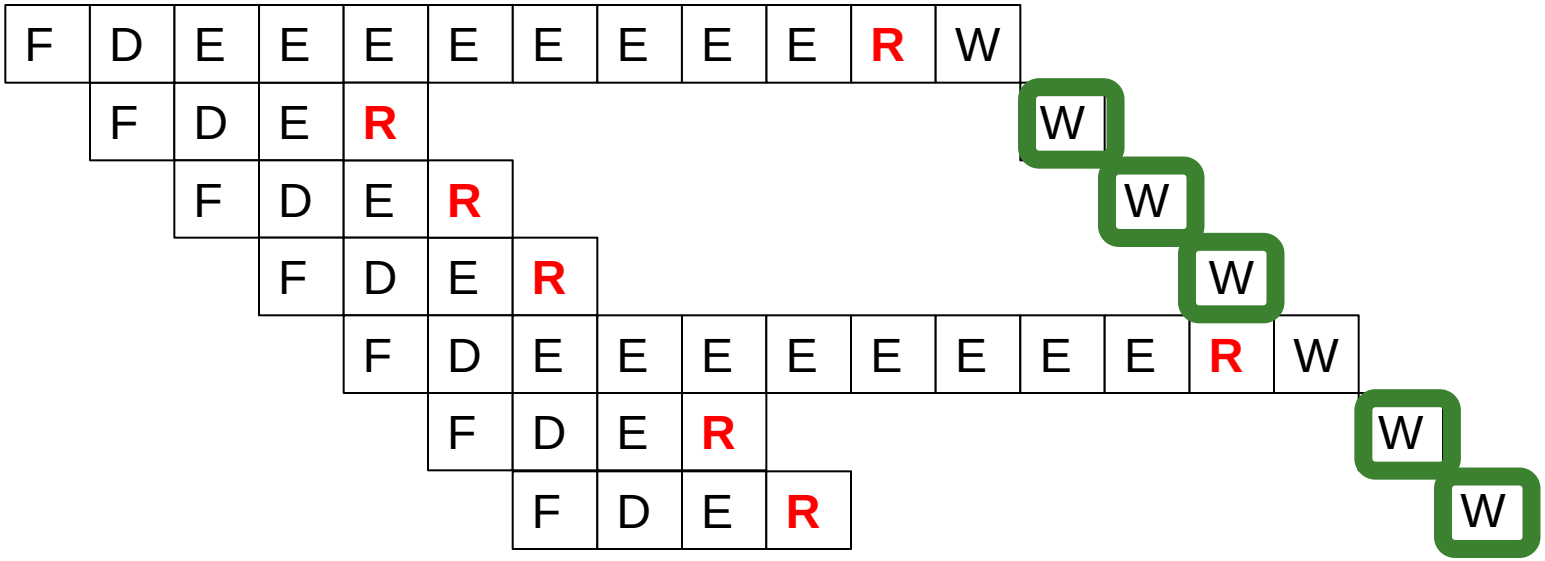
What's in a ROB Entry?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exception?

- Everything required to:
  - correctly reorder instructions back into the program order
  - update the architectural state with the instruction's result(s), if instruction can retire without any issues
  - handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction
- Need valid bits to keep track of readiness of the result(s) and find out if the instruction has completed execution

Reorder Buffer: Independent Operations

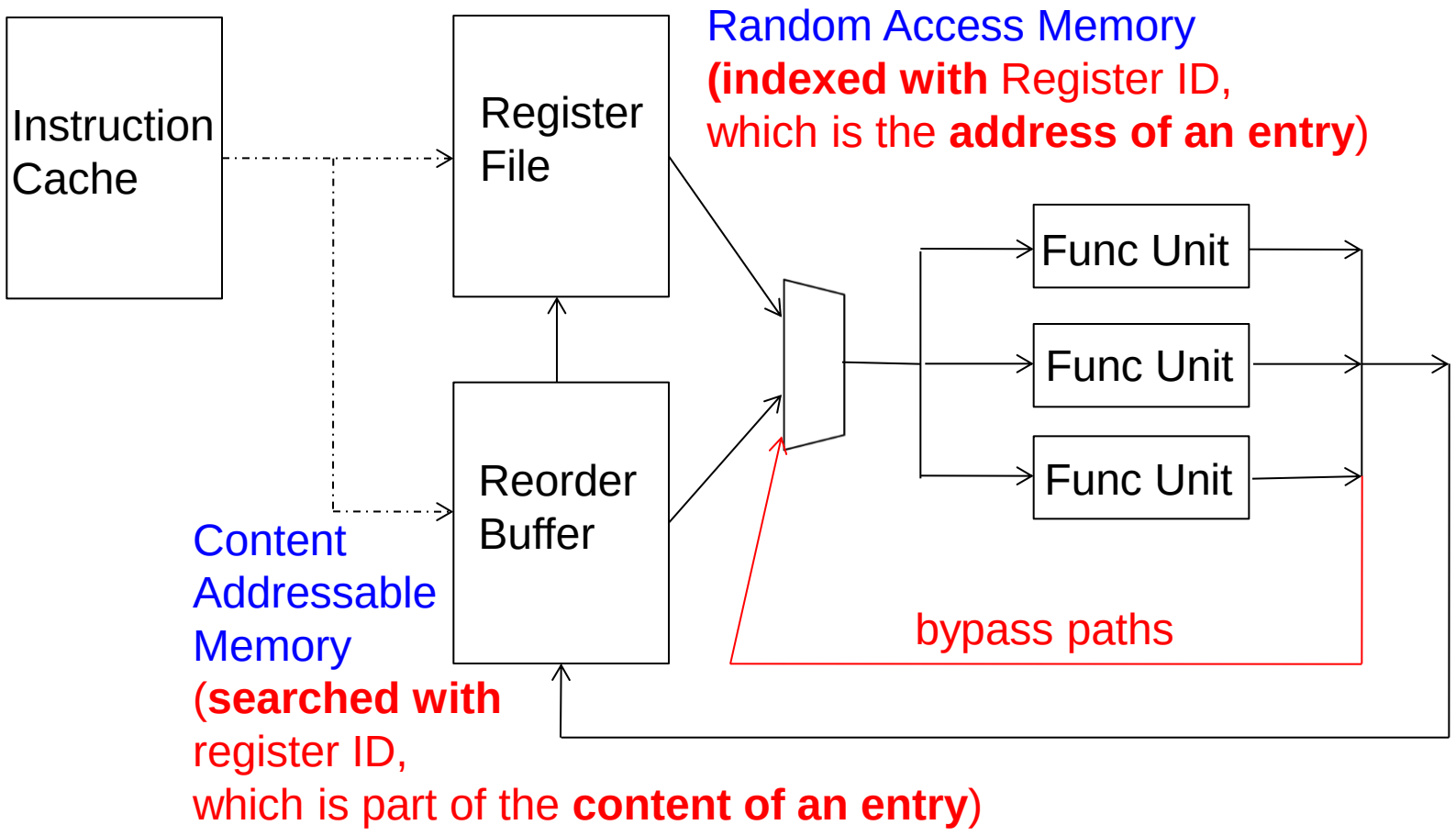
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- What if a later instruction needs a value in the reorder buffer?
  - One option: stall the operation → stall the pipeline
  - Better: Read the value from the reorder buffer. How?

