

Digital Design & Computer Arch.

Lecture 14: Pipelined Processor Design

Prof. Onur Mutlu

ETH Zürich

Spring 2022

8 April 2022

Extra Assignment: Moore's Law (I)

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

- **Optional Assignment – for 1% extra credit**
 - **Write a 1-page review**
 - Upload PDF file to Moodle – **Deadline: April 11**

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

Extra Assignment 2: Moore's Law (II)

■ Guidelines on how to review papers critically

- ❑ **Guideline slides:** [pdf](#) [ppt](#)
- ❑ **Video:** <https://www.youtube.com/watch?v=tOL6FANAj8c>
- ❑ Example reviews on "Main Memory Scaling: Challenges and Solution Directions" ([link to the paper](#))
 - [Review 1](#)
 - [Review 2](#)
- ❑ Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" ([link to the paper](#))
 - [Review 1](#)

Agenda for Today & Next Few Lectures

- Last week: **Microarchitecture Fundamentals**
 - Single-cycle Microarchitectures
 - Multi-cycle Microarchitectures

- This week: **Pipelining**
 - Pipelining
 - Pipelined Processor Design
 - Control & Data Dependence Handling
 - Precise Exceptions: State Maintenance & Recovery

- Next week+: **Out-of-Order Execution**
 - Out-of-Order Execution
 - Issues in OoO Execution: Load-Store Handling, ...

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

Readings

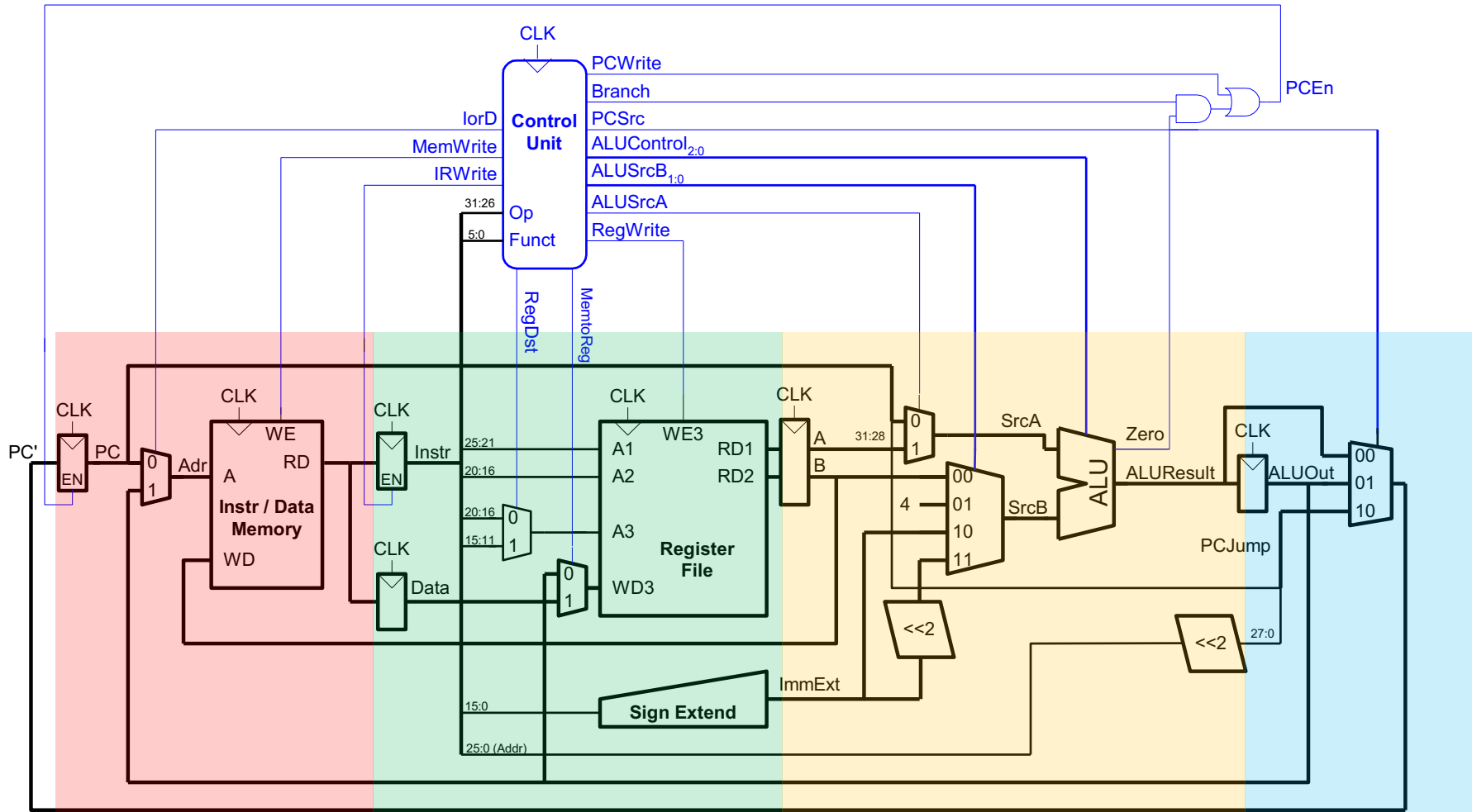
■ This week

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

■ Next week

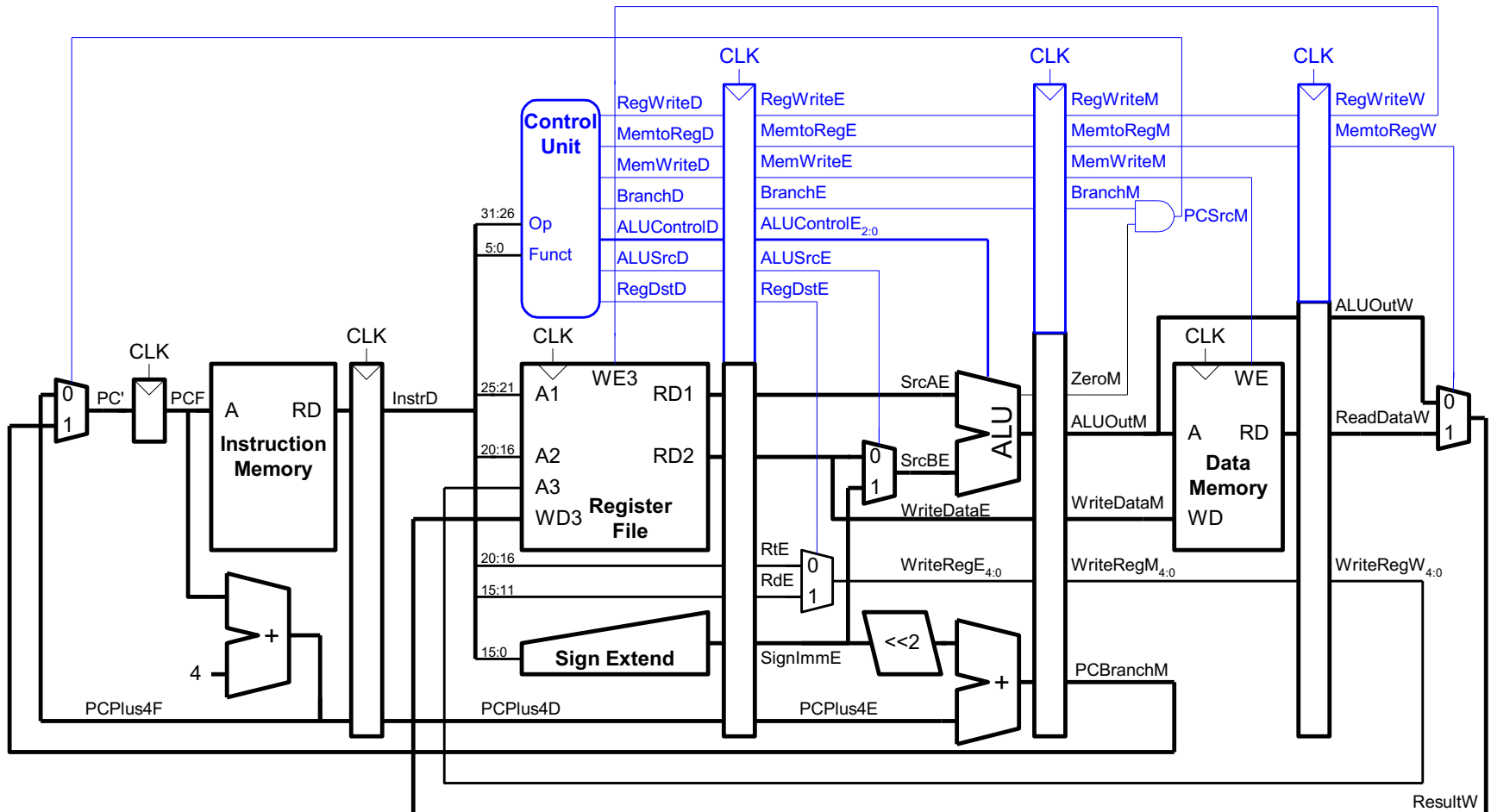
- Out-of-order execution
 - H&H, Chapter 7.8-7.9
- Smith & Sohi, “**The Microarchitecture of Superscalar Processors,**”
Proceedings of the IEEE, 1995
 - More advanced pipelining
 - Interrupt and exception handling
 - Out-of-order and superscalar execution concepts

Review: Pipelining Basic Idea



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

Review: Pipelined Datapath & Control

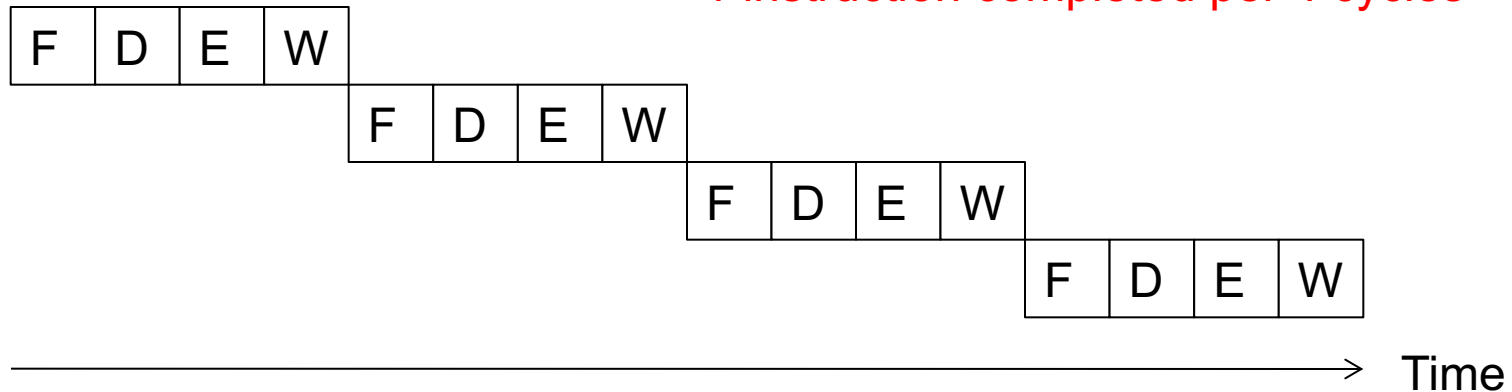


- Same control unit as single-cycle processor
Control delayed to proper pipeline stage

Review: Execution of Four Independent ADDs

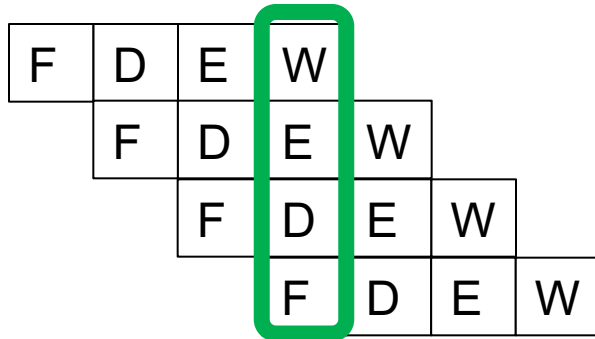
- Multi-cycle: 4 cycles per instruction

1 instruction completed per 4 cycles



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

1 instruction completed per cycle



Is life always this beautiful?

Review: 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*

Review: Causes of Pipeline *Stalls*


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

Data Dependence Handling: Concepts and Implementation

Review: Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence

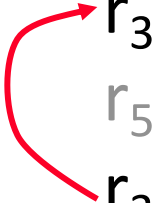
$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Output dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$

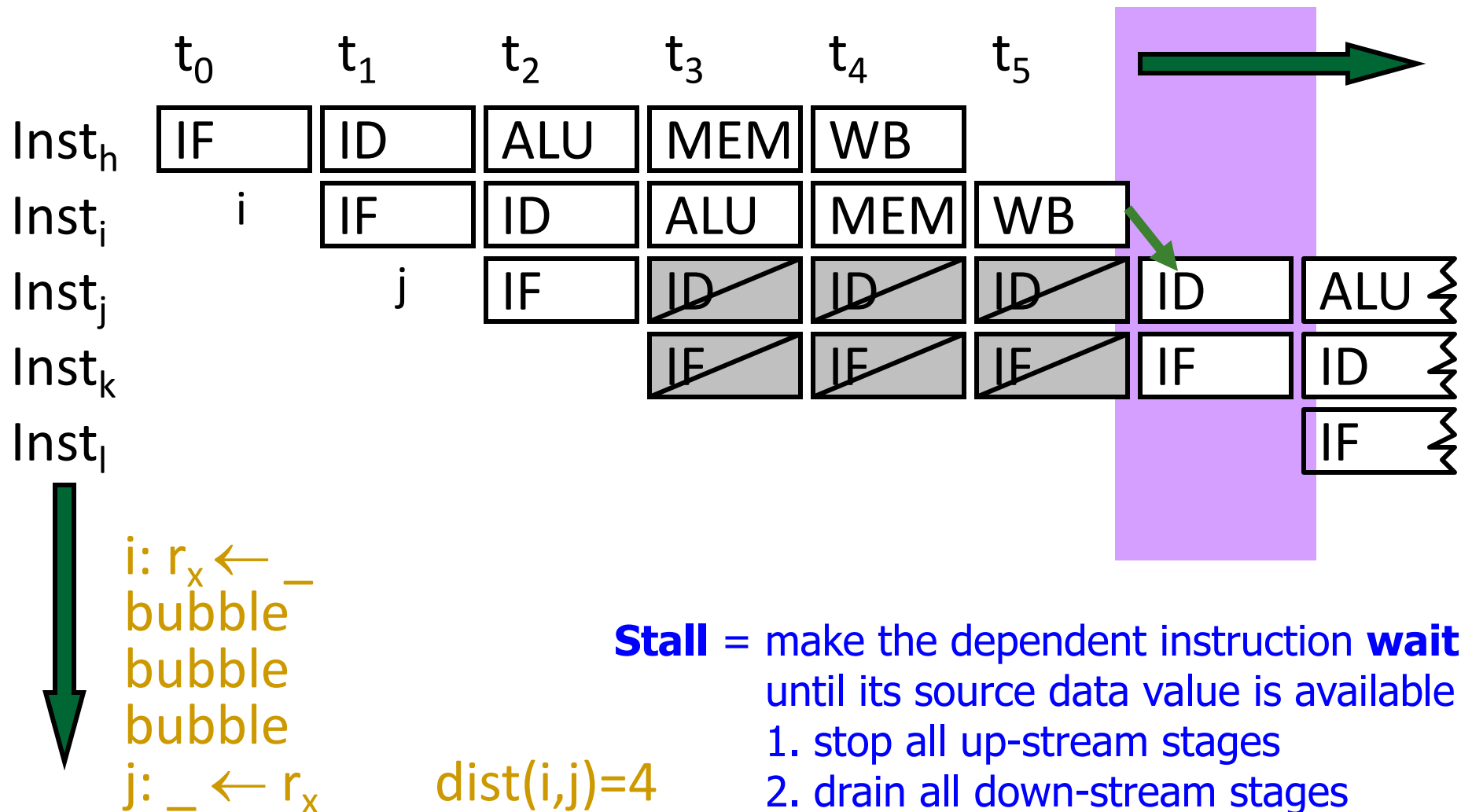


Write-after-Write
(WAW)