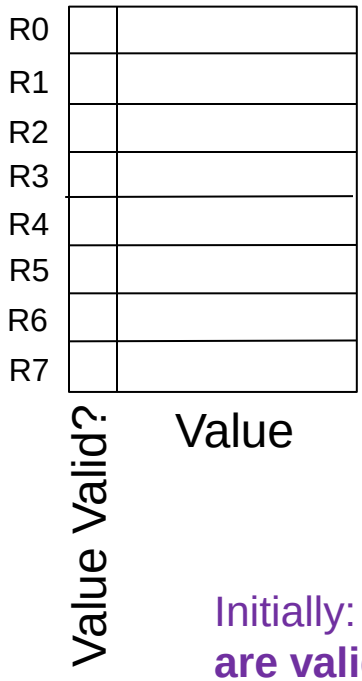
Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)

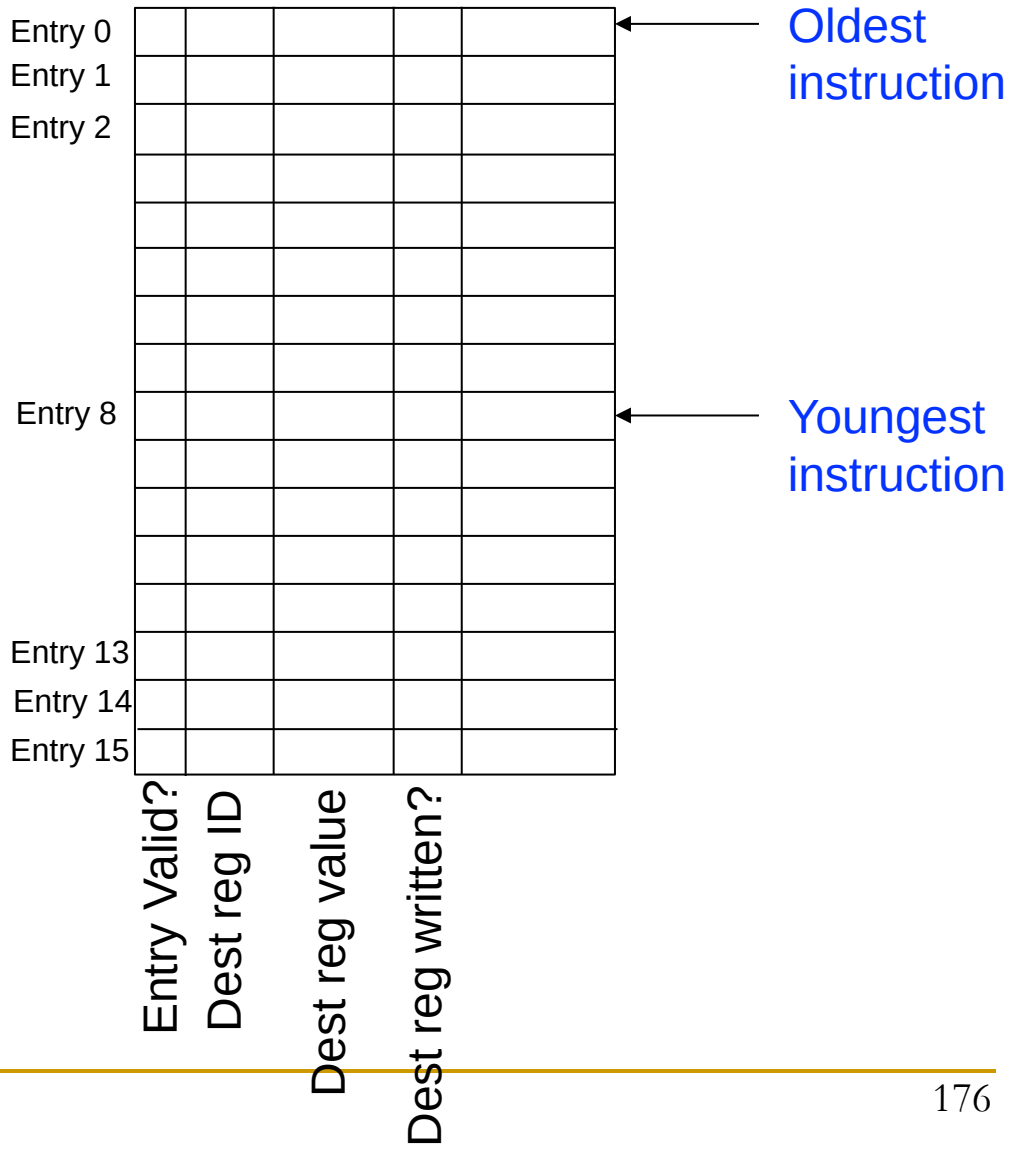


Reorder Buffer Example

Register File (RF)



Reorder Buffer (ROB)



Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Simplifying Reorder Buffer Access

- Idea: Use indirection

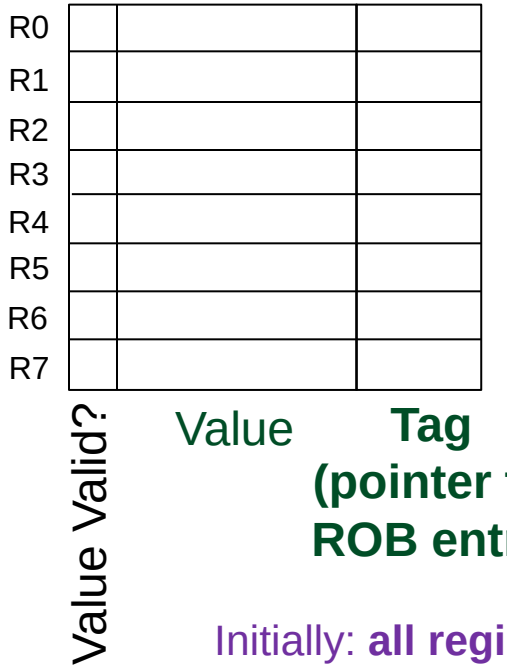
- Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- Access reorder buffer next

- Now, reorder buffer does not need to be content addressable

Reorder Buffer Example

Register File (RF)

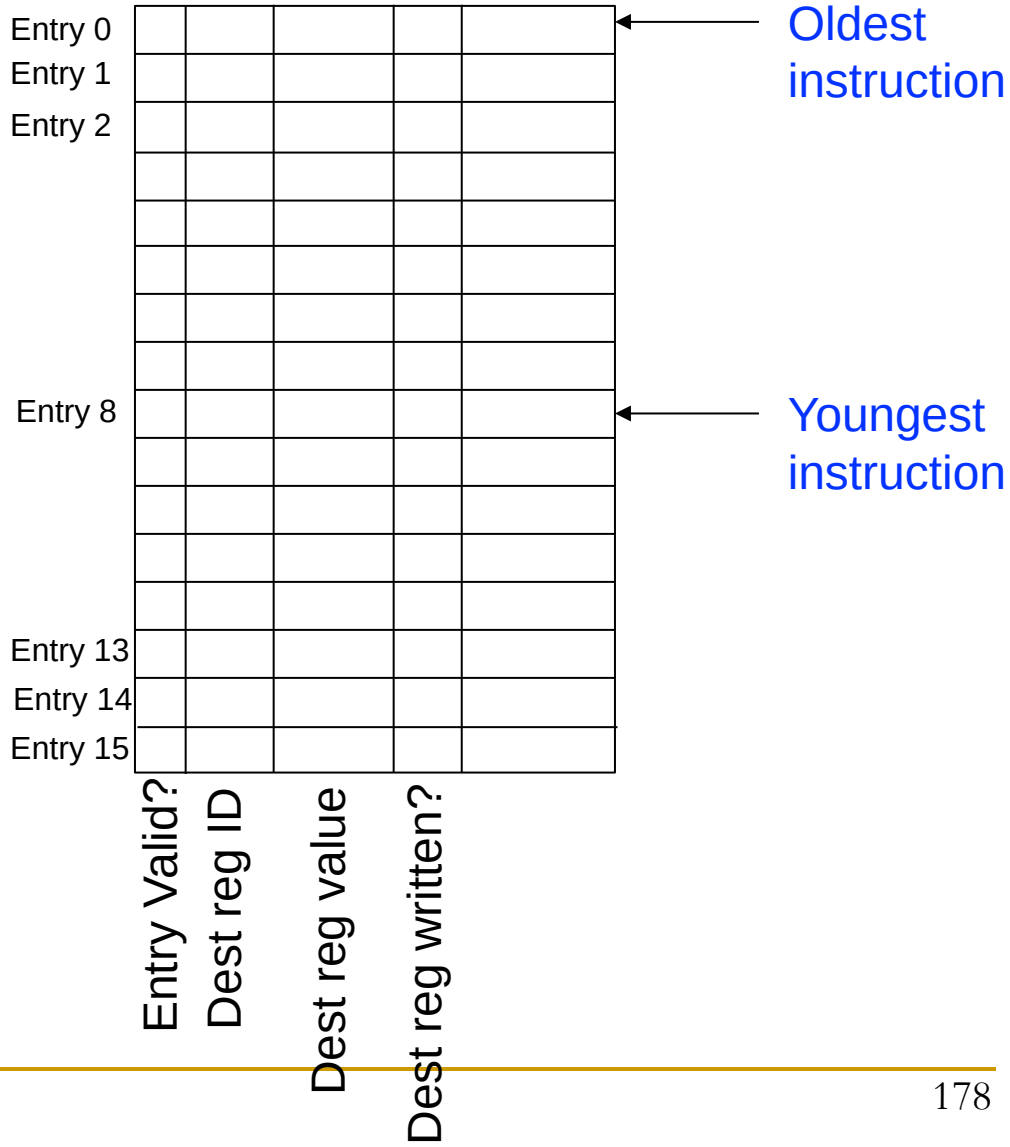


Initially: **all registers are valid in RF & ROB is empty**

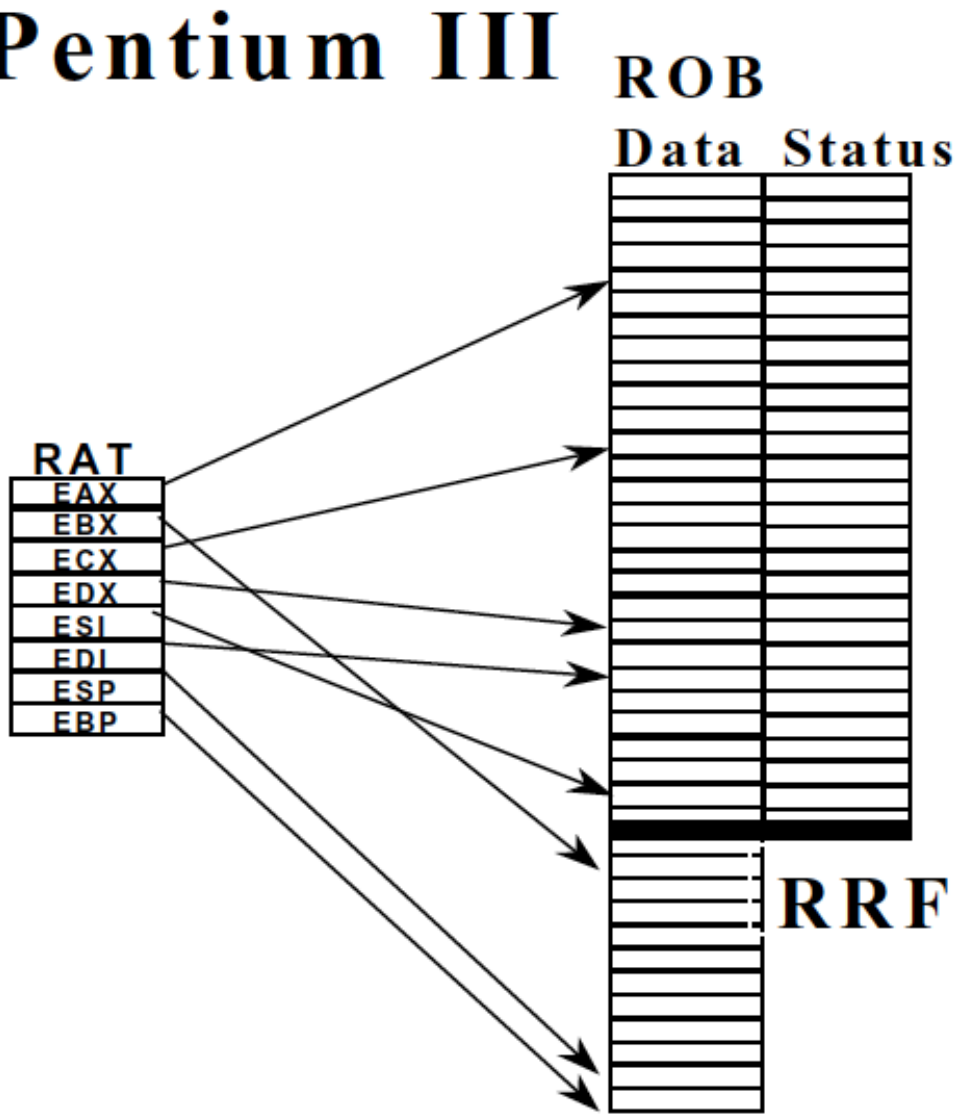
Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer (ROB)



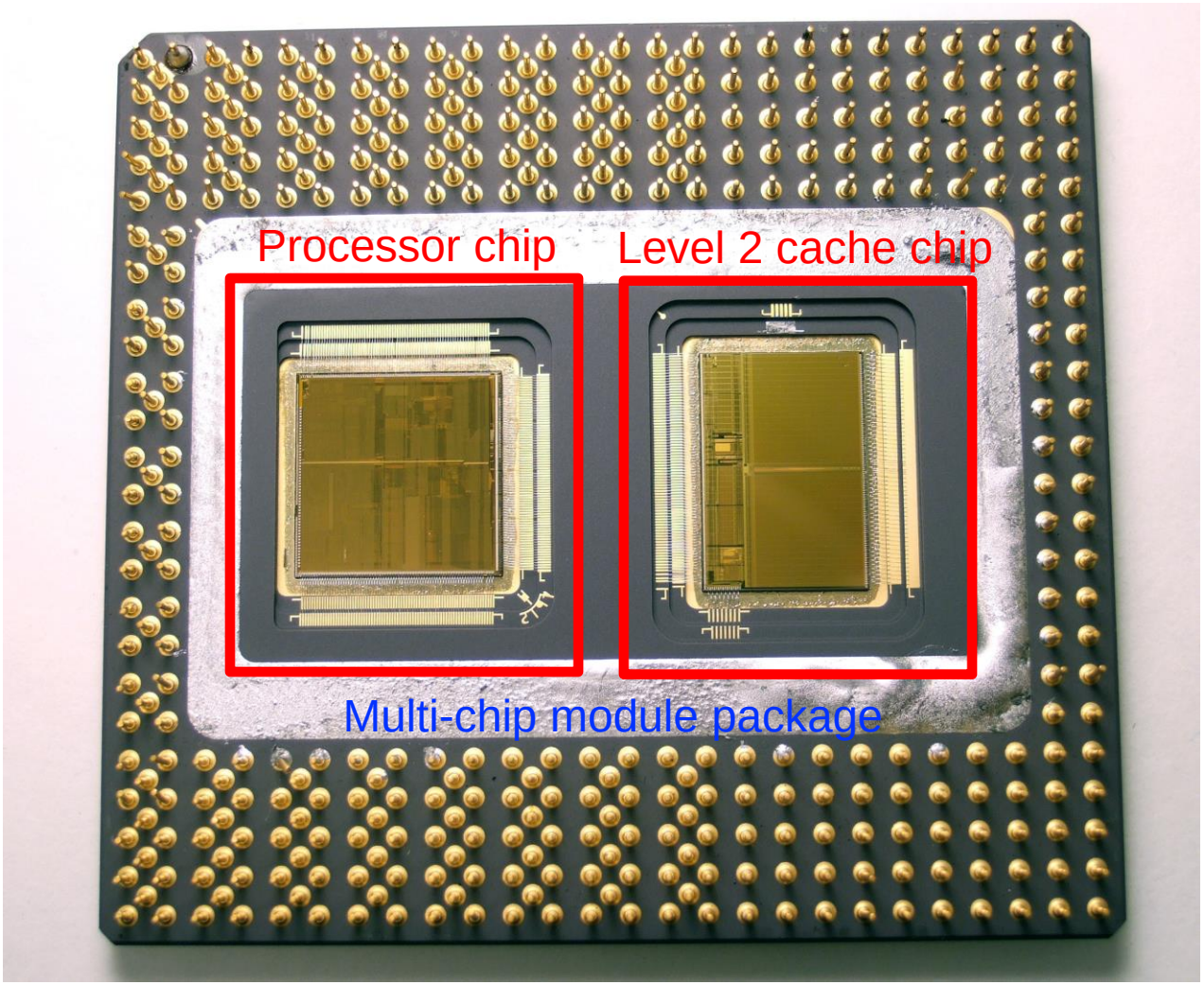
Reorder Buffer in Intel Pentium III/Pro



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

A Register Alias Table (RAT) points to where each register's current value is (or will be)