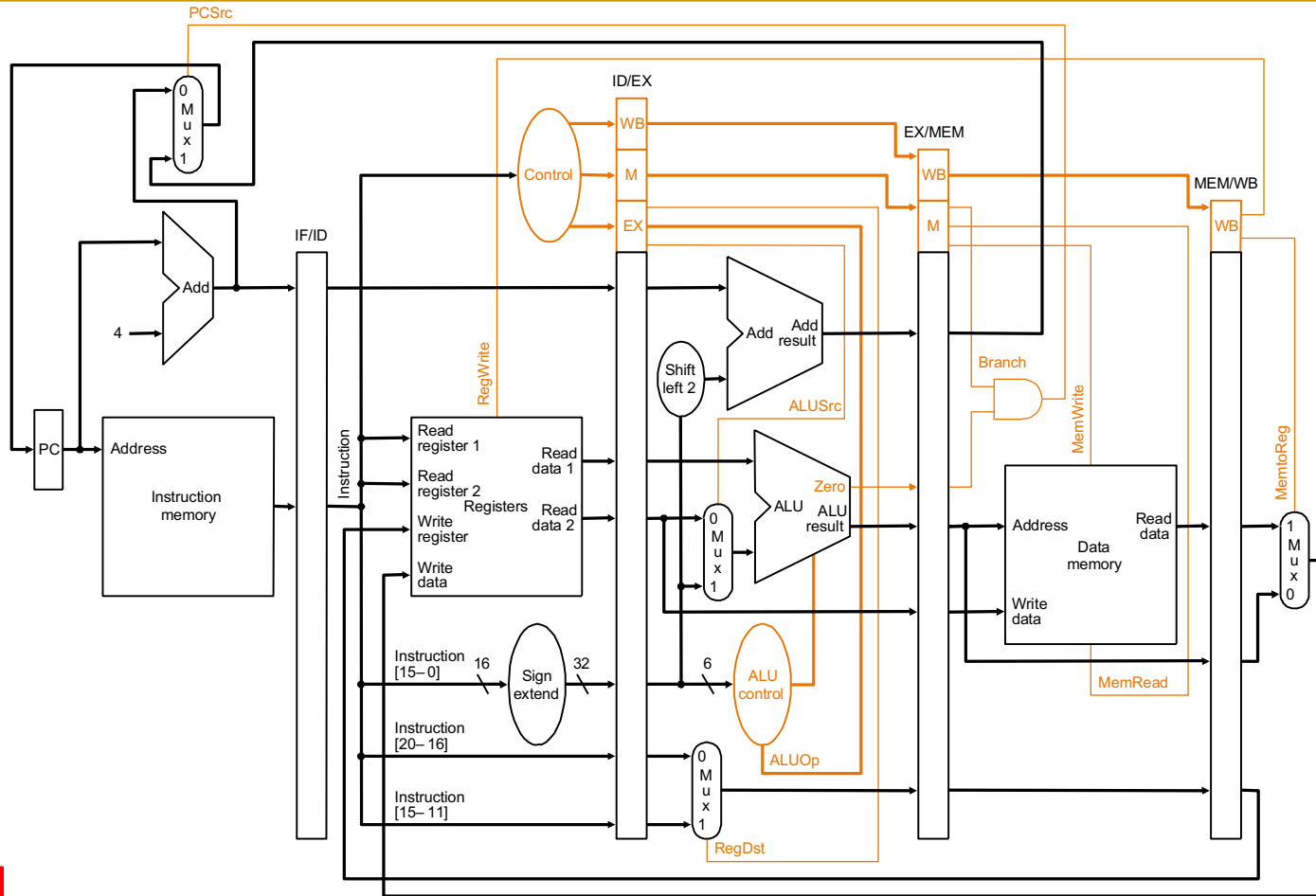
Review: How to Handle Data Dependences

- 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

Review: Pipeline Stall: Resolving Data Dependence



Review: How to Implement Stalling



■ Stall

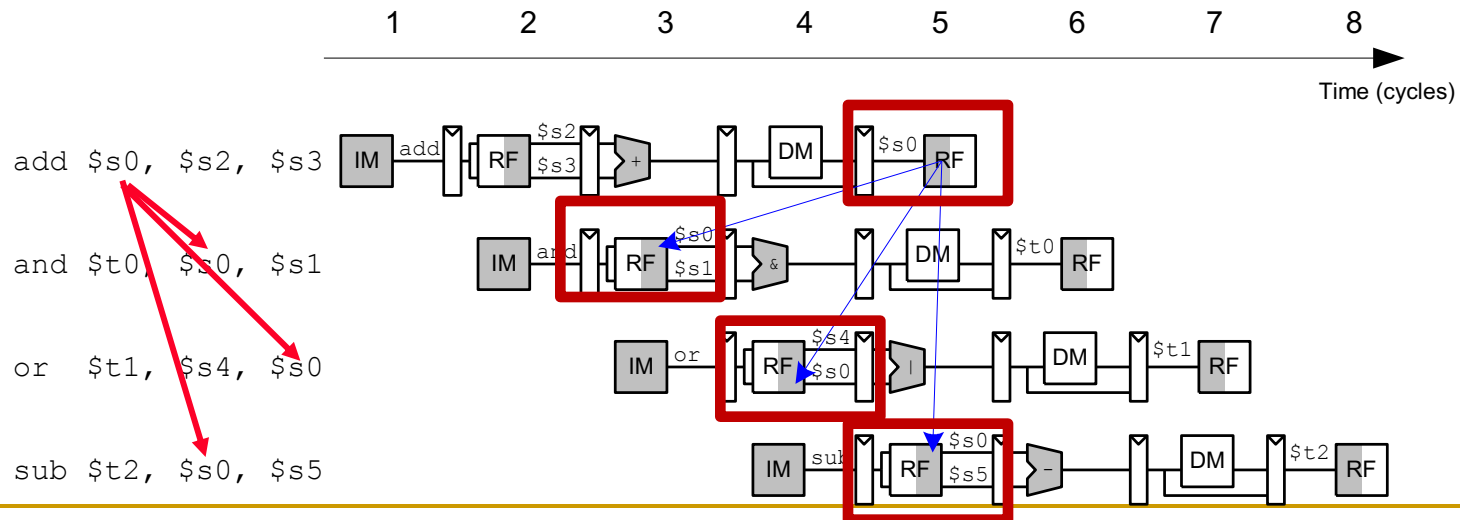
- ❑ disable **PC** and **IF/ID** latching; ensure stalled instruction stays in its stage
- ❑ Insert **"invalid"** instructions/nops into the stage following the stalled one (called **"bubbles"**)

Review: RAW Data Dependence Example

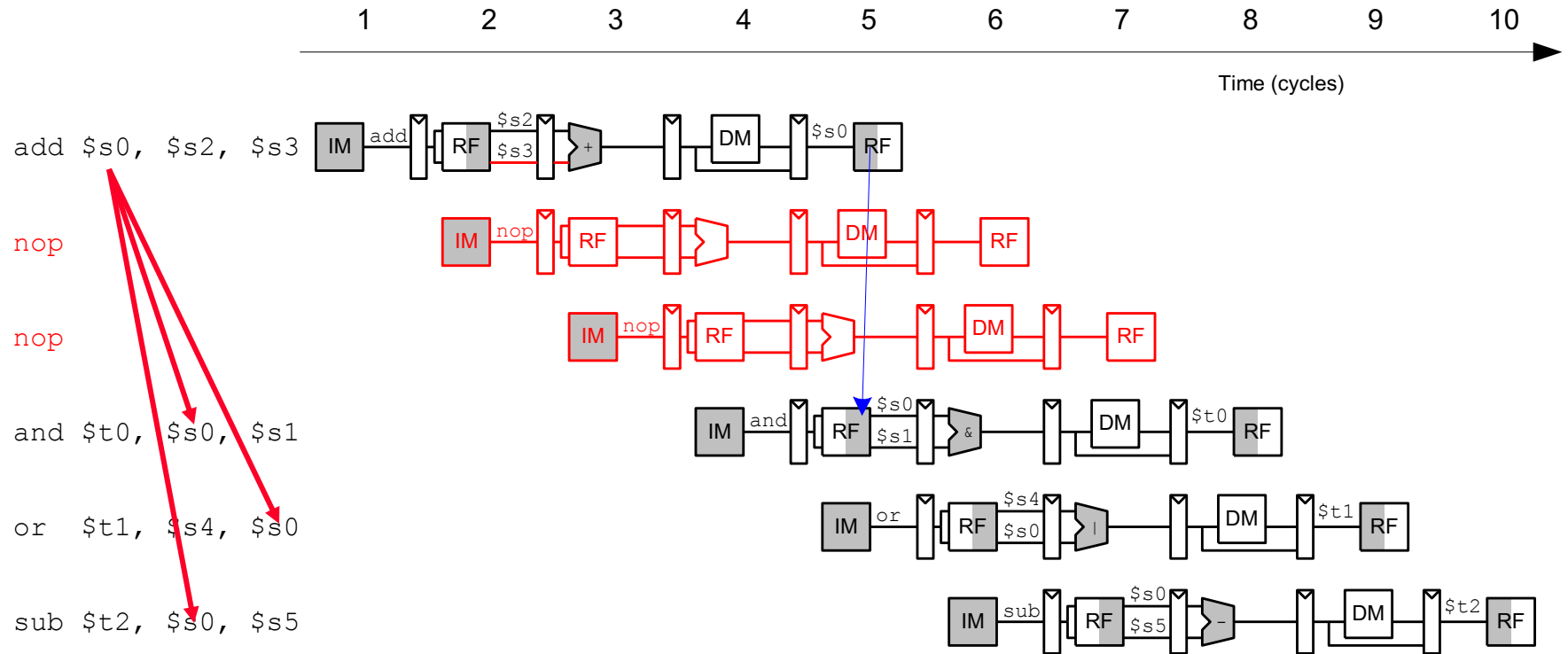
- One instruction writes a register (\$s0) and next instructions read this register => read after write (RAW) dependence.

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

- subsequent instructions read the correct value of \$s0



Review: Compile-Time Detection and Elimination

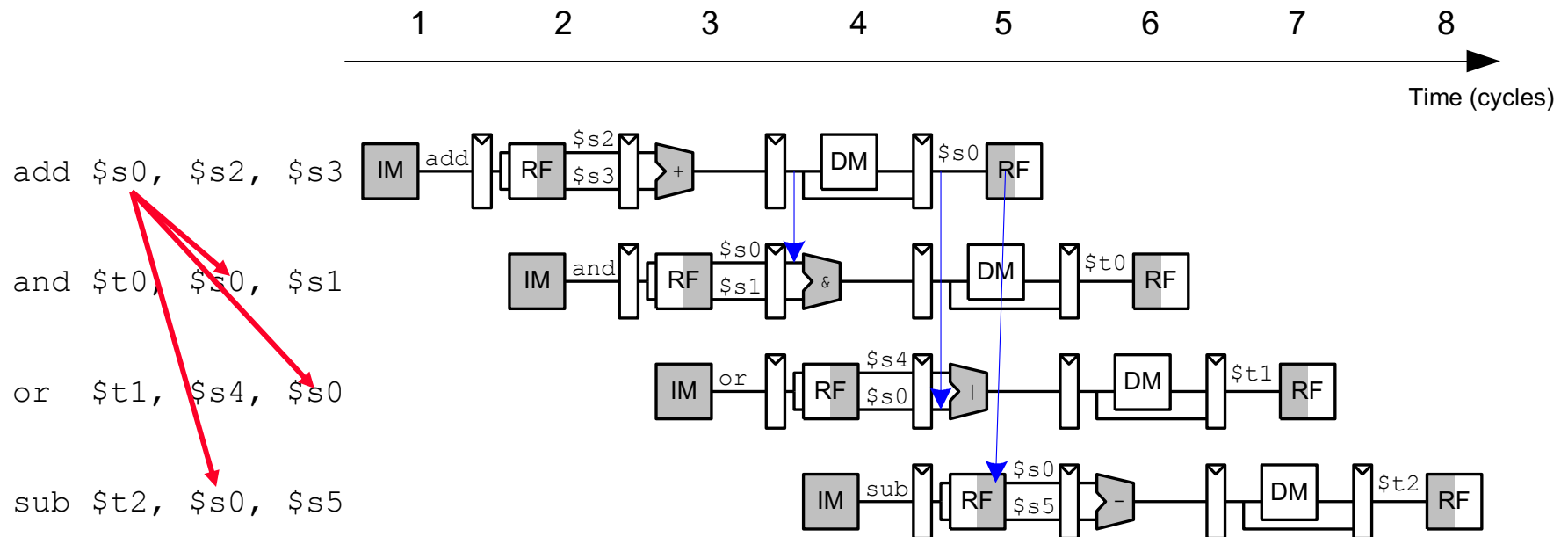


- Insert enough independent instructions for the required result to be ready by the time it is needed by a dependent one
 - Reorder/reschedule/insert instructions at the compiler level

Review: Data Forwarding

- Also called Data Bypassing
 - Forward the result value to the dependent instruction as soon as the value is available
 - We have already seen the basic idea before
 - Remember dataflow?
 - 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



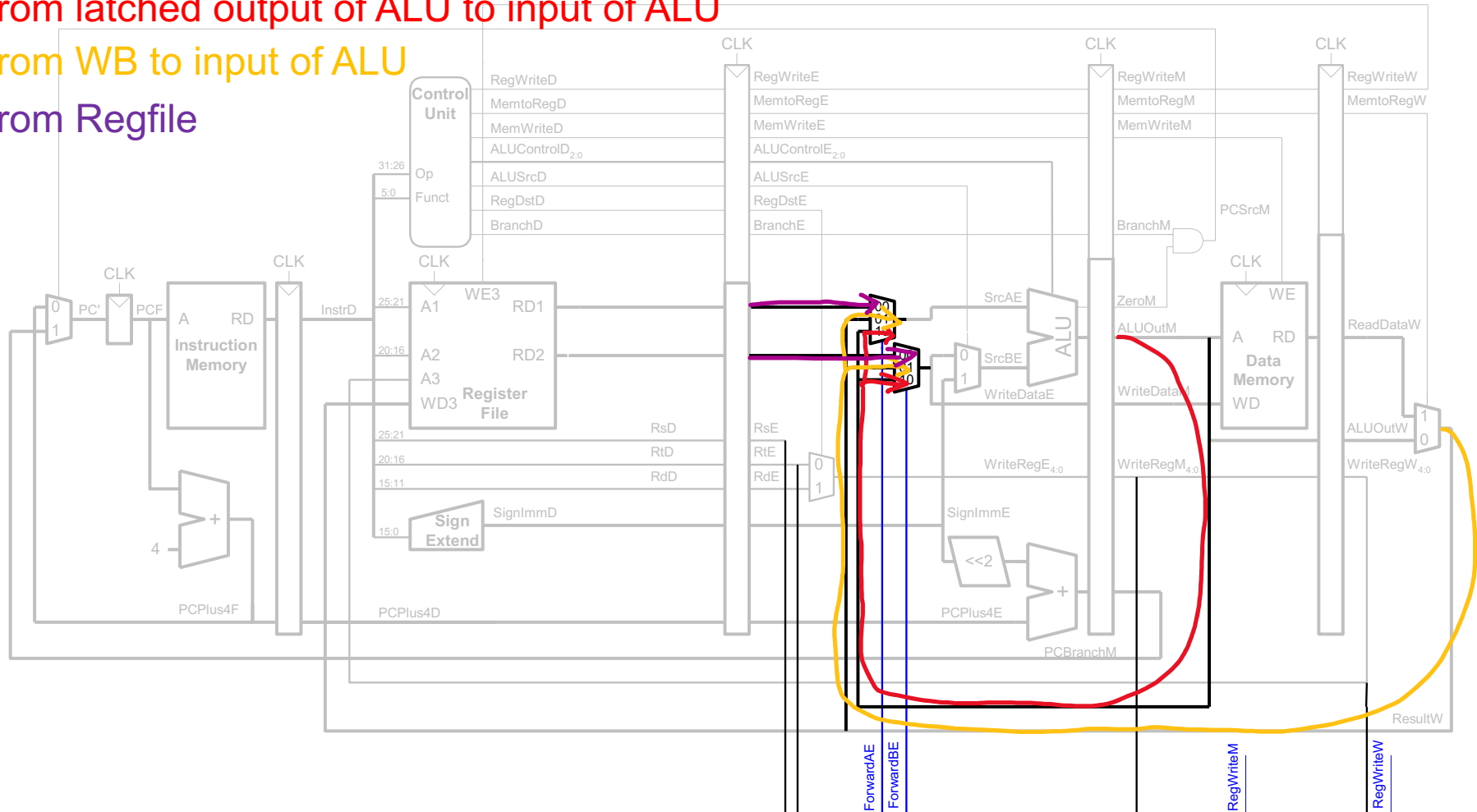
From latched output of ALU to input of ALU
From WB to input of ALU
From WB to RF (internal in Register File)

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:

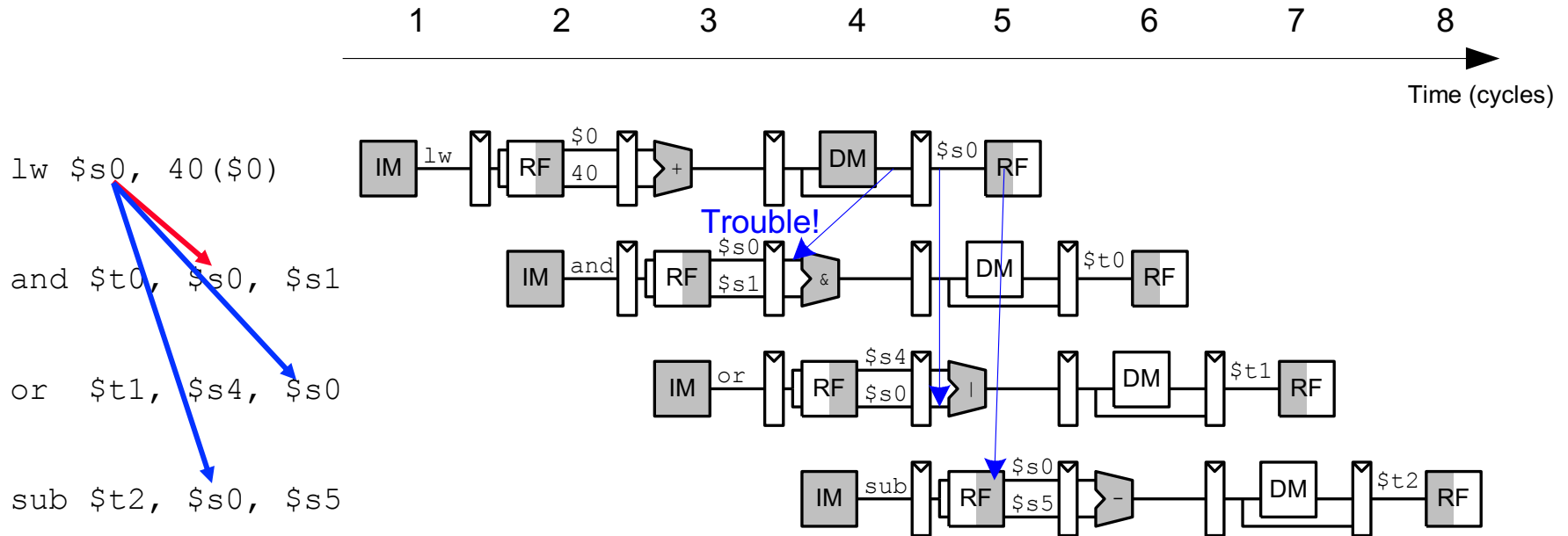
- Memory stage or
- Writeback stage

- 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
```

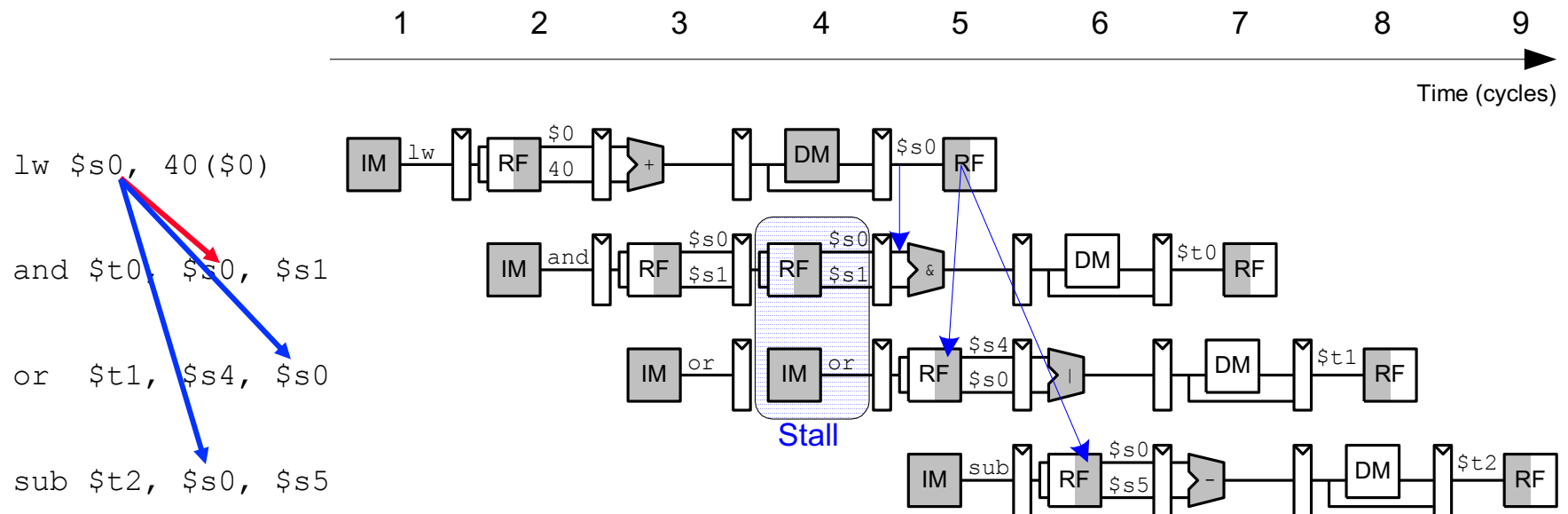
- Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*
-

Data Forwarding Is **Not** Always Possible

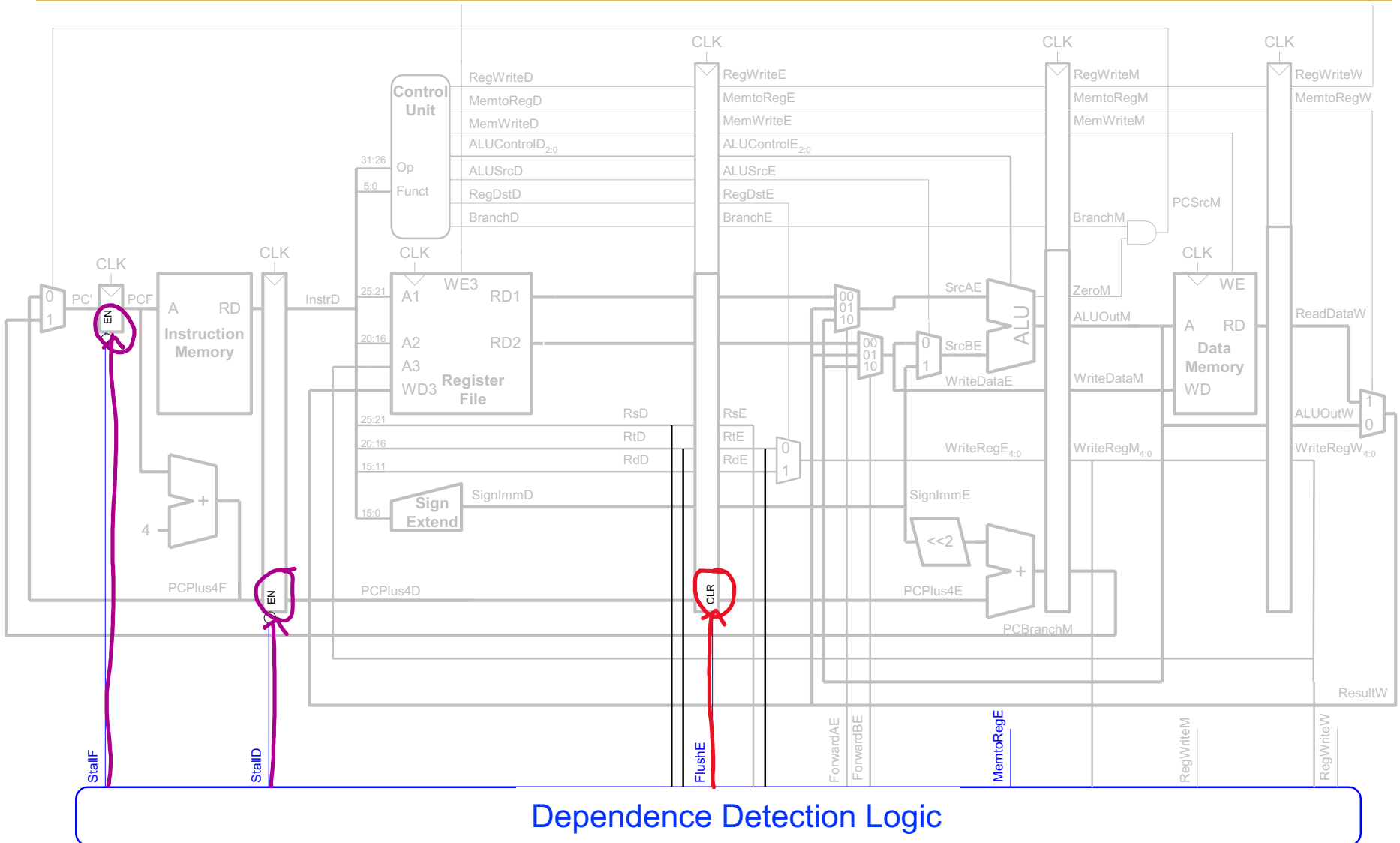


- Forwarding is usually sufficient to resolve RAW data dependences
- Unfortunately, there are cases when forwarding is **not possible**
 - ❑ due to pipeline design and instruction latencies
 - ❑ The 1w instruction **does not finish** reading data until the end of Memory stage
→ its result **cannot be 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

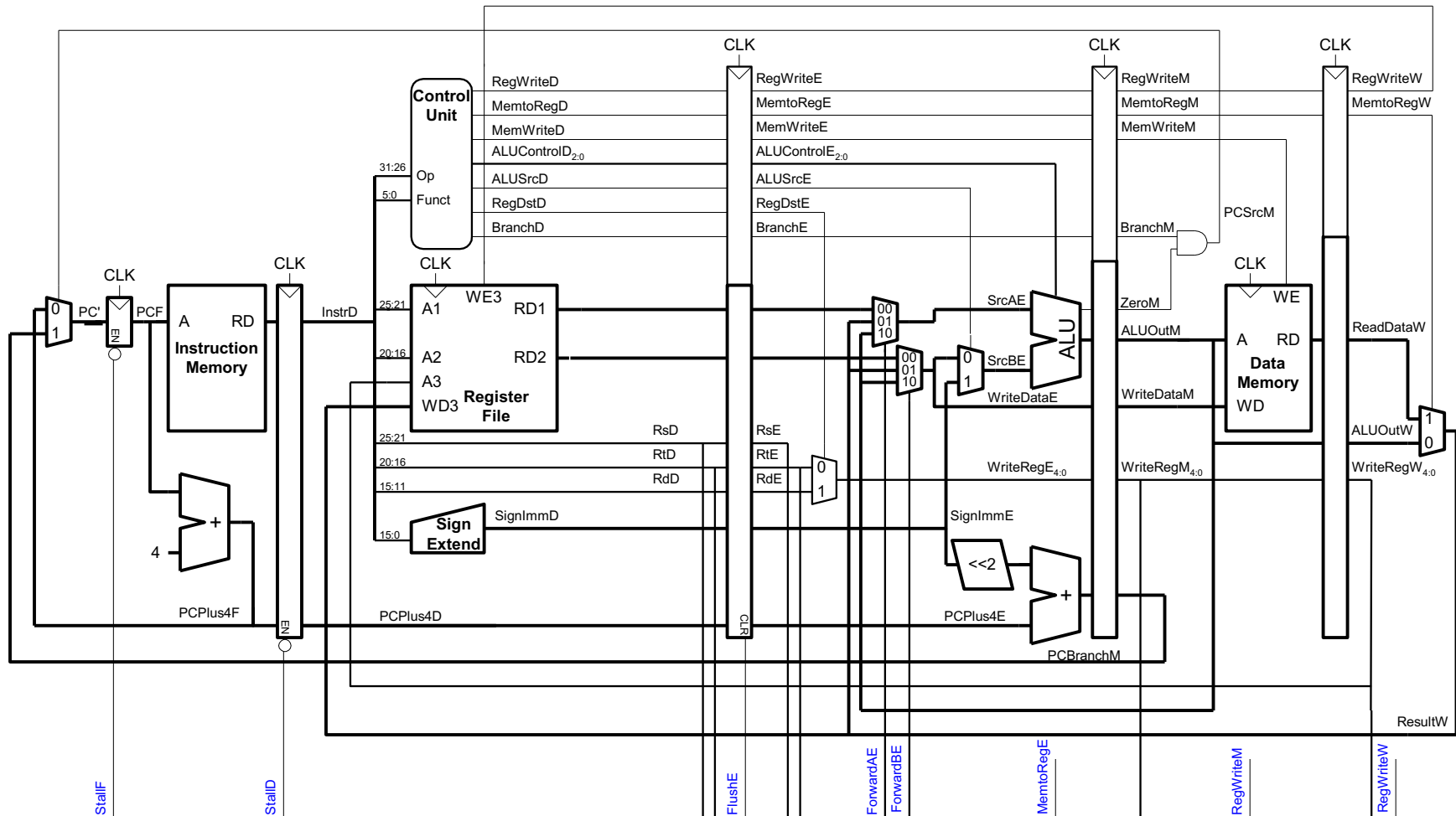
Control Dependence

- 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?

Branch Prediction

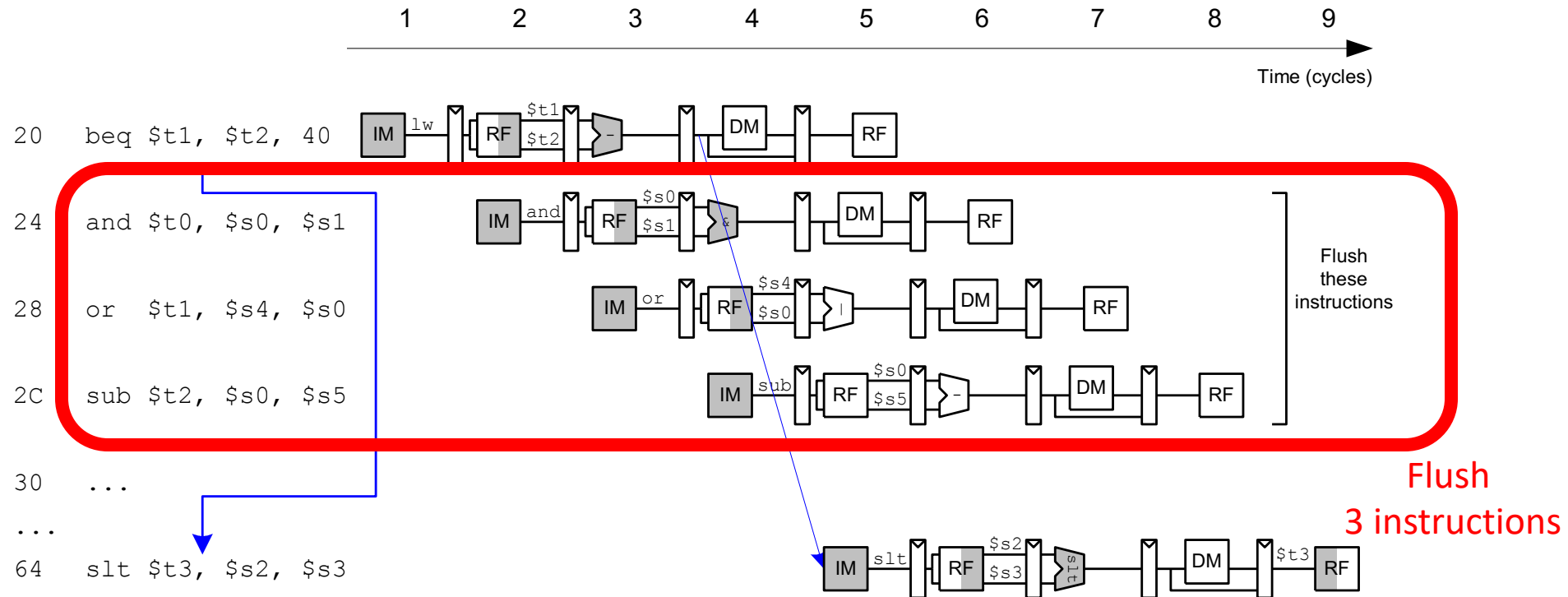
- Special case of data dependence: **dependence on PC**
- **beq:**
 - Conditional branch is not resolved until the fourth stage of the pipeline
 - **Instructions after the branch are fetched before branch is resolved**
 - Simple “**branch prediction**” example:
 - Always predict that the next sequential instruction is fetched
 - Called “**Always not taken**” prediction
 - Flush (invalidate) such instructions if the branch is taken
- **Branch misprediction penalty**
 - **number of instructions flushed when branch is incorrectly predicted**
 - Penalty can be reduced by resolving the branch earlier
 - Called “**Early branch resolution**”

Our Pipeline So Far



Dependence Detection Logic

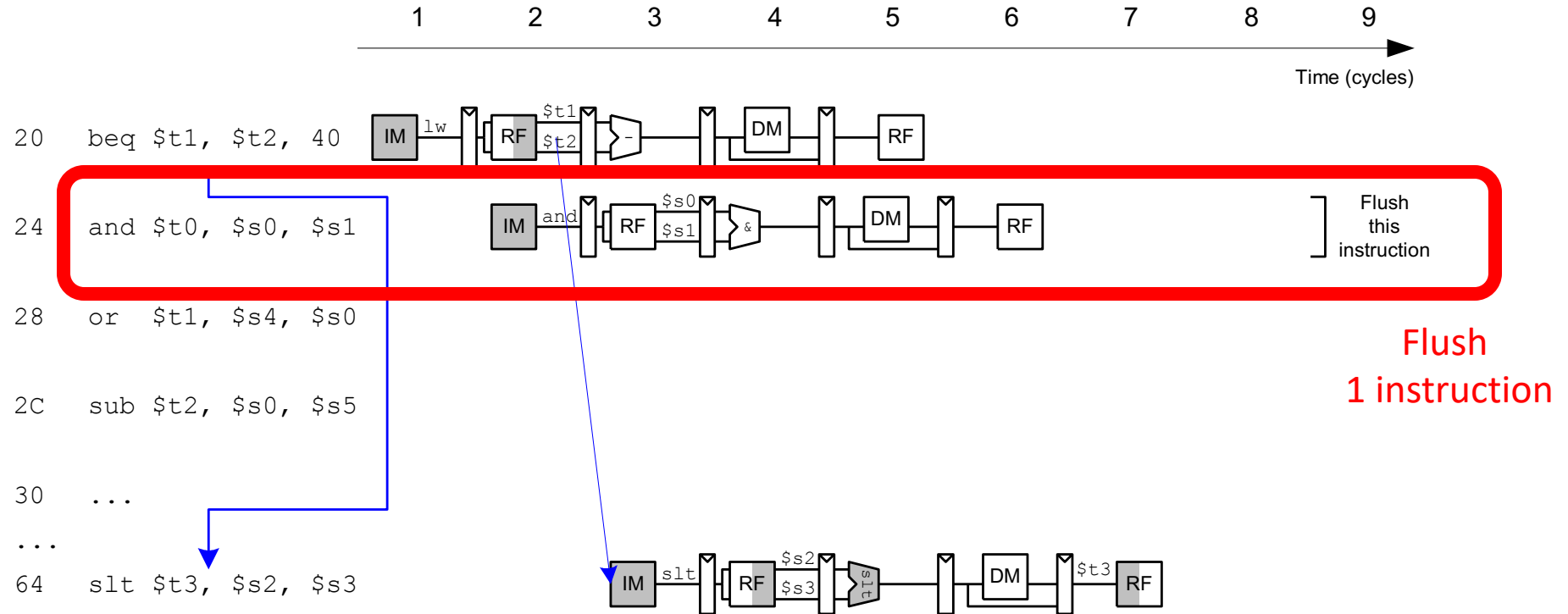
Control Dependence: Flush on Misprediction



Need to calculate branch target and condition in the Decode Stage



Early Branch Resolution



Early Branch Resolution: Good Idea?

■ Advantages

- Reduced branch misprediction penalty
 - Reduced CPI (cycles per instruction)

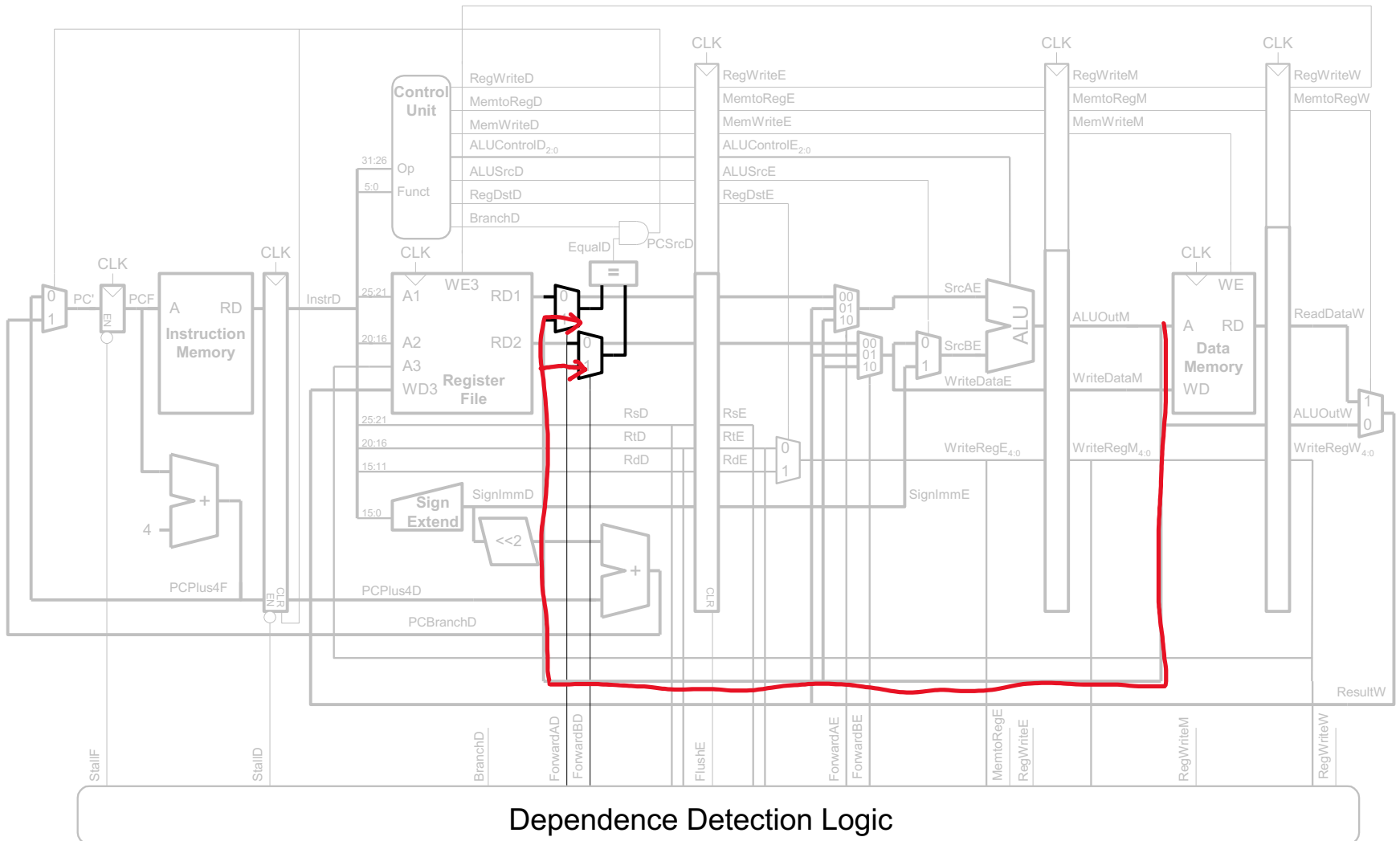
■ Disadvantages

- Potential increase in clock cycle time?
 - Higher clock period and lower frequency?
- Additional hardware cost
 - Specialized and likely not used by other instructions

Recall: Performance Analysis Basics

- Execution time of a single instruction
 - **{CPI} x {clock cycle time}**
 - CPI: Number of cycles it takes to execute an instruction
- Execution time of an entire program
 - Sum over all instructions [**{CPI} x {clock cycle time}**]
 - **{# of instructions} x {Average CPI} x {clock cycle time}**

Data Forwarding for Early Branch Resolution



Data forwarding for early branch resolution adds even more complexity

Forwarding and Stalling Hardware Control