Intel Pentium Pro (1995)



Important: Register Renaming with a Reorder Buffer

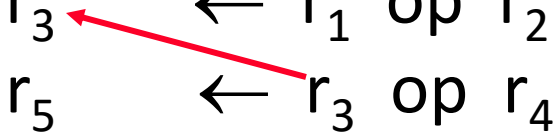
- Output and anti dependences are **not true dependences**
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID's (i.e. names) in the ISA**

- The register ID is **renamed** to the reorder buffer entry that will hold the register's value
  - Register ID → ROB entry ID
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register

- This eliminates anti and output dependences
  - Gives the illusion that there are a large number of registers

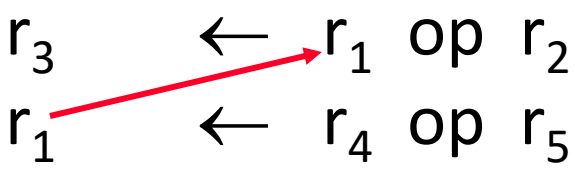
Recall: Data Dependence Types

Flow dependence



Read-after-Write (RAW)

Anti dependence



Write-after-Read (WAR)

Output dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

Register Renaming Example (On Your Own)

- Assume
  - Register file has a pointer to the reorder buffer entry that contains or will contain the value, if the register is not valid
  - Reorder buffer works as described before

- Where is the latest definition of R3 for each instruction below in sequential order?

  LD R0(0) → R3

  LD R3, R1 → R10

  MUL R1, R2 → R3

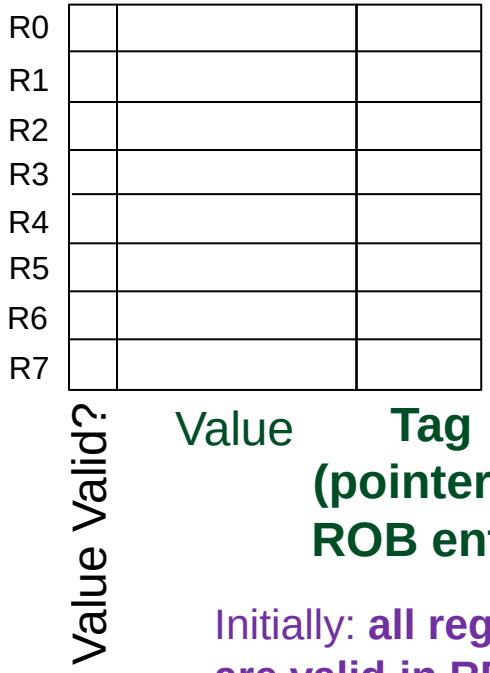
  MUL R3, R4 → R11

  ADD R5, R6 → R3

  ADD R3, R8 → R12

Reorder Buffer Example

Register File (RF)

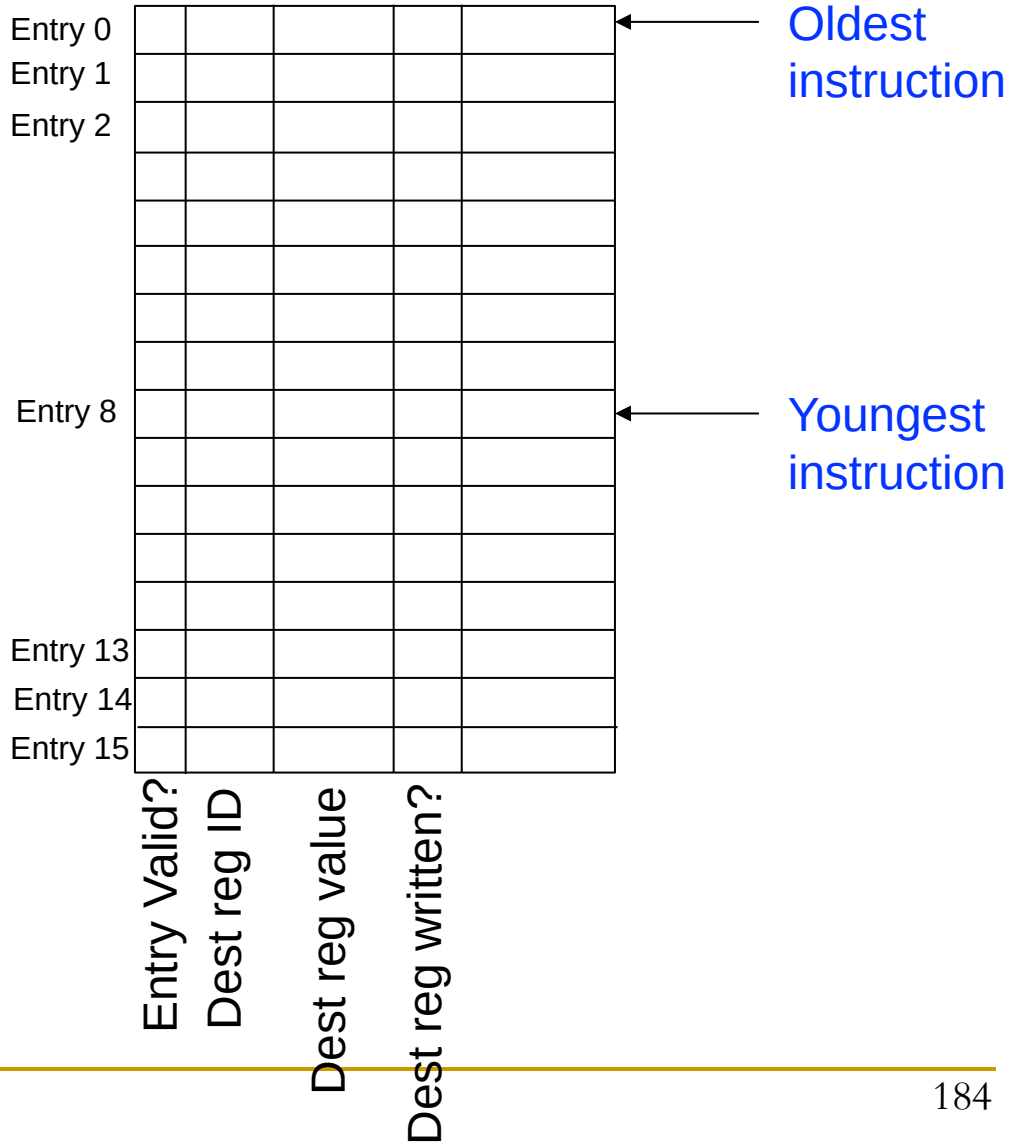


Initially: **all registers are valid in RF & ROB is empty**

Simulate:

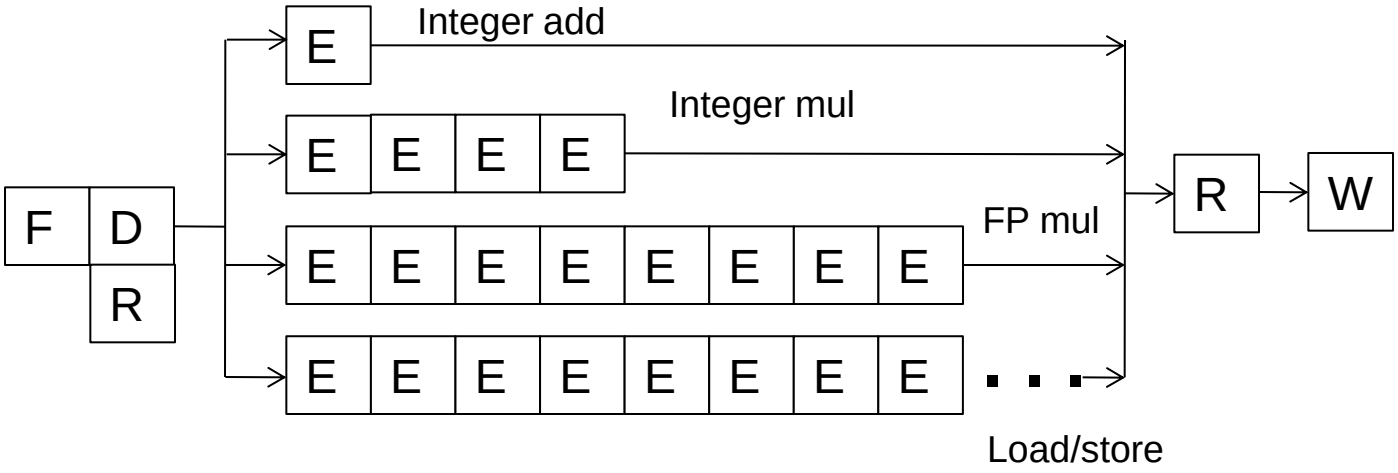
LD R0(0) → R3
LD R3, R1 → R10
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer (ROB)



In-Order Pipeline with Reorder Buffer

- Decode (D): Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to reorder buffer
- Retirement/Commit (W): Check oldest instruction for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- In-order dispatch/execution, out-of-order completion, in-order retirement



ROB is implemented as a circular queue in hardware

Reorder Buffer Tradeoffs

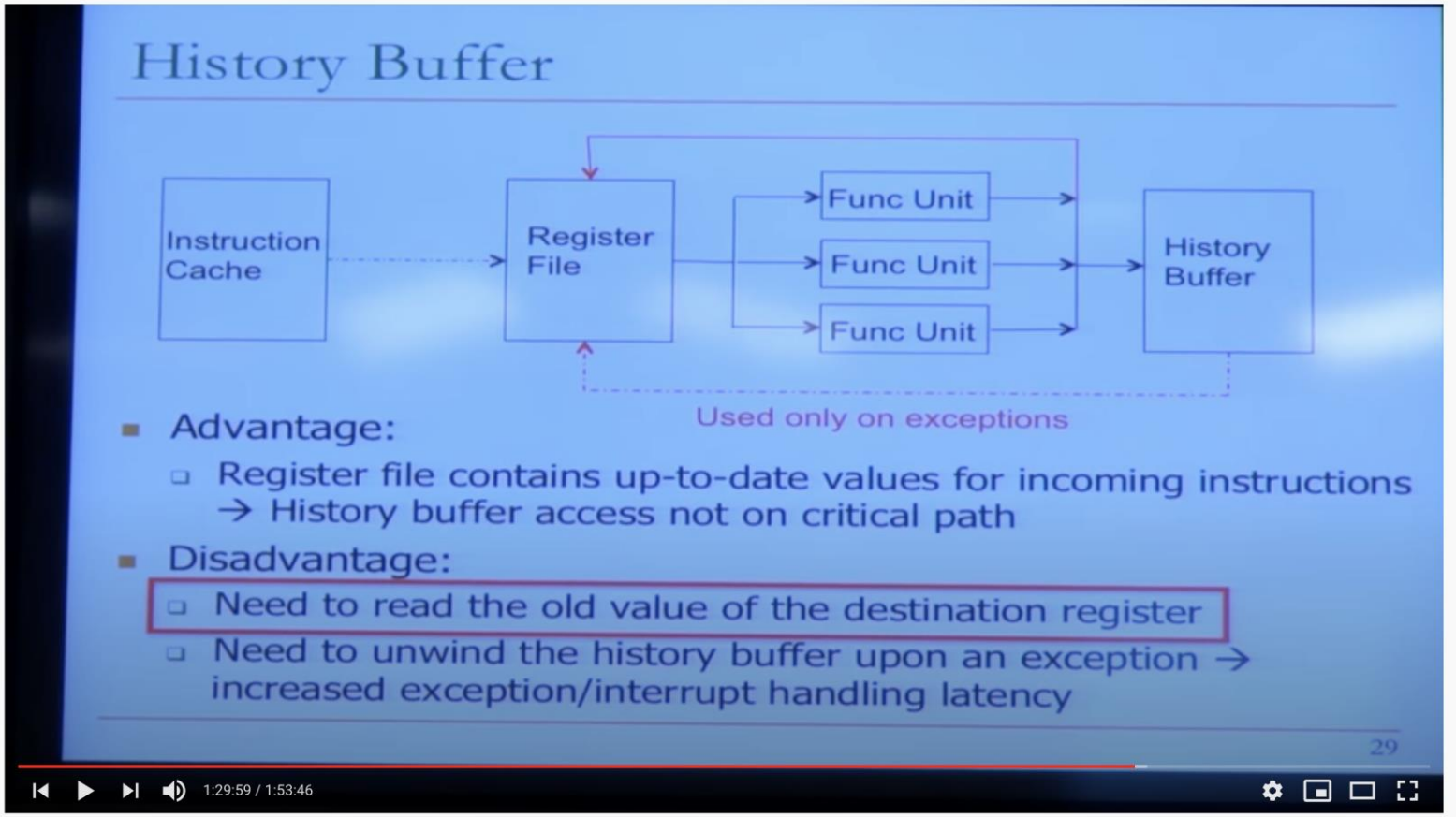
- Advantages
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependences

- Disadvantages
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection → increased latency and complexity

- Other solutions aim to eliminate the disadvantages
  - History buffer
  - Future file
  - Checkpointing

We will not cover these
See suggested lecture video from Spring 2015
Also see backup slides

More on State Maintenance & Precise Exceptions



Lecture 11. Precise Exceptions, State Maintenance/Recovery - CMU - Comp. Arch. 2015 - Onur Mutlu

11,990 views • Feb 12, 2015 77 0 SHARE SAVE

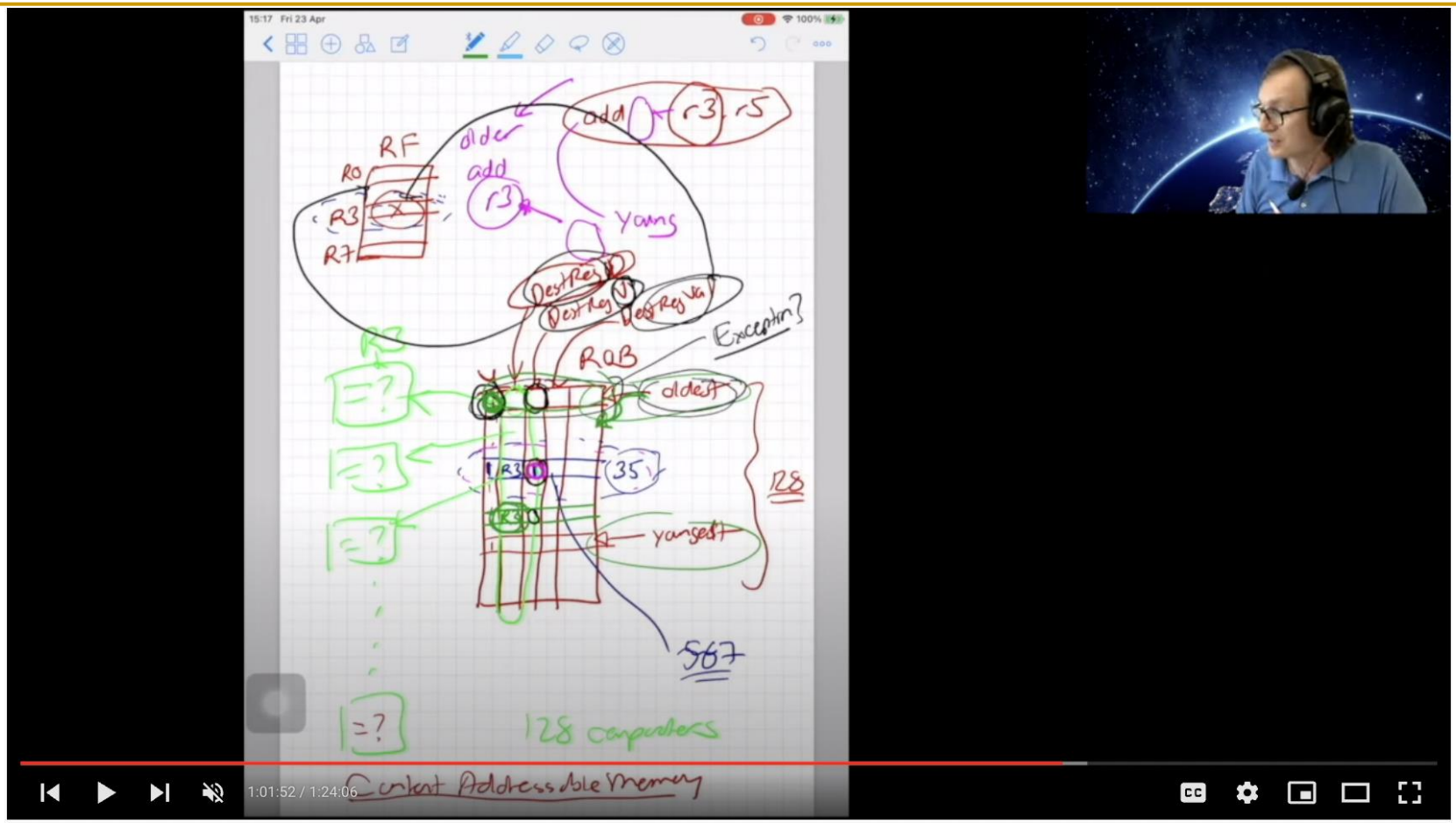
Carnegie Mellon Computer Architecture
23K subscribers