```
// Forwarding logic:
assign ForwardAD = (rsD != 0) & (rsD == WriteRegM) & RegWriteM;
assign ForwardBD = (rtD != 0) & (rtD == WriteRegM) & RegWriteM;

//Stalling logic:
assign lwstall = ((rsD == rtE) | (rtD == rtE)) & MemtoRegE;

assign branchstall = (BranchD & RegWriteE &
                      (WriteRegE == rsD | WriteRegE == rtD))
                    |
                      (BranchD & MemtoRegM &
                      (WriteRegM == rsD | WriteRegM == rtD));

// Stall signals;
assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FlushE = lwstall | branchstall;
```

Final Pipelined MIPS Processor (H&H)

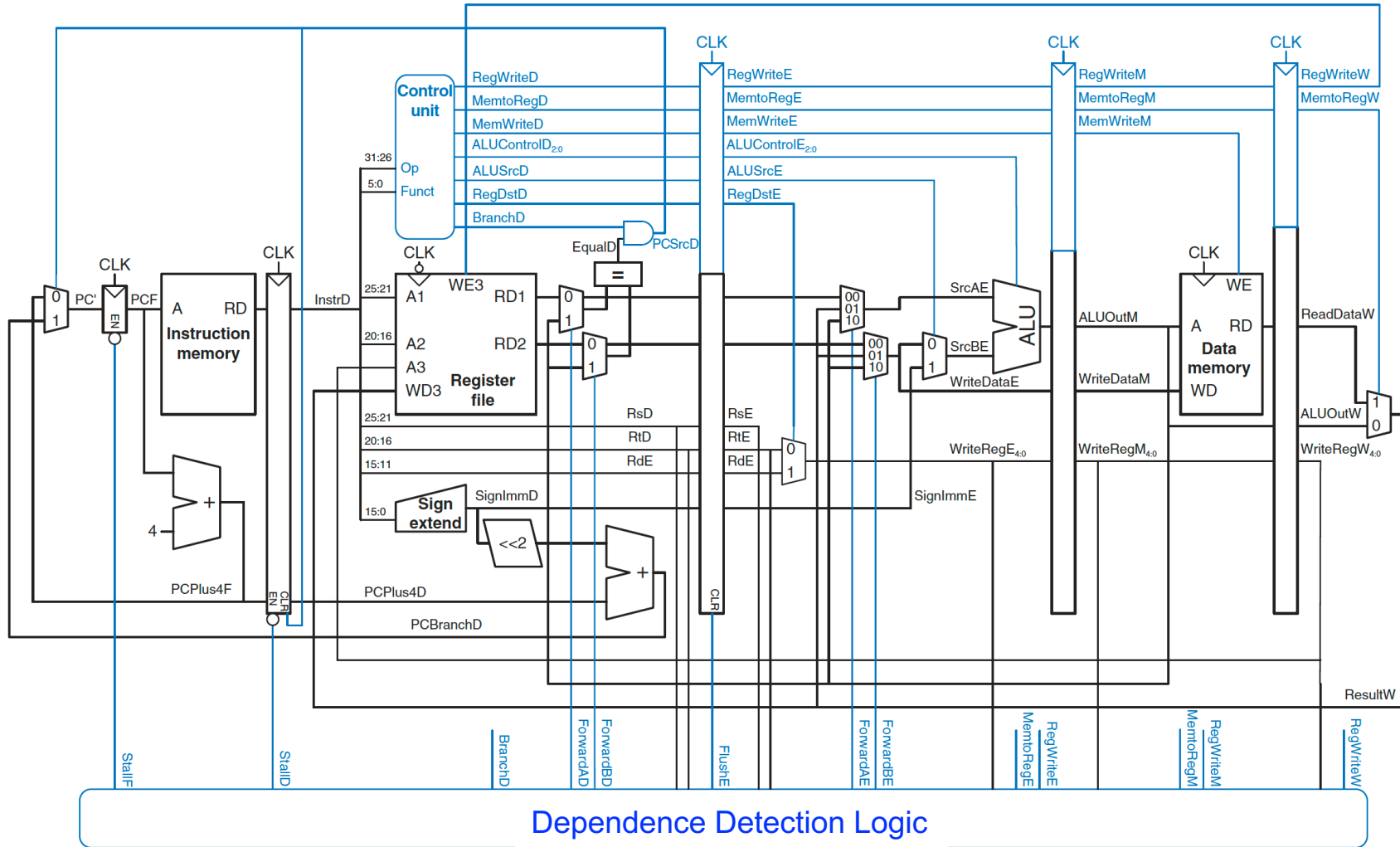


Figure 7.58 Pipelined processor with full hazard handling

Includes always-taken br prediction, early branch resolution, forwarding, stall logic


Doing Better: Smarter Branch Prediction

- **Guess whether or not branch will be taken**
 - Backward branches are usually taken (loops iterate many times)
 - History of whether branch was previously taken can improve the guess
- **Accurate branch prediction reduces the fraction of branches requiring a flush**
- **Many sophisticated techniques are employed in modern processors**
 - Including simple machine learning methods (perceptrons)
 - We will see them in Branch Prediction lectures

More on Branch Prediction (I)

Importance of The Branch Problem

- Assume $N = 20$ (20 pipe stages), $W = 5$ (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch
- How long does it take to fetch 500 instructions?
 - 100% accuracy
 - 100 cycles (all instructions fetched on the correct path)
 - No wasted work; $IPC = 500/100$
 - 99% accuracy
 - 100 (correct path) + $20 * 1$ (wrong path) = 120 cycles
 - 20% extra instructions fetched; $IPC = 500/120$
 - 90% accuracy
 - 100 (correct path) + $20 * 10$ (wrong path) = 300 cycles
 - 200% extra instructions fetched; $IPC = 500/300$
 - 60% accuracy
 - 100 (correct path) + $20 * 40$ (wrong path) = 900 cycles
 - 800% extra instructions fetched; $IPC = 500/900$



19:43 / 1:05:13

47 0 SHARE SAVE ...

ETH ZÜRICH
Digital Design & Computer Architecture - Lecture 16b: Branch Prediction I (ETH Zürich, Spring 2020)
2,612 views • Apr 20, 2020

Onur Mutlu Lectures
15.5K subscribers

Digital Design and Computer Architecture, ETH Zürich, Spring 2020
(<https://safari.ethz.ch/digitaltechnik...>)

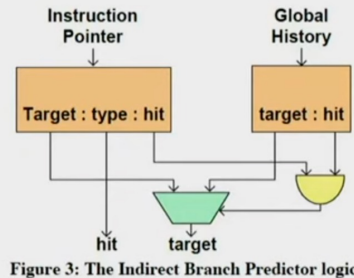
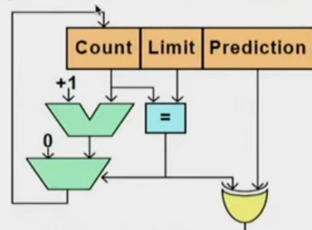
Lecture 16b: Branch Prediction I
Lecturer: Professor Onur Mutlu (<https://people.inf.ethz.ch/omutlu/>)
Date: April 23, 2020

ANALYTICS EDIT VIDEO

More on Branch Prediction (II)

Intel Pentium M Predictors: Loop and Jump

The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor's [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.



Gochman et al.,
"The Intel Pentium M Processor: Microarchitecture and Performance,"
Intel Technology Journal, May 2003.

39

ETH ZÜRICH

Digital Design & Computer Architecture - Lecture 17: Branch Prediction II (ETH Zürich, Spring 2020)

2,846 views • Apr 20, 2020



Onur Mutlu Lectures
15.5K subscribers

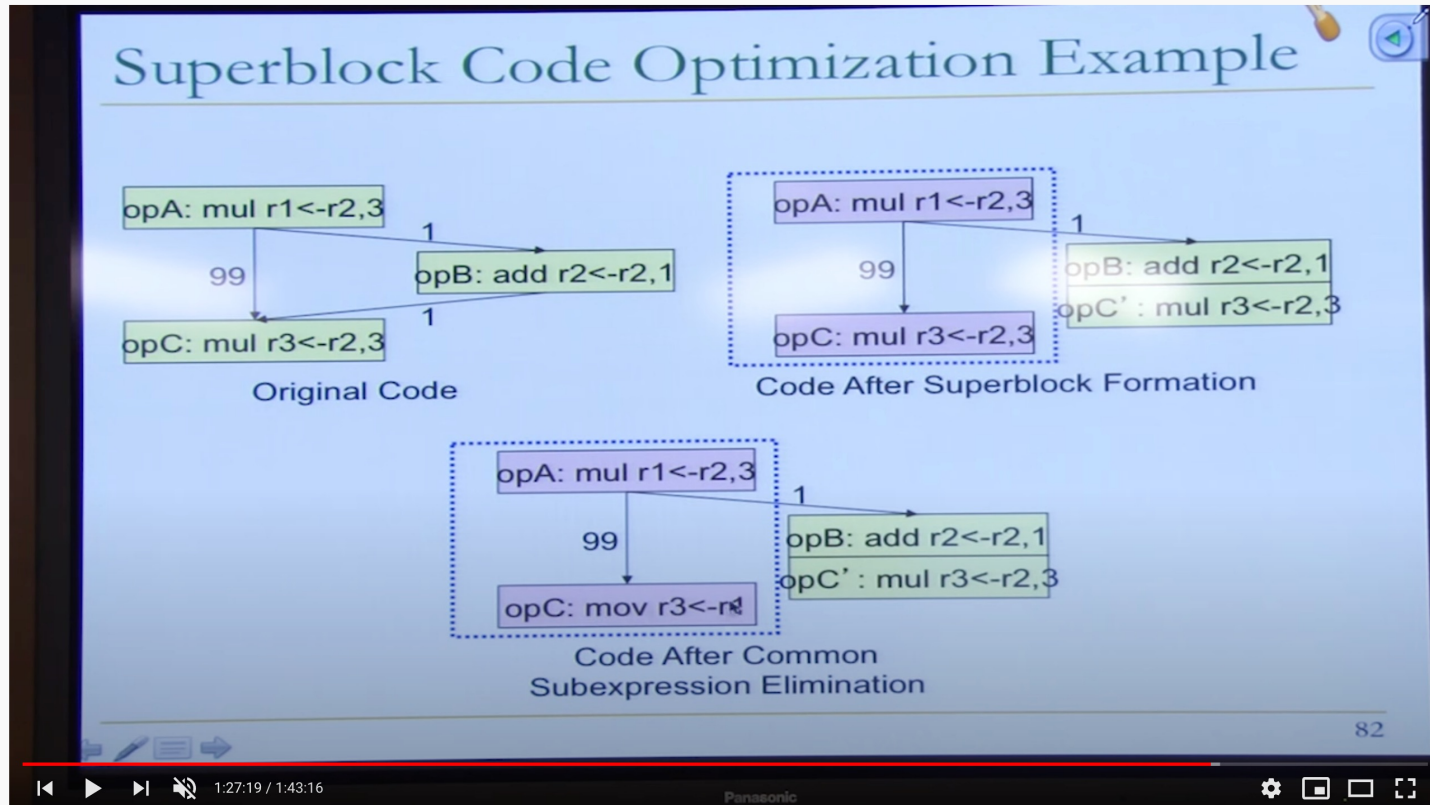
Digital Design and Computer Architecture, ETH Zürich, Spring 2020
(<https://safari.ethz.ch/digitaltechnik...>)

Lecture 17: Branch Prediction II
Lecturer: Professor Onur Mutlu (<https://people.inf.ethz.ch/omutlu/>)
Date: April 24, 2020

42 0 SHARE SAVE ...

ANALYTICS EDIT VIDEO

More on Branch Prediction (III)



18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5

4,696 views • Sep 23, 2015

41 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23K subscribers

Lecture 5: Advanced Branch Prediction

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 16, 2014.

Lecture 5 slides (pdf): <http://www.ece.cmu.edu/~ece740/f15/li...>

Lecture 5 slides (ppt): <http://www.ece.cmu.edu/~ece740/f15/li...>

SUBSCRIBED



<https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAU9N76TShJqfYDt&index=4>

Lectures on Branch Prediction

- Digital Design & Computer Architecture, Spring 2020, Lecture 16b
 - Branch Prediction I (ETH Zurich, Spring 2020)
 - https://www.youtube.com/watch?v=h6l9yYSyZHM&list=PL5Q2soXY2Zi_FRrloMa2fUYWPGiZUBQo2&index=22
- Digital Design & Computer Architecture, Spring 2020, Lecture 17
 - Branch Prediction II (ETH Zurich, Spring 2020)
 - https://www.youtube.com/watch?v=z77VpggShvg&list=PL5Q2soXY2Zi_FRrloMa2fUYWPGiZUBQo2&index=23
- Computer Architecture, Spring 2015, Lecture 5
 - Advanced Branch Prediction (CMU, Spring 2015)
 - <https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAU9N76TShJqfYDt&index=4>

Pipelined Performance Example

Pipelined Performance Example

- **An important program consists of:**
 - 25% loads
 - 10% stores
 - 11% branches
 - 2% jumps
 - 52% R-type
- **Suppose:**
 - 40% of loads used by next instruction
 - 25% of branches mispredicted
- **All jumps flush the next instruction fetched**
- **What is the average CPI?**

Pipelined Performance Example: CPI

- Load/Branch CPI = 1 when no stall/flush, 2 when stall/flush.

Thus:

- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$

Average CPI for load

- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

Average CPI for branch