ANALYTICS EDIT VIDEO

Lecture 11. Precise Exceptions, State Maintenance, State Recovery
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: Feb 11th, 2015

https://www.youtube.com/watch?v=nMfbtzWizDA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=13

More on State Maintenance & Precise Exceptions



DEPARTMENT OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING (D-ITET)

Digital Design & Computer Arch. - Lecture 15a: Precise Exceptions (ETH Zürich, Spring 2021)

482 views • Premiered Feb 12, 2022

16 DISLIKE SHARE CLIP SAVE ...

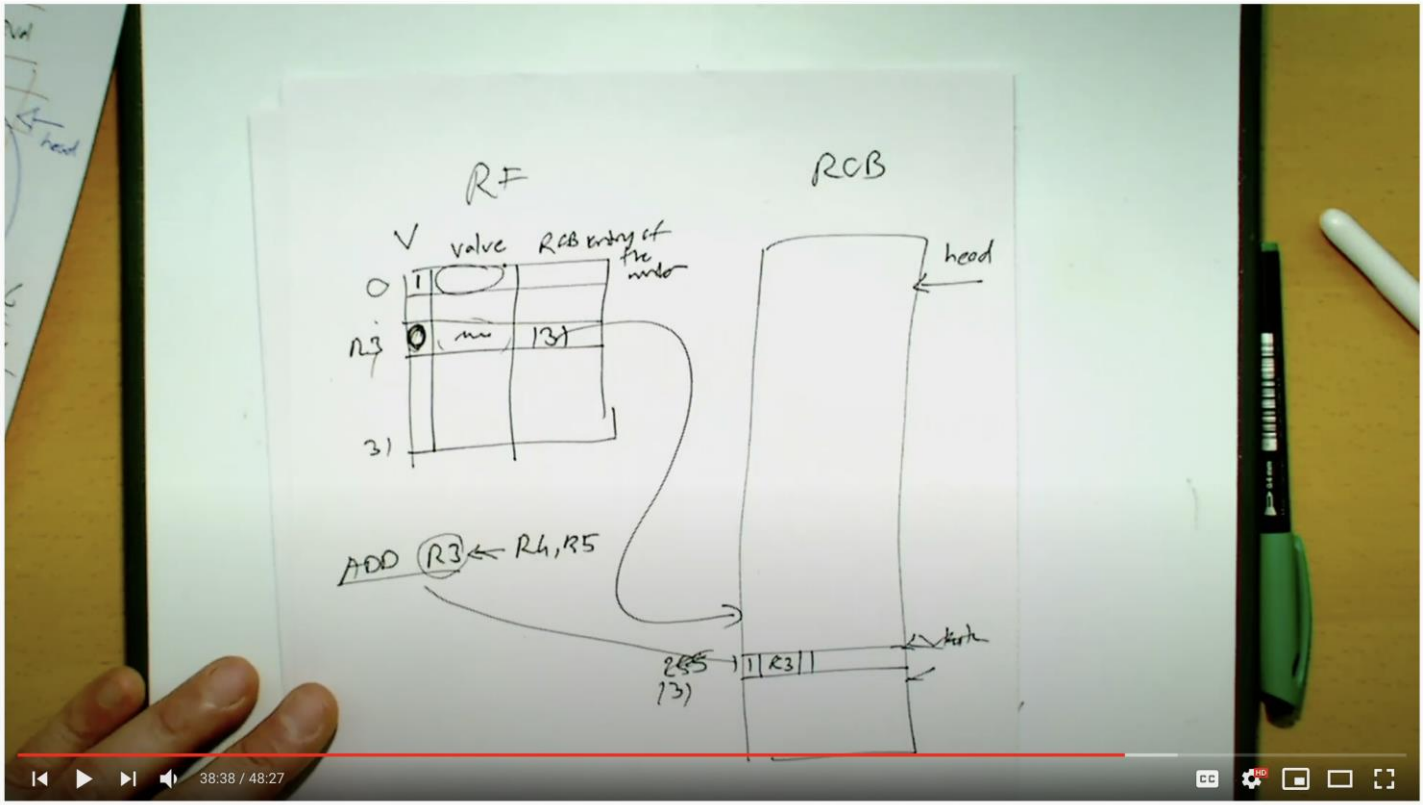
Onur Mutlu Lectures

23.8K subscribers

SUBSCRIBED

https://www.youtube.com/watch?v=upJPVXEuqIQ&list=PL5Q2soXY2Zi-iBn_sw_B63HtdbTNmphLc&index=18

More on State Maintenance & Precise Exceptions



ETH ZÜRICH HAUPTGEBÄUDE

Design of Digital Circuits - Lecture 15a: Reorder Buffer (ETH Zürich, Spring 2019)

2,462 views • Apr 15, 2019

35 0 SHARE SAVE ...

Onur Mutlu Lectures
15.6K subscribers

ANALYTICS EDIT VIDEO

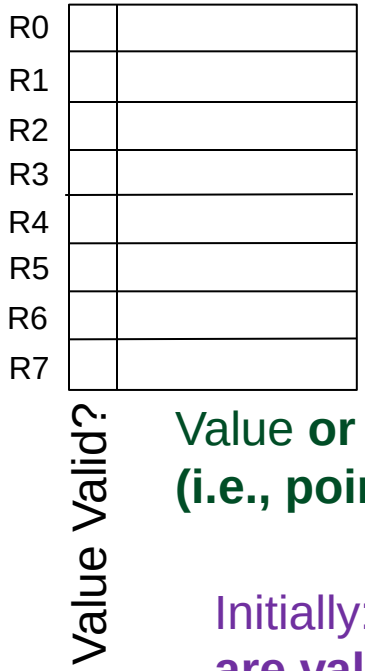
Design of Digital Circuits, ETH Zürich, Spring 2019 (https://safari.ethz.ch/digitaltechnik...)
Professor Onur Mutlu (http://people.inf.ethz.ch/omutlu)

Lecture 15a: Reorder Buffer
Lecturer: Onur Mutlu
Date: April 11, 2019

https://www.youtube.com/watch?v=9yo3yhUijQs&list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&index=17

Reorder Buffer Example

Register File (RF)