- And

- *Average CPI* =

Pipelined Performance Example: CPI

- Load/Branch CPI = 1 when no stall/flush, 2 when stall/flush.

Thus:

- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$

Average CPI for load

- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

Average CPI for branch

- And

- *Average CPI* = $(0.25)(1.4) +$
 $(0.1)(1) +$
 $(0.11)(1.25) +$
 $(0.02)(2) +$
 $(0.52)(1)$

load
store
beq
jump
r-type

= **1.15**

Pipelined Performance Example: Cycle Time

- There are 5 stages, and 5 different timing paths:

$$T_c = \max \{$$

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$$

$$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$$

$$t_{pcq} + t_{memwrite} + t_{setup}$$

$$2(t_{pcq} + t_{mux} + t_{RFwrite})$$

$$\}$$

fetch

decode

execute

memory

writeback

- The clock cycle *depends* on the *slowest stage*
- Decode and Writeback use register file and have only half a clock cycle to complete → that is why there is a 2 in front of them

Final Pipelined MIPS Processor (H&H)

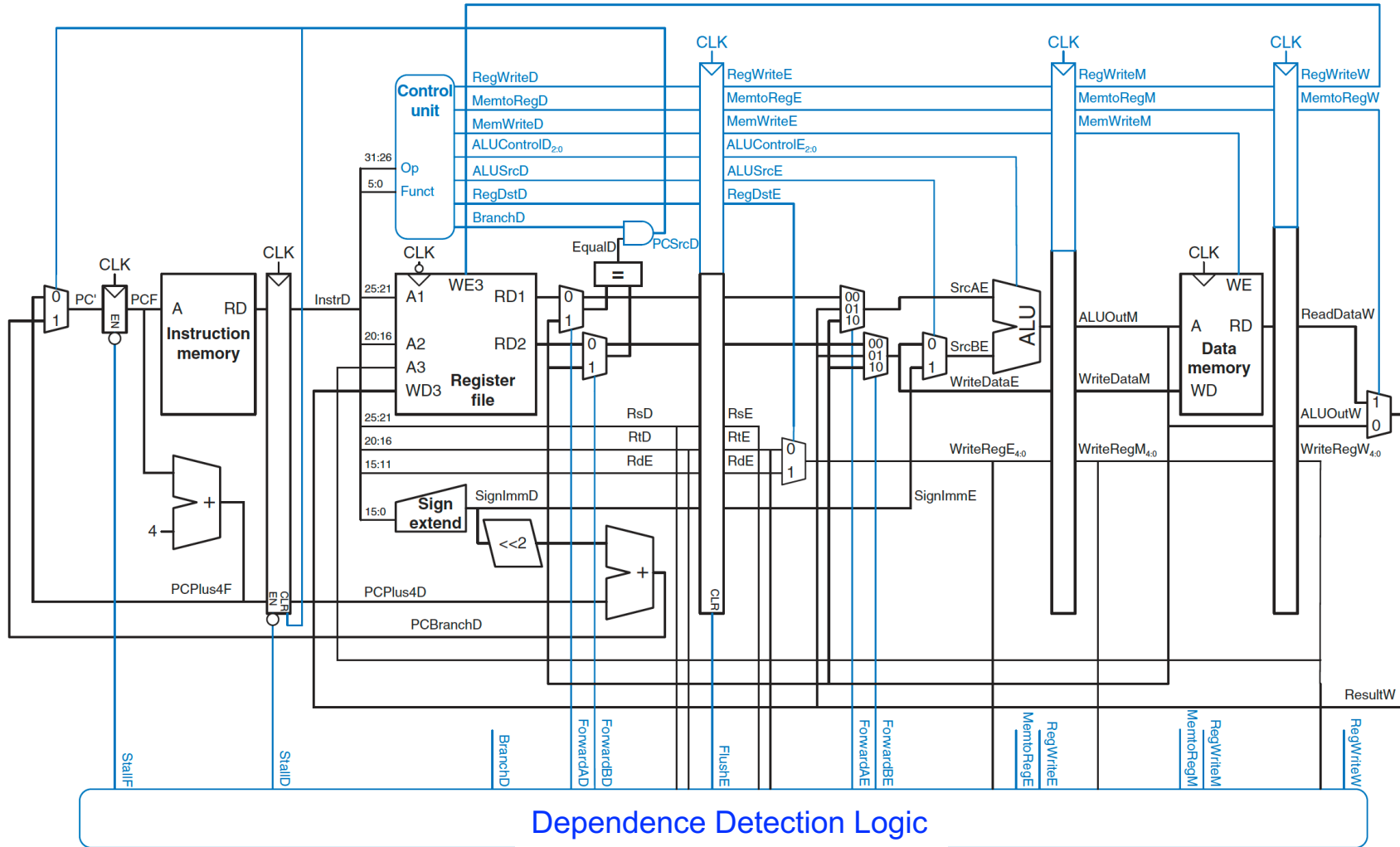


Figure 7.58 Pipelined processor with full hazard handling

Includes always-taken br prediction, early branch resolution, forwarding, stall logic

Pipelined Performance Example: Cycle Time

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	$T_{memwrite}$	220
Register file write	$t_{RFwrite}$	100

$$\begin{aligned}T_c &= 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \\&= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} \\&= 550 \text{ ps}\end{aligned}$$

Pipelined Performance Example: Exec Time

- For a program with 100 billion instructions executing on a pipelined MIPS processor:
 - $CPI = 1.15$
 - $T_c = 550 \text{ ps}$
- Execution Time $= (\# \text{ instructions}) \times CPI \times T_c$
 $= (100 \times 10^9)(1.15)(550 \times 10^{-12})$
 $= 63 \text{ seconds}$

Performance Summary for 3 MIPS microarch.

Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Multicycle	133	0.71
Pipelined	63	1.51

- **Pipelined implementation is the fastest of 3 implementations**
- **Even though we have a 5-stage pipeline, speedup is not 5X over the single-cycle or the multi-cycle system!**

Recall: 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
- 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: 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
- 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

Question to Ponder: Hardware vs. Software

- What is the role of the hardware vs. the software in data dependence handling?
 - Software based interlocking
 - Hardware based interlocking
 - Who inserts/manages the pipeline bubbles?
 - Who finds the independent instructions to fill “empty” pipeline slots?
 - What are the advantages/disadvantages of each?
 - Think of the performance equation as well

Question to Ponder: Hardware vs. Software

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
 - ❑ Software based instruction scheduling → static scheduling
 - ❑ Hardware based instruction scheduling → dynamic scheduling

- How does each impact different metrics?
 - ❑ Performance (and parts of the performance equation)
 - ❑ Complexity
 - ❑ Power consumption
 - ❑ Reliability
 - ❑ Cost
 - ❑ ...

More on Software vs. Hardware

- Software based scheduling of instructions → static scheduling
 - Hardware executes the **instructions in the compiler-dictated order**
 - Contrast this with **dynamic scheduling**: hardware can execute instructions out of the compiler-specified order
 - How does the compiler know the latency of each instruction?
- What information does the compiler not know that makes static scheduling difficult?
 - Answer: Anything that is determined at run time
 - **Variable-length operation latency, memory address, branch direction**
- How can the compiler alleviate this (i.e., estimate the unknown)?
 - Answer: Profiling (done statically or dynamically)

More on Static Instruction Scheduling

Trace Scheduling Idea

(a) (b) (c) (d)

TRACE SCHEDULING LOOP-FREE CODE

43

Panasonic

58:18 / 1:41:17

Lecture 16. Static Instruction Scheduling - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

7,136 views • Feb 26, 2015

46 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23K subscribers

Lecture 16: Static Instruction Scheduling
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Feb 23rd, 2015

Lecture 16 slides (pdf): <http://www.ece.cmu.edu/~ece447/s15/li...>

SUBSCRIBED



<https://www.youtube.com/watch?v=isBEVkljgGA&list=PL5PHm2jkkXmi5Cxxl7b3JCL1TWybTDtKq&index=18>

Lectures on Static Instruction Scheduling

- Computer Architecture, Spring 2015, Lecture 16

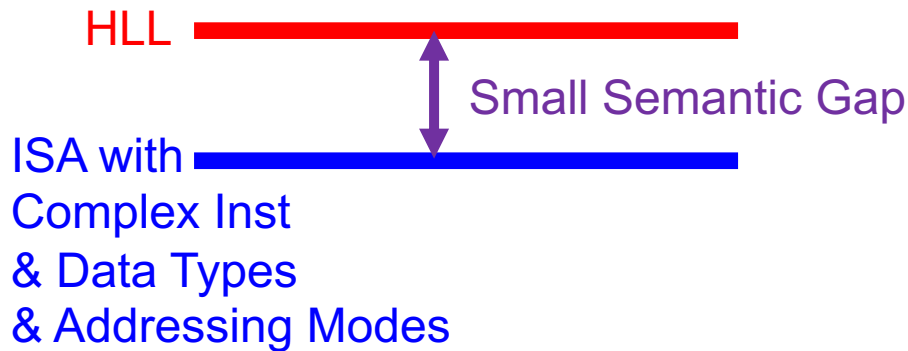
- Static Instruction Scheduling (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=isBEVkJgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18>

- Computer Architecture, Spring 2013, Lecture 21

- Static Instruction Scheduling (CMU, Spring 2013)
- <https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21>

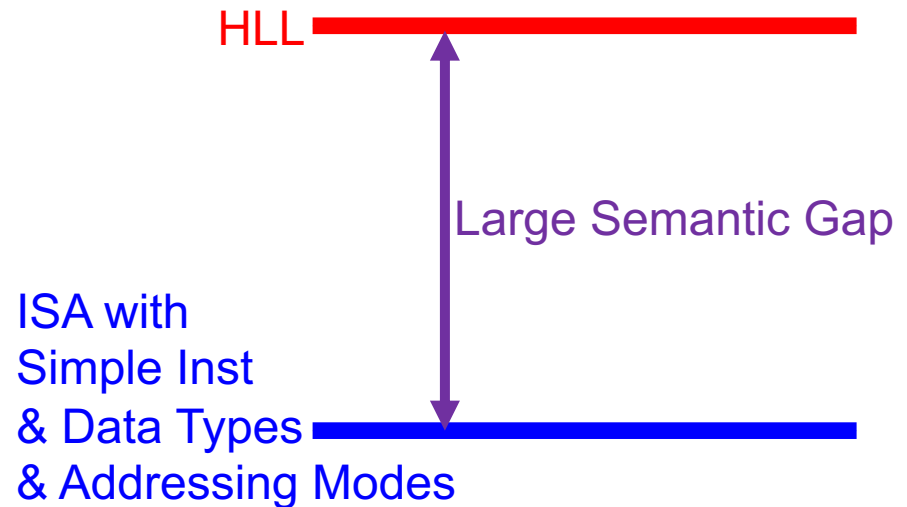
Recall: Semantic Gap

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



HW Control Signals

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