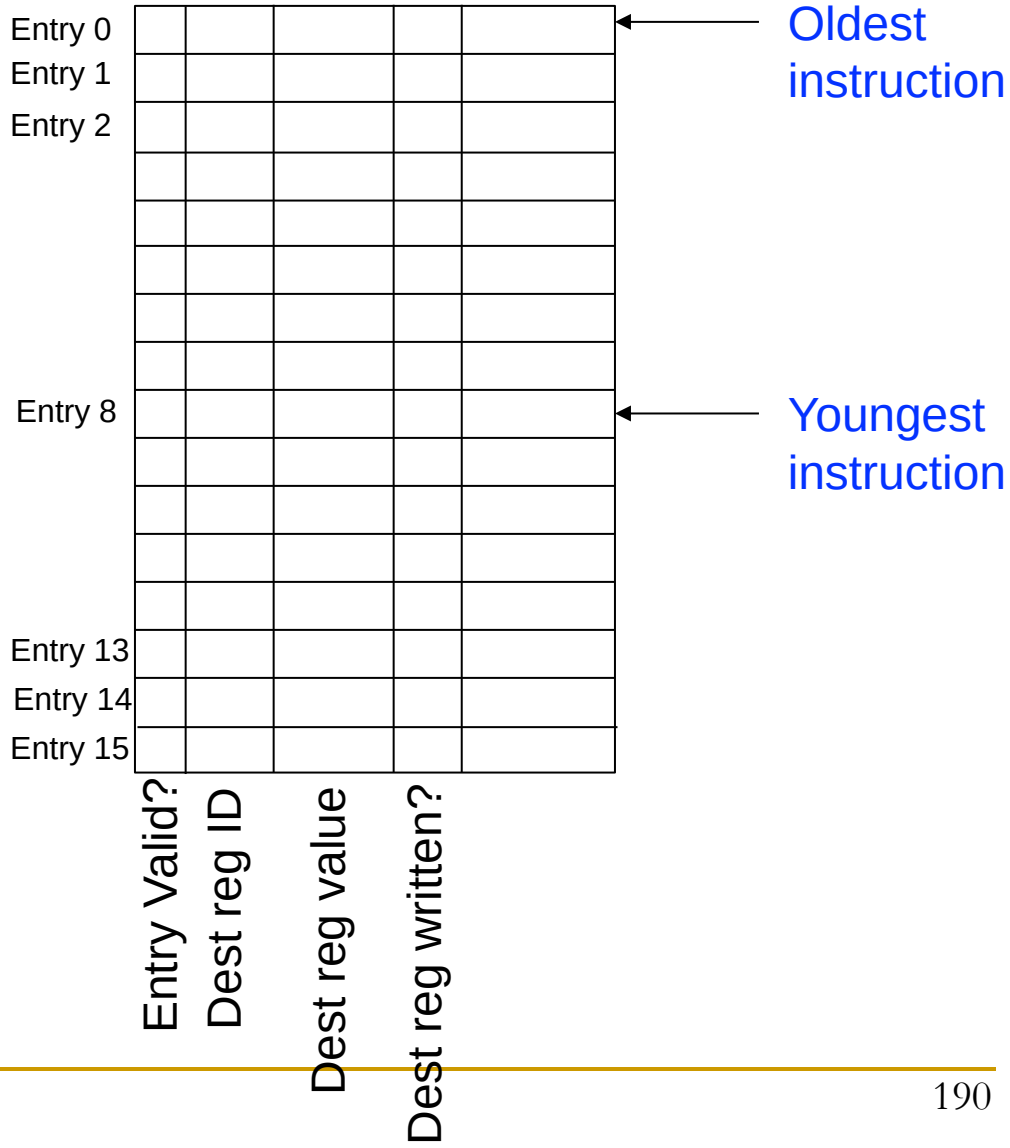
Value **or Tag**
(i.e., pointer to ROB entry)

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

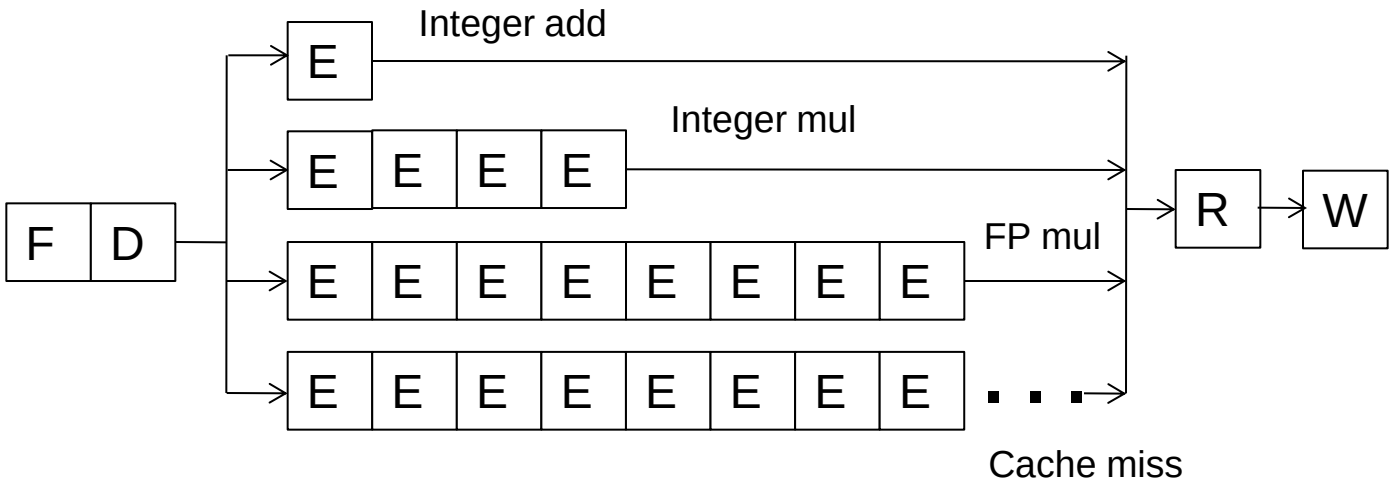
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer (ROB)



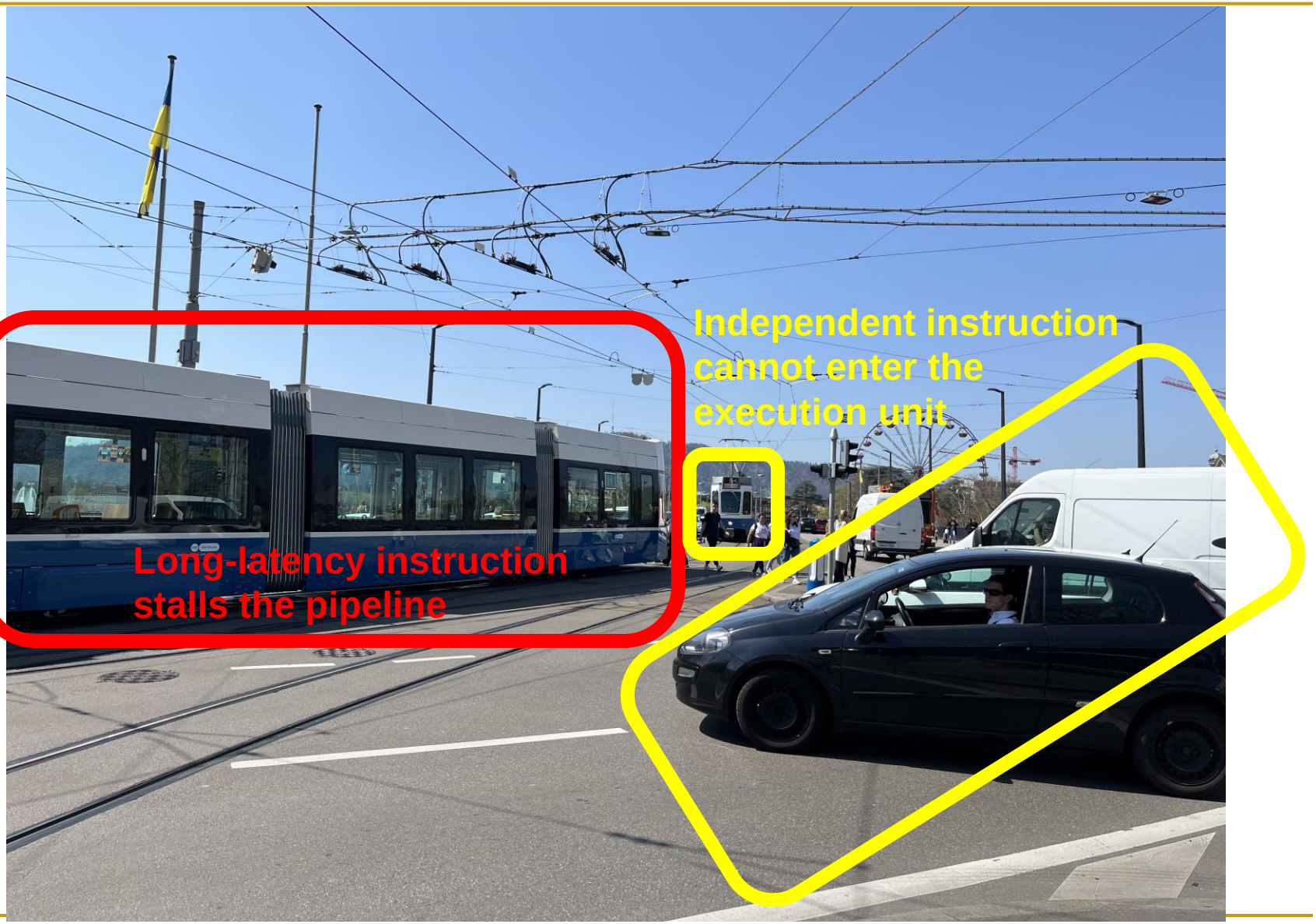
Out-of-Order Execution (Dynamic Instruction Scheduling)