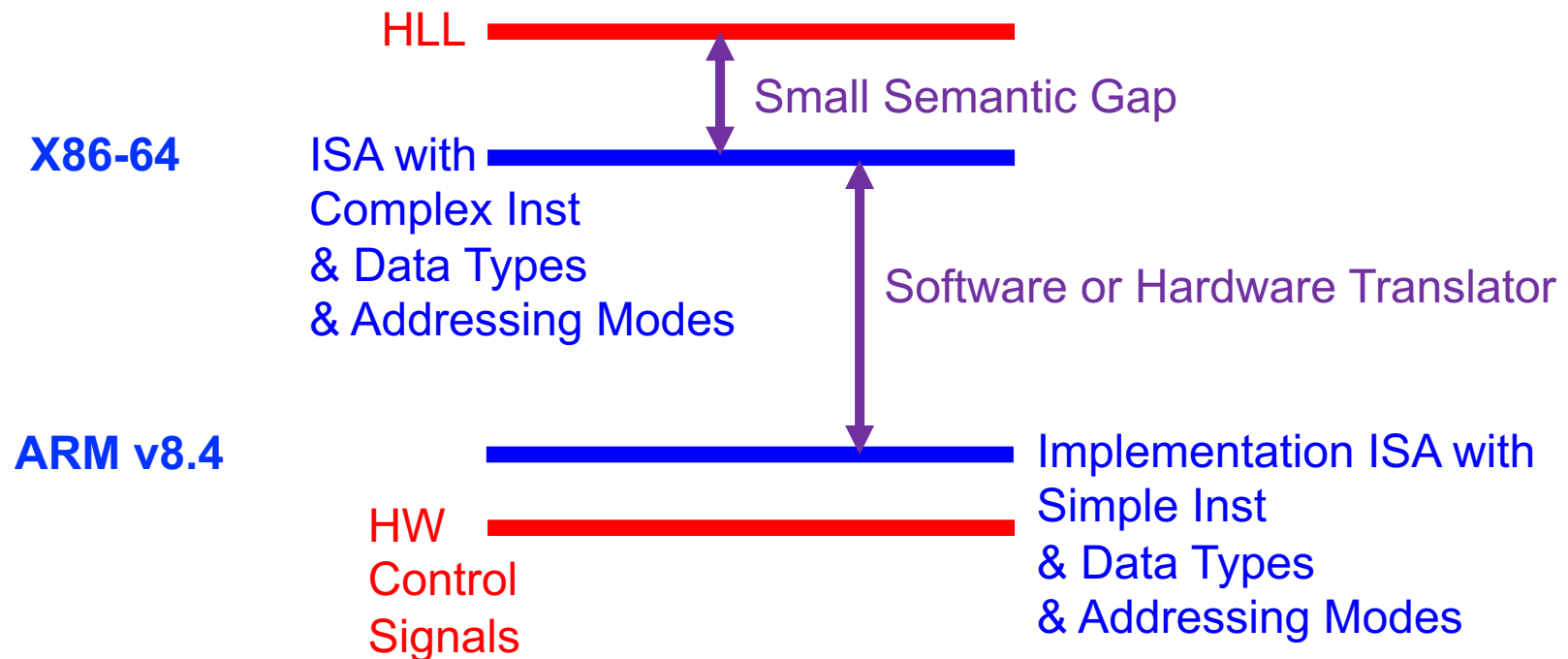


HW Control Signals

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

Recall: How to Change the Semantic Gap Tradeoffs

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



An Example: Rosetta 2 Binary Translator

Rosetta 2 [\[edit \]](#)

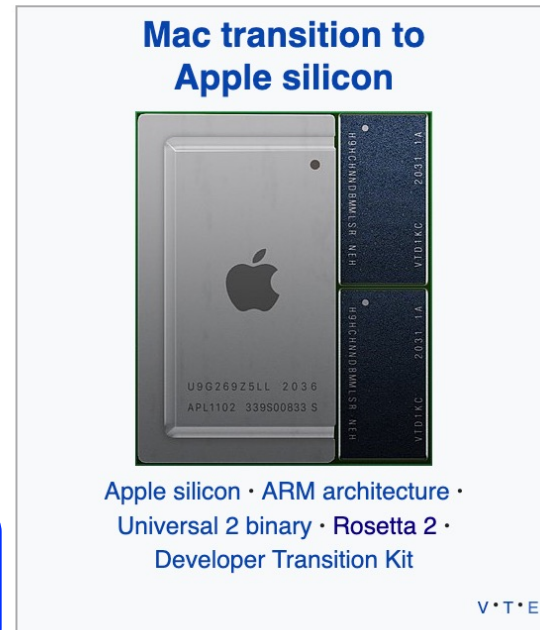
In 2020, Apple announced Rosetta 2 would be bundled with [macOS Big Sur](#), to aid in the [Mac transition to Apple silicon](#). The software permits many applications compiled exclusively for execution on [x86-64-based processors](#) to be translated for execution on Apple silicon.^{[2][8]}

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

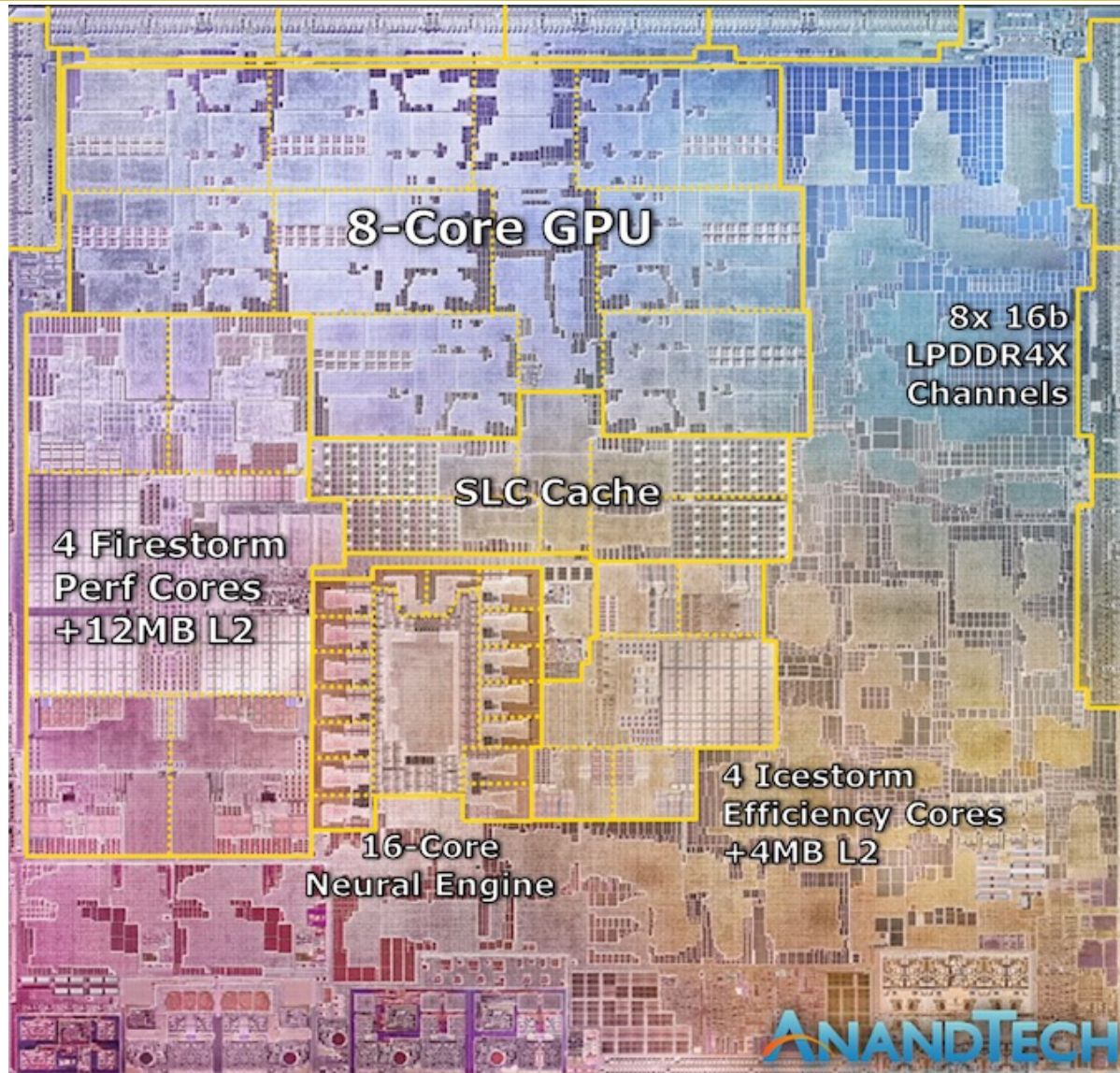
Rosetta 2's performance has been praised greatly.^{[10][11]} In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency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

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

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



An Example: Rosetta 2 Binary Translator



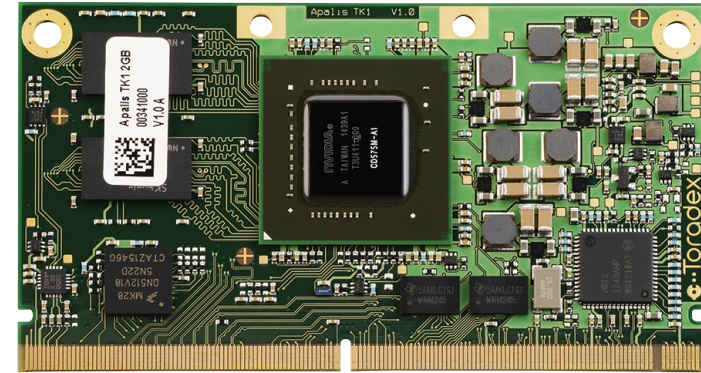
Apple M1,
2021

Another Example: NVIDIA Denver

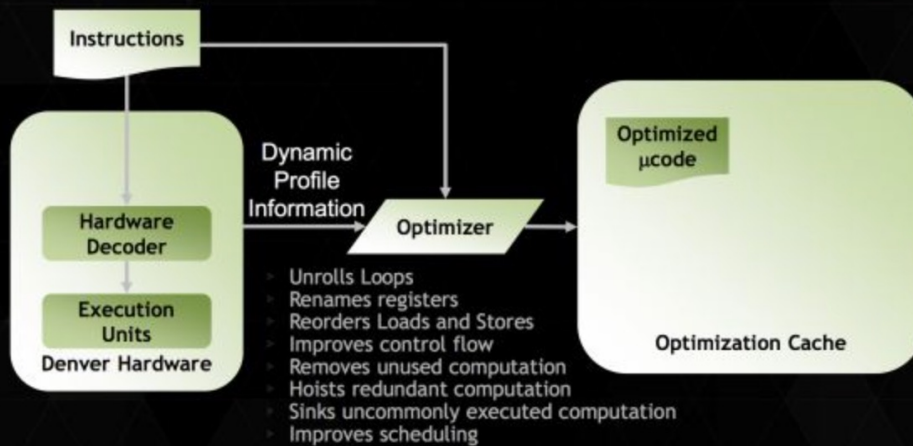
The Secret of Denver: Binary Translation & Code Optimization

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

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



DYNAMIC CODE OPTIMIZATION OPTIMIZE ONCE, USE MANY TIMES



The DCO system employed in the Denver CPU is codesigned software that extends ideas from prior system-level binary translators.² The primary function is to execute the user's code. The secondary function is to profile execution, create, optimize, and manage regions of tens to thousands of ARM instructions to form equivalent microcode-optimized regions that execute efficiently on the underlying microarchitecture.

More on NVIDIA Denver Code Optimizer

DENVER: NVIDIA'S FIRST 64-BIT ARM PROCESSOR

NVIDIA'S FIRST 64-BIT ARM PROCESSOR, CODE-NAMED DENVER, LEVERAGES A HOST OF NEW TECHNOLOGIES, SUCH AS DYNAMIC CODE OPTIMIZATION, TO ENABLE HIGH-PERFORMANCE MOBILE COMPUTING. IMPLEMENTED IN A 28-NM PROCESS, THE DENVER CPU CAN ATTAIN CLOCK SPEEDS OF UP TO 2.5 GHZ. THIS ARTICLE OUTLINES THE DENVER ARCHITECTURE, DESCRIBES ITS TECHNOLOGICAL INNOVATIONS, AND PROVIDES RELEVANT COMPARISONS AGAINST COMPETING MOBILE PROCESSORS.

Codesigning a hardware processor with a DCO software system creates both additional validation exposure and benefits. The DCO system can be upgraded in the field to address functionality, performance, or security issues.

The Denver hardware decoder provides a mechanism for periodically profiling recently taken branches. This branch history is moved into a shared buffer that can be processed from other cores, thereby minimizing the latency of the interruption. The DCO system will then run a thread that uses this profile to evaluate the dynamic properties of code executing and to assemble a picture of which code regions are hottest across all the processors. On finding sufficiently hot code, the DCO system will begin an optimization process to turn this input ARM code into a microcode execution region. The optimization process uses well-known traditional³ and more speculative compiler techniques to reduce work and increase efficiency of execution on the underlying skewed pipeline. To keep the latency of interruptions to a minimum, the optimizer thread is time-sliced with ARM execution (if any) and runs in a mode that can be quickly interrupted.

Transmeta: x86 to VLIW Translation

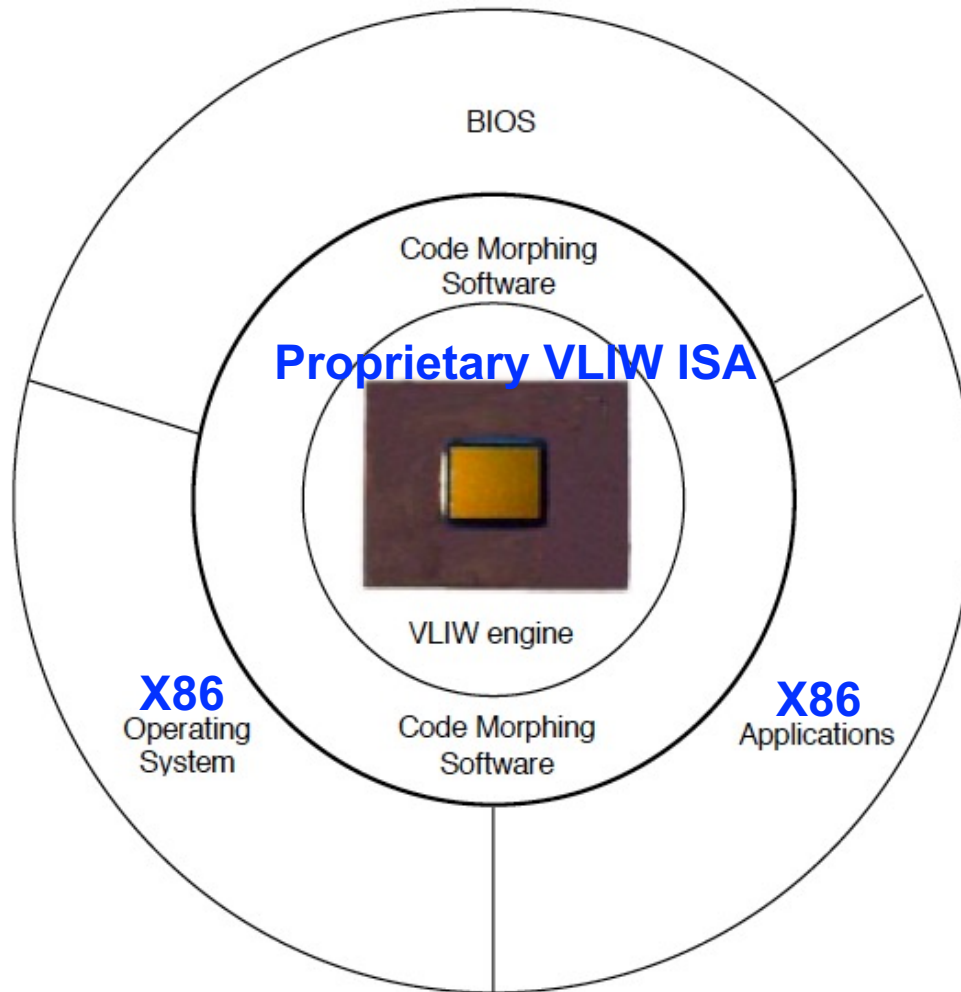


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, “[The Technology Behind Crusoe Processors](#),” Transmeta White Paper 2000.

More on Static Instruction Scheduling

Trace Scheduling Idea

(a) (b) (c) (d)

TRACE SCHEDULING LOOP-FREE CODE

43

Panasonic

58:18 / 1:41:17

Lecture 16. Static Instruction Scheduling - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

7,136 views • Feb 26, 2015

46 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23K subscribers

Lecture 16: Static Instruction Scheduling
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Feb 23rd, 2015

Lecture 16 slides (pdf): <http://www.ece.cmu.edu/~ece447/s15/li...>

SUBSCRIBED



<https://www.youtube.com/watch?v=isBEVkljgGA&list=PL5PHm2jkkXmi5Cxxl7b3JCL1TWybTDtKq&index=18>

Lectures on Static Instruction Scheduling

- Computer Architecture, Spring 2015, Lecture 16

- Static Instruction Scheduling (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=isBEVkJjgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18>

- Computer Architecture, Spring 2013, Lecture 21

- Static Instruction Scheduling (CMU, Spring 2013)
- <https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21>

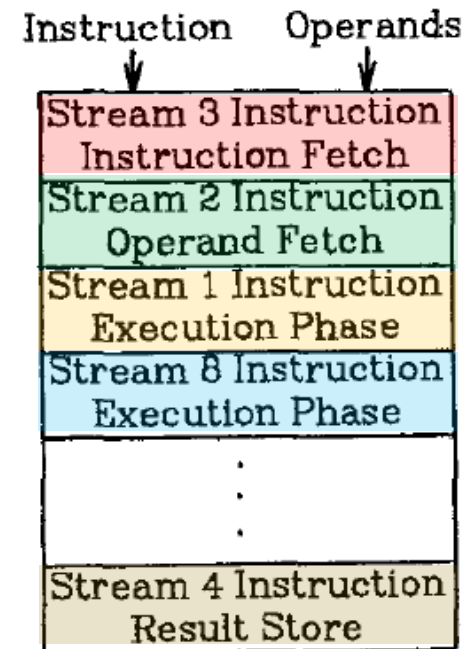
Recall: How to Handle Data Dependences

- 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
- 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

Fine-Grained Multithreading

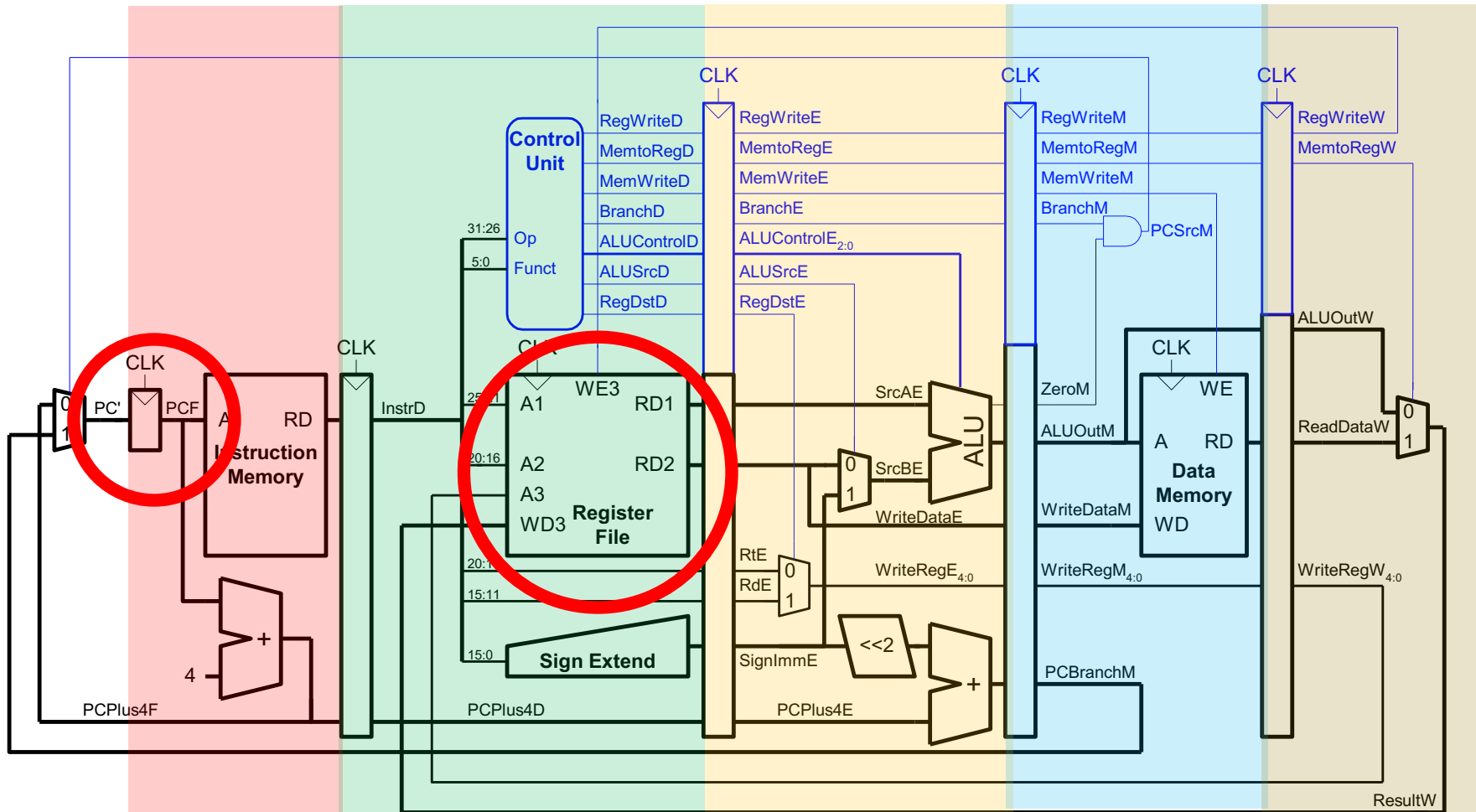
Fine-Grained Multithreading

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
 - ❑ Hardware has multiple thread contexts (PC+registers per thread)