An In-order Pipeline



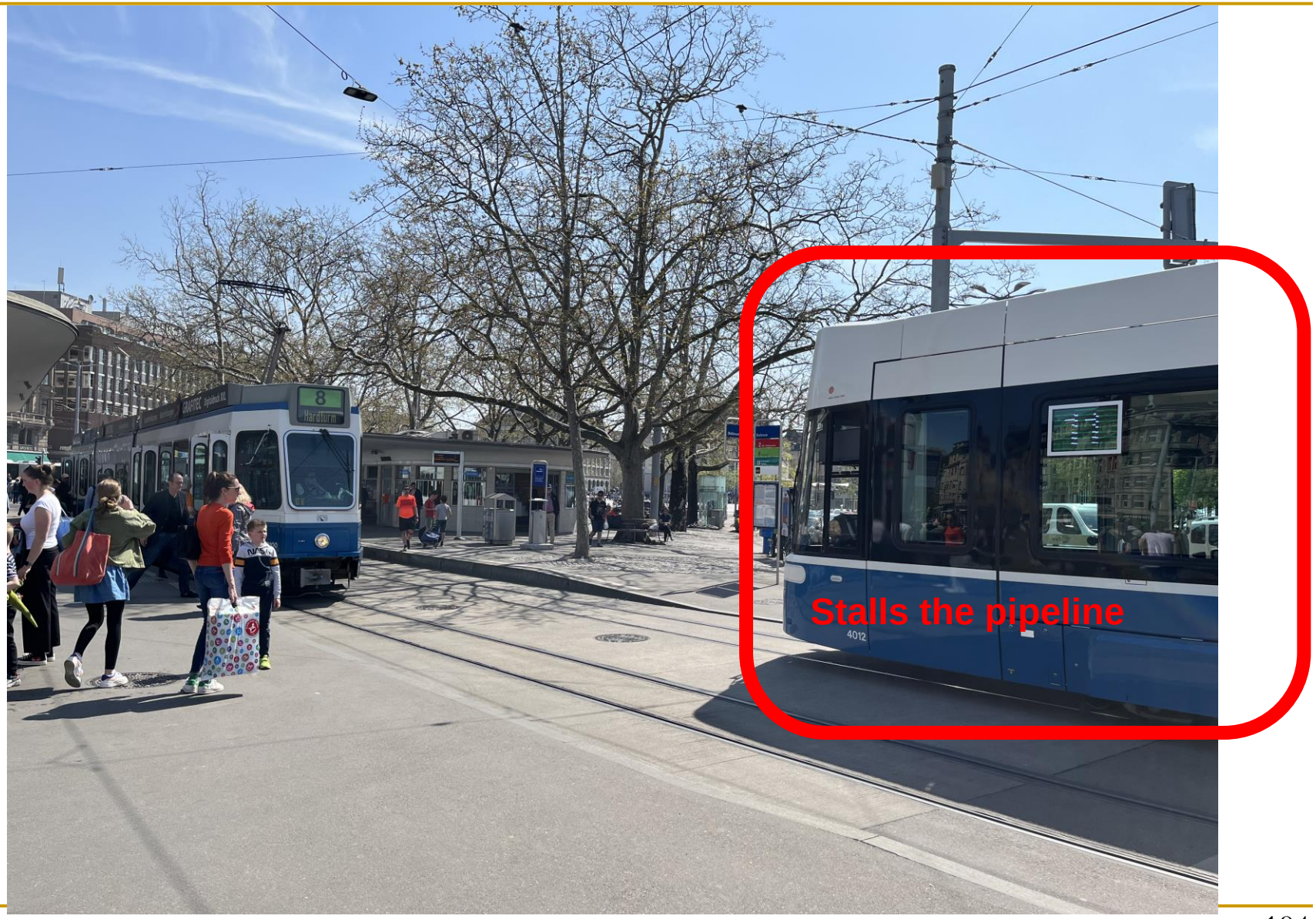
- Dispatch: Act of sending an instruction to a functional unit
- Renaming with ROB eliminates stalls due to false dependences
- Problem: A non-ready instruction stalls dispatch of younger instructions into functional (execution) units

An Example Non-Ready Instruction



Time: 12:55

An Example Non-Ready Instruction



Time: 12:57

An Example Non-Ready Instruction

Time: 12:58

An Example Non-Ready Instruction

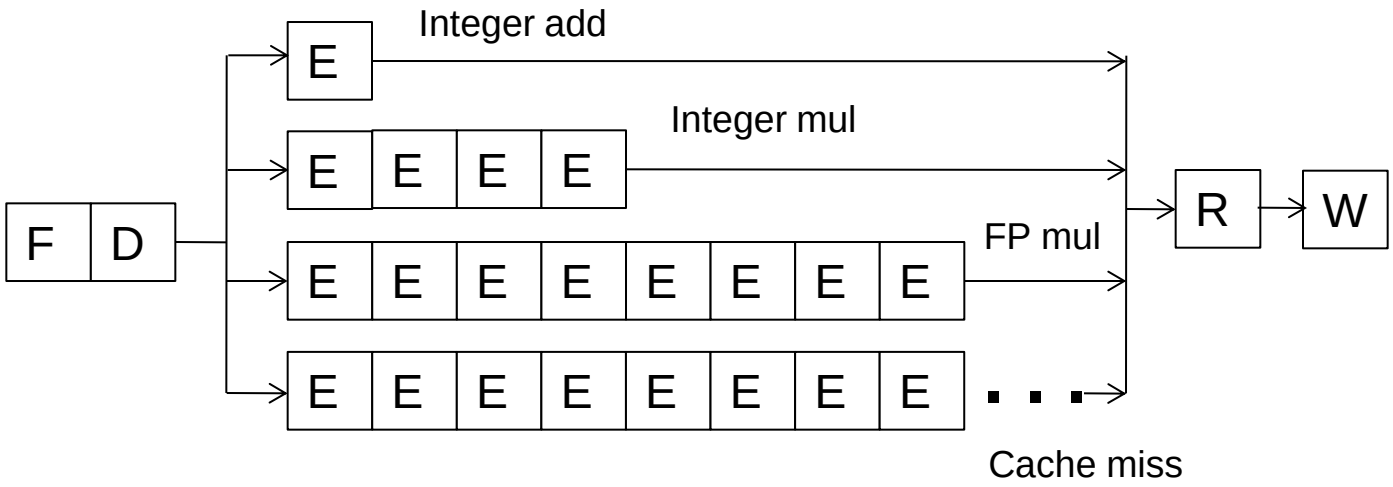
Time: 13:00

Another View

Stalling Done & Independents Execute

Time: 13:06

An In-order Pipeline



- Dispatch: Act of sending an instruction to a functional unit
- Renaming with ROB eliminates stalls due to false dependences
- **Problem: A non-ready instruction stalls dispatch of younger instructions into functional (execution) units**

Can We Do Better?