 - ❑ Threads are completely independent
 - ❑ No instruction is fetched from the same thread until the prior branch/instruction from the thread completes
- + No logic needed for handling control and data dependences within a thread
- + High thread-level throughput
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Throughput loss when there are not enough threads to keep the pipeline full



Each pipeline stage has an instruction from a different, completely-independent thread

Fine-Grained Multithreading: Basic Idea



Each pipeline stage has an instruction from a different, completely-independent thread

We need a PC and a register file for each thread + muxes and control

Fine-Grained Multithreading (II)

- Idea: Fetch from a different thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates control and data dependence resolution latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Improves thread-level throughput but sacrifices per-thread throughput & latency
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

Fine-Grained Multithreading: History

- CDC 6600's peripheral processing unit is fine-grained multithreaded
 - Thornton, “[Parallel Operation in the Control Data 6600](#),” AFIPS 1964.
 - Processor executes a different I/O thread every cycle
 - An operation from the same thread is executed every 10 cycles

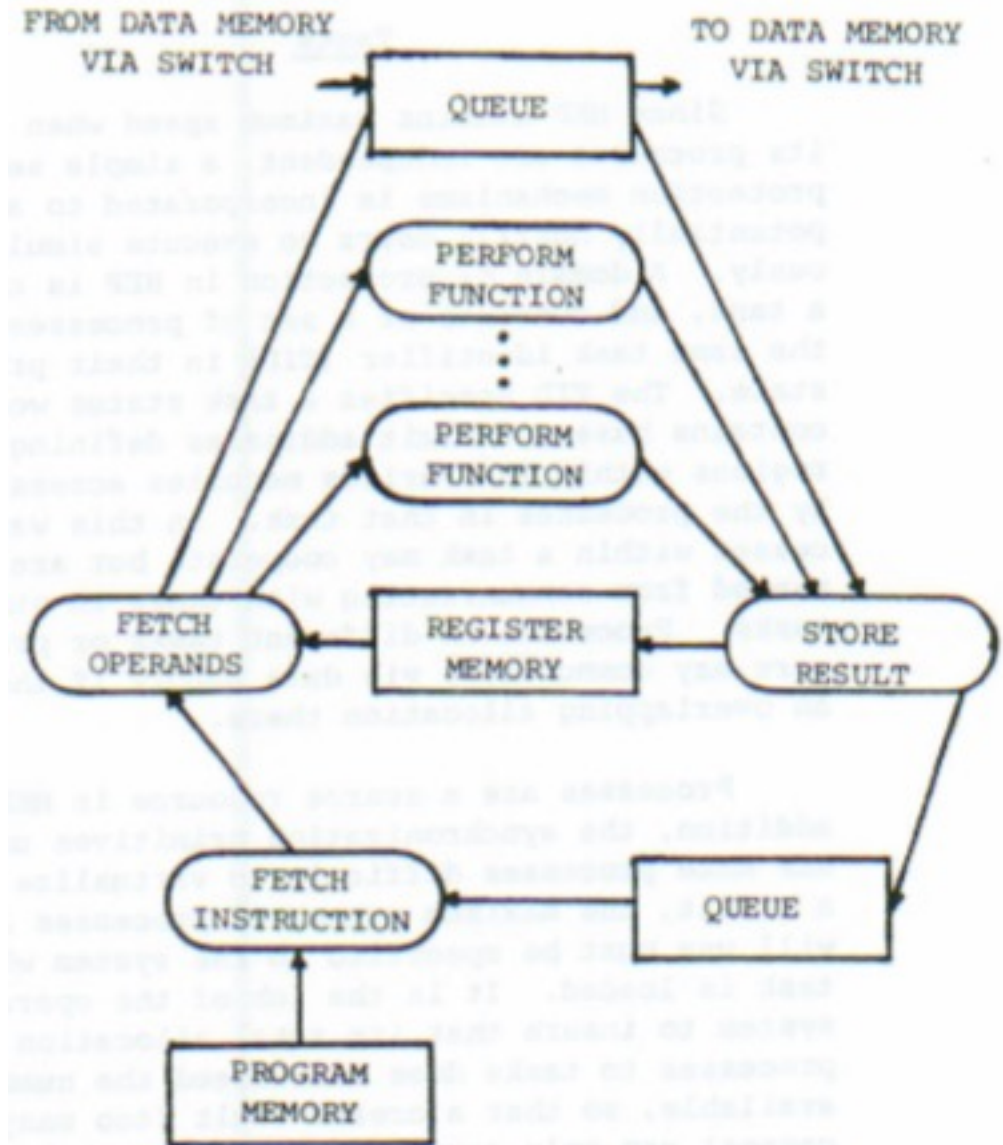
- Denelcor HEP (Heterogeneous Element Processor)
 - Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
 - 120 threads/processor
 - Available queue vs. unavailable (waiting) queue for threads
 - Each thread can have only 1 instruction in the processor pipeline
 - Each thread independent
 - To each thread, processor looks like a non-pipelined machine
 - [System throughput vs. single thread performance tradeoff](#)

Fine-Grained Multithreading in HEP

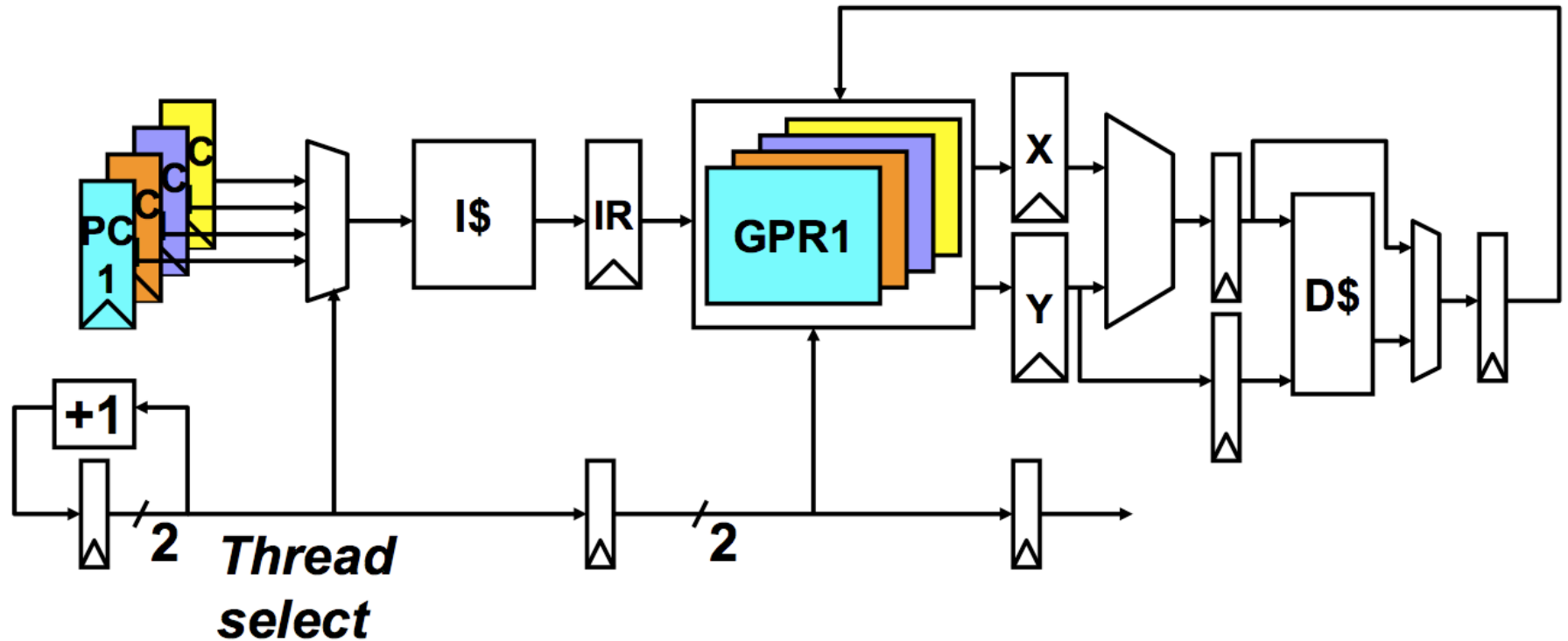
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
 - ❑ assuming no memory access
- No control and data dependence checking



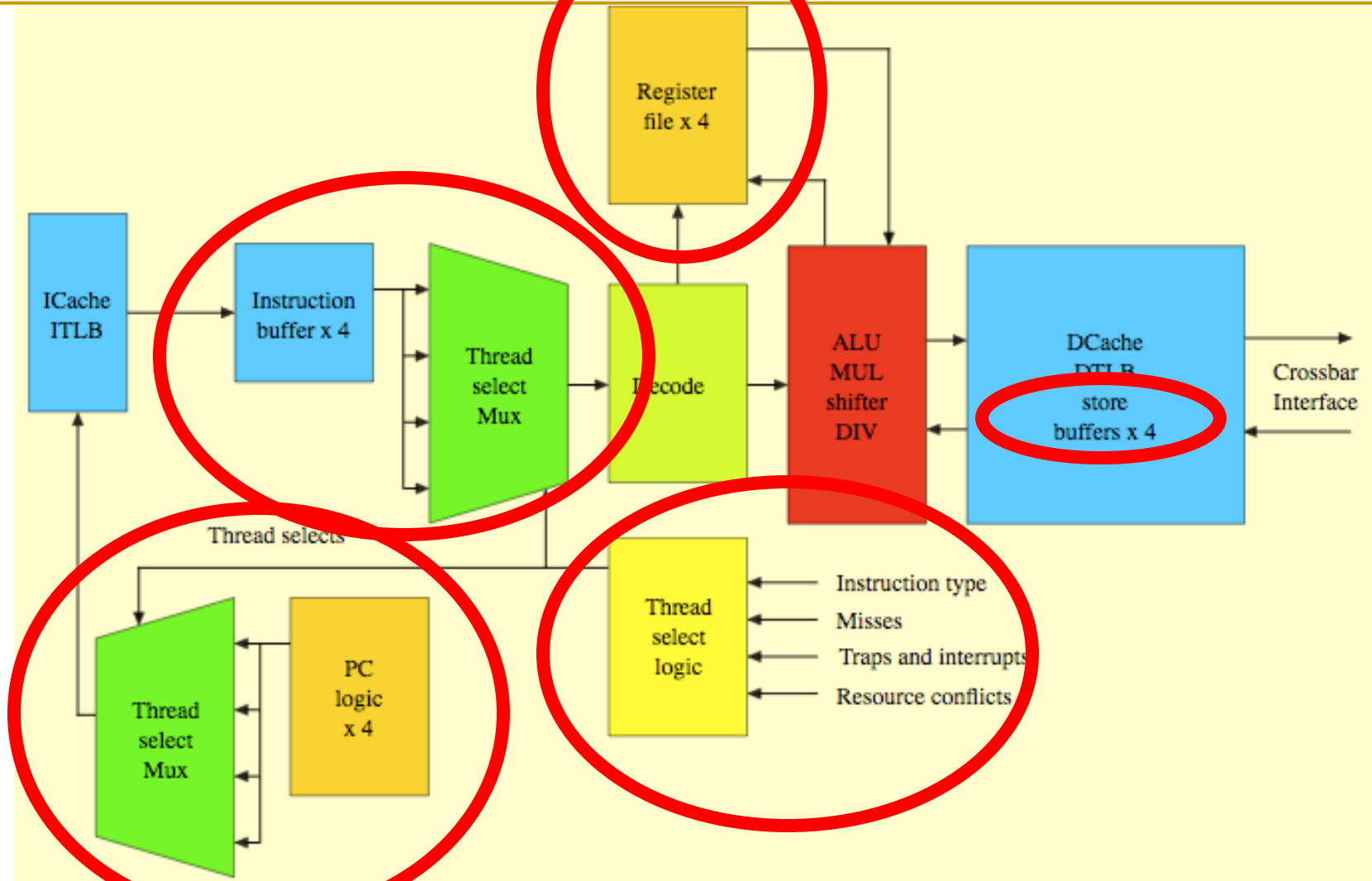
Burton Smith
(1941-2018)



Multithreaded Pipeline Example



Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Fine-Grained Multithreading

■ Advantages

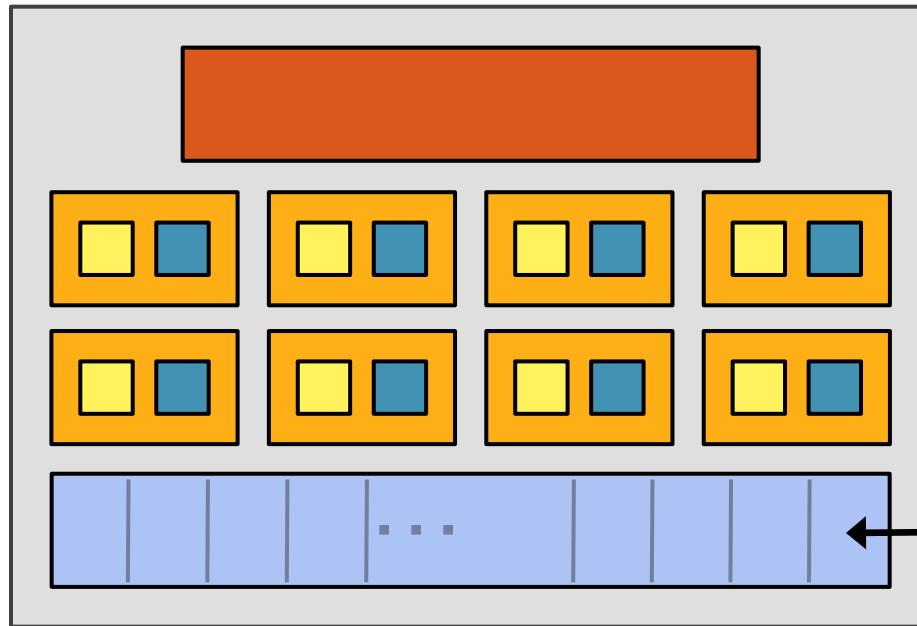
- + No need for dependence checking between instructions
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, pipeline utilization

■ Disadvantages

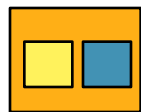
- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles from the same thread)
- Resource contention between threads in caches and memory
- Dependence checking logic *between* threads may be needed (load/store)

Modern GPUs are FGMT Machines

NVIDIA GeForce GTX 285 “core”



64 KB of storage
for thread contexts
(registers)



= data-parallel (SIMD) func. unit,
control shared across 8 units



= multiply-add



= multiply

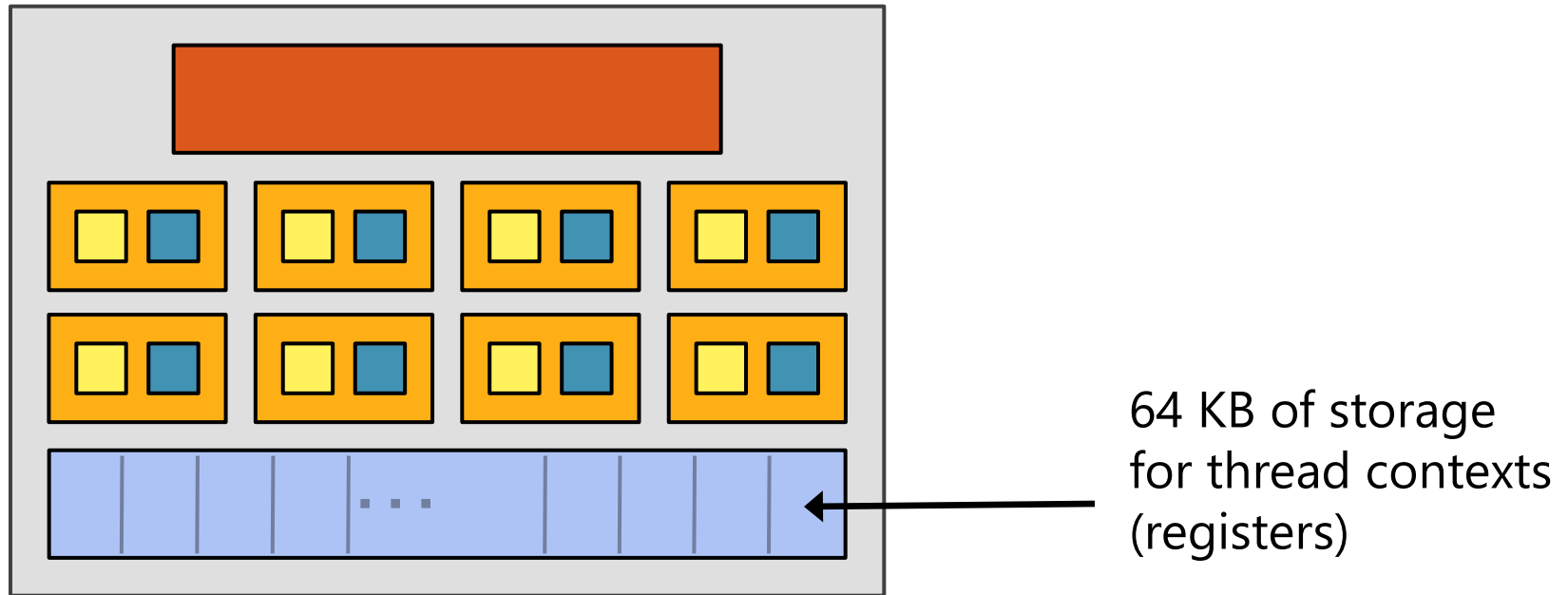


= instruction stream decode



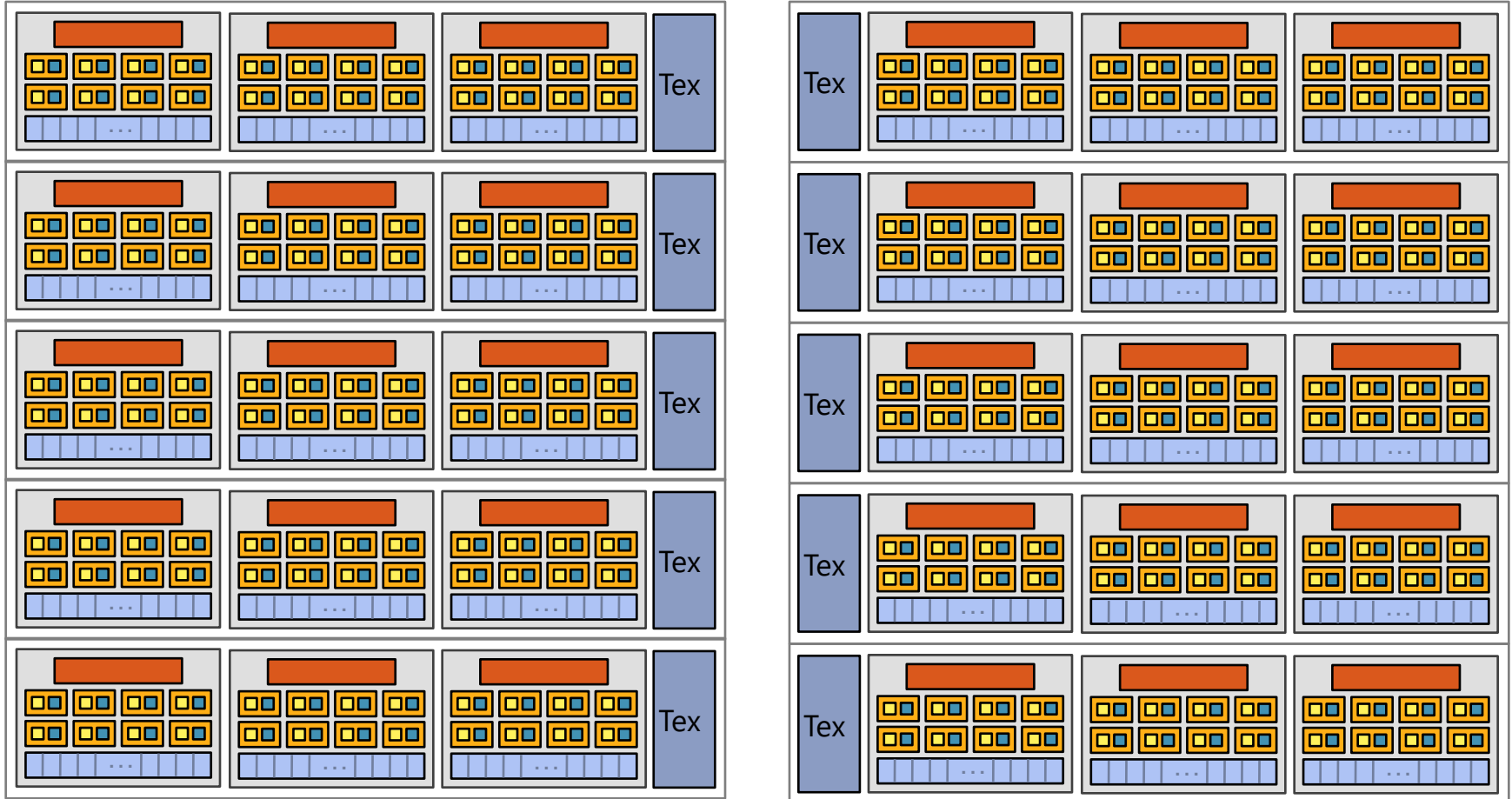
= execution context storage

NVIDIA GeForce GTX 285 “core”



- Groups of 32 **threads** share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285 (~2009)



30 cores on the GTX 285: 30,720 threads

Further Reading for the Interested (I)

A PIPELINED, SHARED RESOURCE MIMD COMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205



Burton Smith
(1941-2018)

Architecture and applications of the HEP multiprocessor computer system

Burton J. Smith
Denelcor, Inc., 14221 E. 4th Avenue, Aurora, Colorado 80011

Further Reading for the Interested (II)

The Tera Computer System*

Robert Alverson

David Callahan
Allan Porterfield

Daniel Cummings
Burton Smith

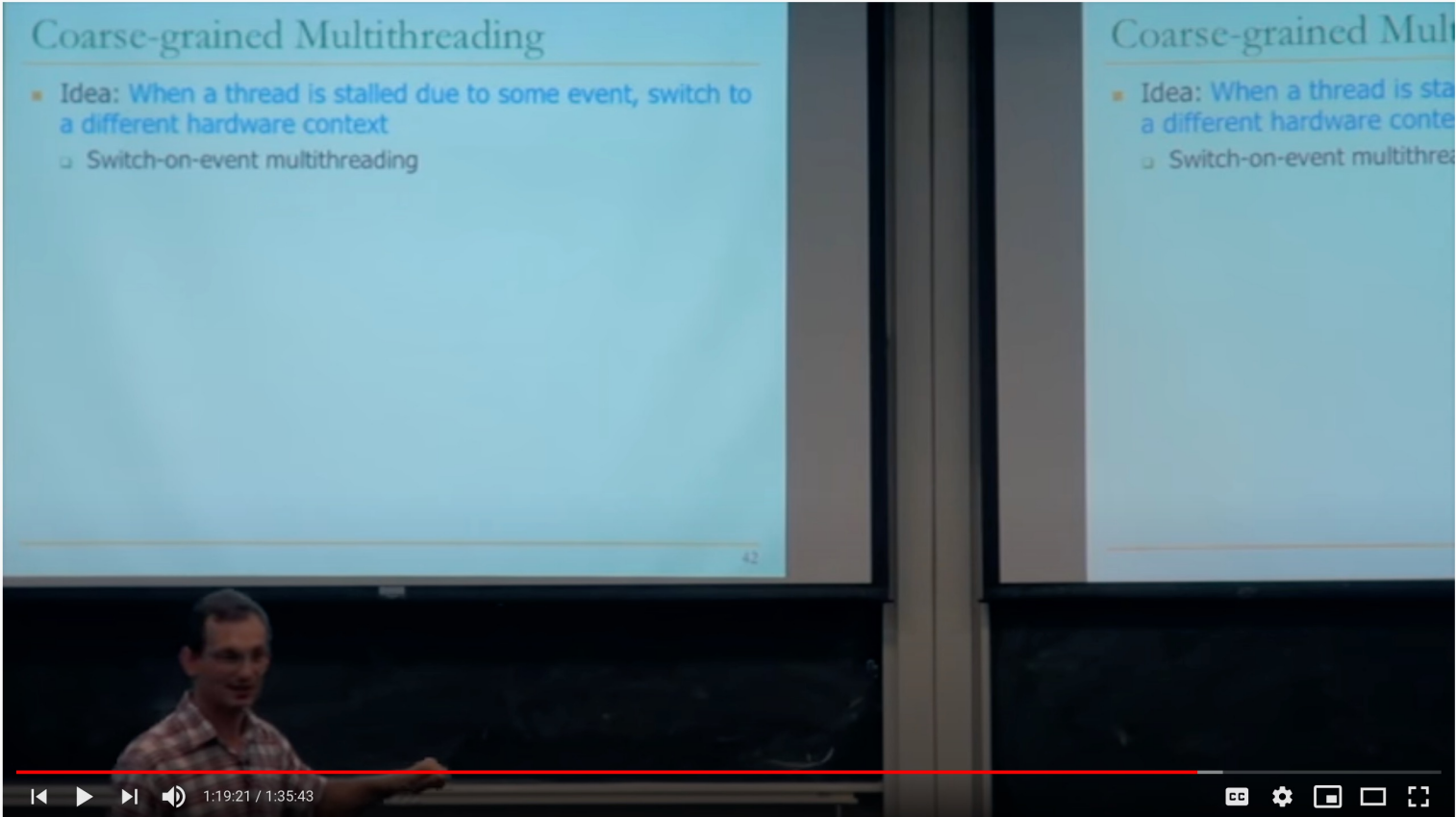
Brian Koblenz

Tera Computer Company
Seattle, Washington USA

4 Processors

Each processor in a Tera computer can execute multiple instruction streams simultaneously. In the current implementation, as few as one or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different stream may be issued in each tick without interfering with its predecessors. When an instruction finishes, the stream to which it belongs thereby becomes ready to execute the next instruction. As long as there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is being fully utilized. Thus, it is only necessary to have enough streams to hide the expected latency (perhaps 70 ticks on average); once latency is hidden the processor is running at peak performance and additional streams do not speed the result.

More on Multithreading (I)



The video player shows a lecture slide titled "Coarse-grained Multithreading". The slide content is as follows:


- Idea: When a thread is stalled due to some event, switch to a different hardware context
 - Switch-on-event multithreading

The video player interface includes a progress bar at 1:19:21 / 1:35:43, a like count of 10, and a comment count of 0. The video is titled "Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 9 - Multithreading" and has 1,252 views as of Nov 19, 2013. The channel is "Carnegie Mellon Computer Architecture" with 23K subscribers. The video description includes the lecture title, lecturer (Prof. Onur Mutlu), and date (September 26, 2013).

Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 9 - Multithreading

1,252 views · Nov 19, 2013

10 0 SHARE SAVE ...

 **Carnegie Mellon Computer Architecture**
23K subscribers

Lecture 9: Multithreading
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: September 26, 2013.

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=iqi9wFqFiNU&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=51

More on Multithreading (II)

Intel Pentium 4 Hyperthreading

- latency load handling
- multi-level scheduling window
- partitioned structures
 - Instruction Queues
 - Store buffer
 - Reorder buffer
- 5% area overhead due to SMT

area overhead due to SMT

Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal 2002.

Intel Pentium 4 Hyperthreading

- Long latency load handling
 - Multi-level scheduling window
- More partitioned structures
 - I-TLB
 - Instruction Queues
 - Store buffer
 - Reorder buffer
- 5% area overhead due to SMT
- Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal 2002.

Carnegie Mellon -Parallel Computer Architecture 2012 - Onur Mutlu - Lecture 10 - Multithreading II

1,594 views • Sep 21, 2013

11 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 10: Multithreading II

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 28, 2012.

SUBSCRIBED



More on Multithreading (III)

g (Tandem, Compaq Himalaya)

Microprocessor R1 ← (R2) and R2 ← (R1)

Input Register and Output Register

Memory covered by ECC
RAID array covered by parity
Servernet covered by CRC

the processor, compare the results of two
re committing an instruction

Lockstepping (Tandem, Compaq Himalaya)

Microprocessor R1 ← (R2) and R2 ← (R1)

Input Register and Output Register

Memory covered by ECC
RAID array covered by parity
Servernet covered by CRC

Idea: Replicate the processor, compare the results of two
processors before committing an instruction

Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 13-Multi-threading II

1,132 views • Sep 21, 2013

8 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 13: Multi-threading III

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: October 5, 2012.

SUBSCRIBED



https://www.youtube.com/watch?v=7vkDpZ1-hHM&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=53

More on Multithreading (IV)



Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 15 - Speculation 1

915 views • Sep 21, 2013

9 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 15: Speculation I

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: October 10, 2012.

SUBSCRIBED



https://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=54

Lectures on Multithreading

■ Parallel Computer Architecture, Fall 2012, Lecture 9

- ❑ Multithreading I (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=iqi9wFqFiNU&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=51

■ Parallel Computer Architecture, Fall 2012, Lecture 10

- ❑ Multithreading II (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=e8lfl6MbILg&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=52

■ Parallel Computer Architecture, Fall 2012, Lecture 13

- ❑ Multithreading III (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=7vkDpZ1-hHM&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=53

■ Parallel Computer Architecture, Fall 2012, Lecture 15

- ❑ Speculation I (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnnyV6D&index=54

Digital Design & Computer Arch.

Lecture 14: Pipelined Processor Design

Prof. Onur Mutlu

ETH Zürich

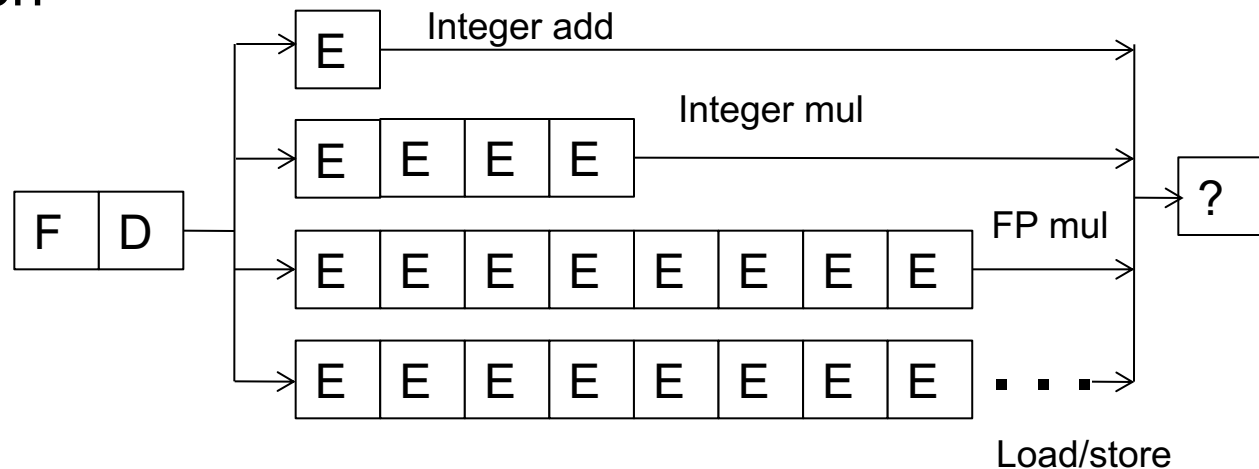
Spring 2022

8 April 2022

Pipelining and Precise Exceptions: Preserving Sequential Semantics

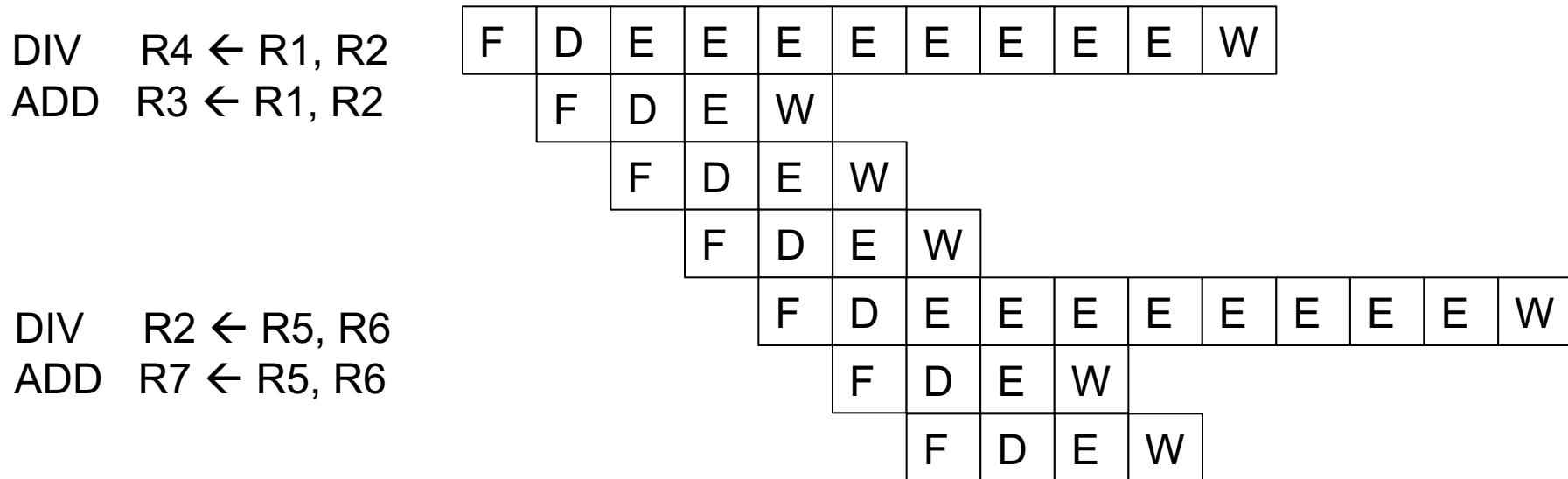
Multi-Cycle Execution

- Not all instructions take the same amount of time for “execution”
- 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?

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
 - ❑ return back to program execution (if possible and makes sense)

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

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 boundaries == Cycle boundaries
- 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

Precise Exceptions in Multi-Cycle FSM

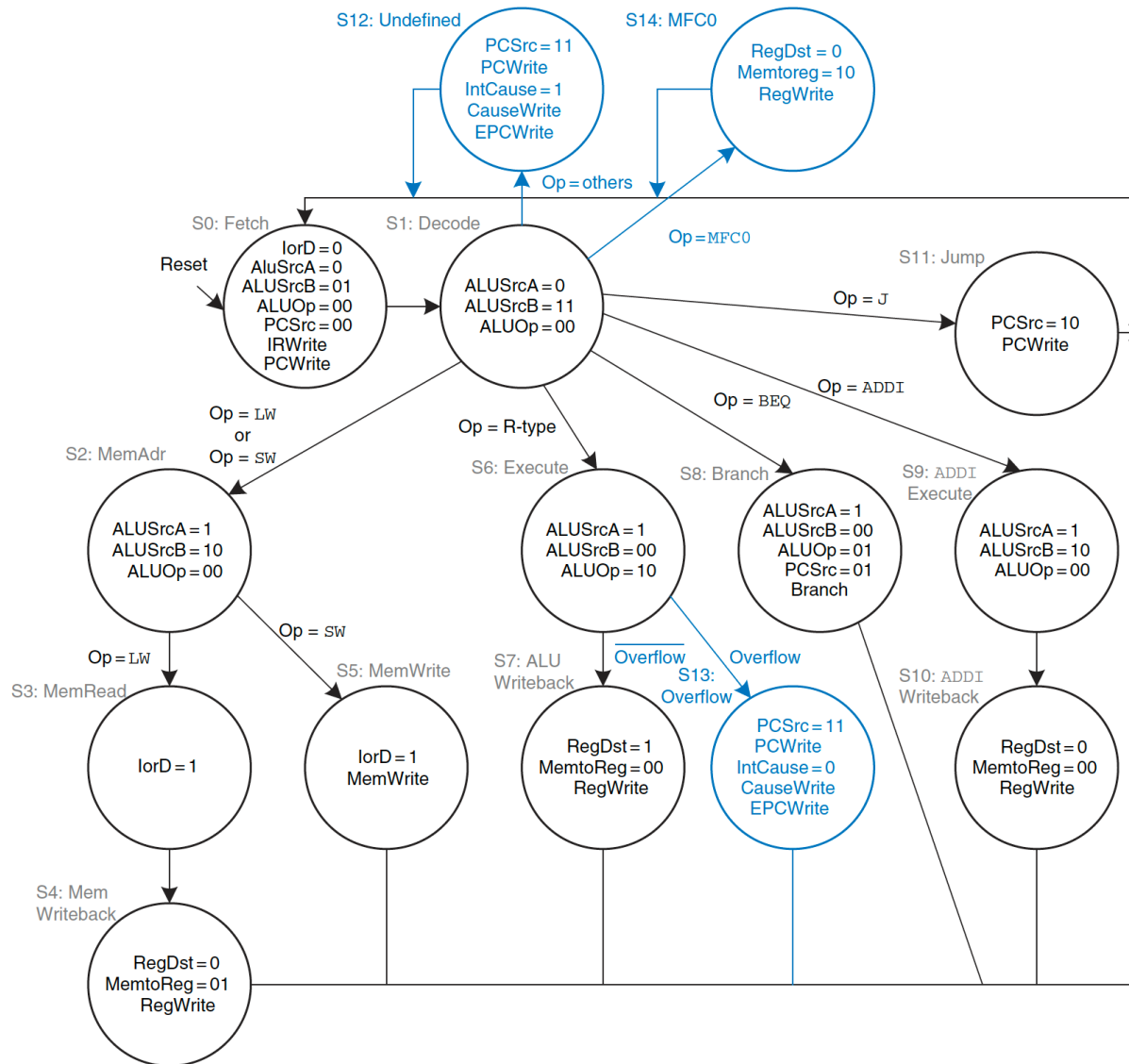


Figure 7.64 Controller supporting exceptions and mfc0

Precise Exceptions in Multi-Cycle Datapath

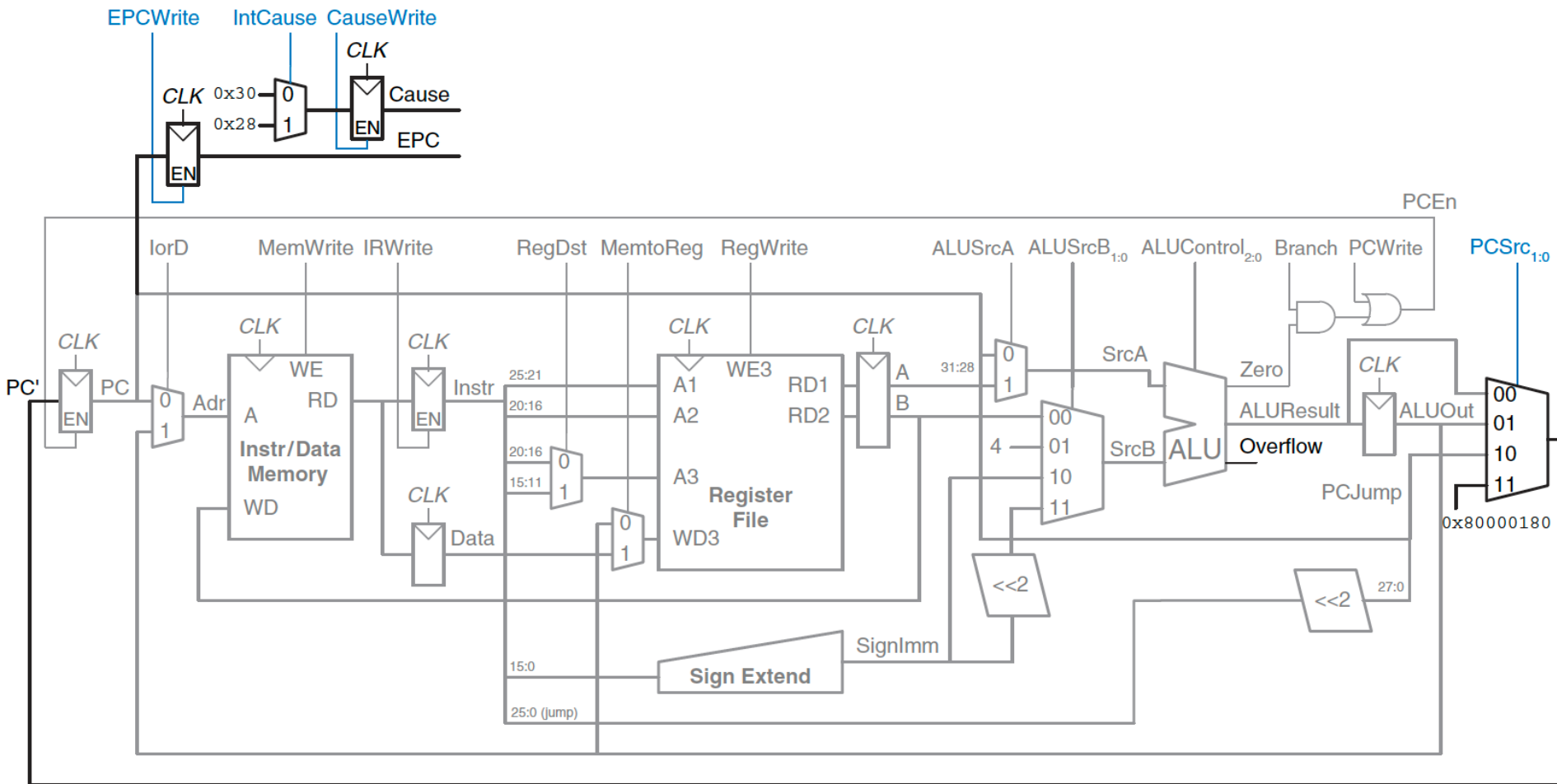
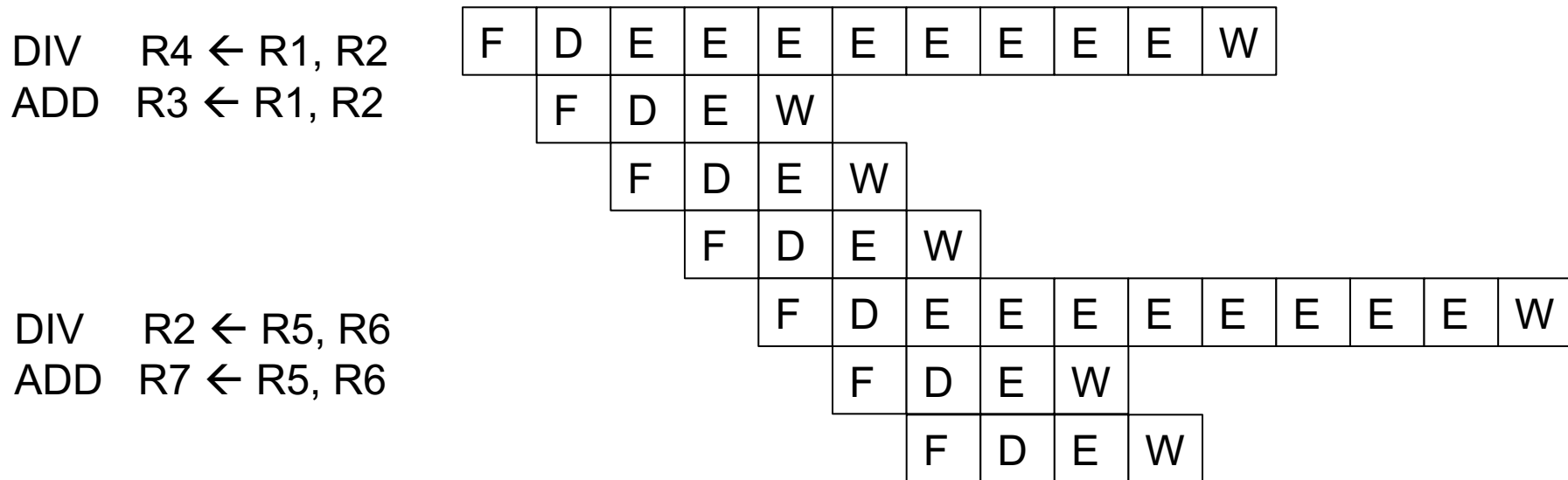


Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

Multi-Cycle Execute: More Complications

- 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?

Ensuring Precise Exceptions in Pipelining

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

DIV $R3 \leftarrow R1, R2$
ADD $R4 \leftarrow R1, R2$

F	D	E	E	E	E	E	E	E	E	W									
	F	D	E	E	E	E	E	E	E	E	W								
		F	D	E	E	E	E	E	E	E	E	W							
			F	D	E	E	E	E	E	E	E	E	W						
				F	D	E	E	E	E	E	E	E	E	W					
					F	D	E	E	E	E	E	E	E	E	W				
						F	D	E	E	E	E	E	E	E	E	W			
							F	D	E	E	E	E	E	E	E	E	W		
								F	D	E	E	E	E	E	E	E	E	W	

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

Solutions

- Reorder buffer

- History buffer

- Future register file

- Checkpointing

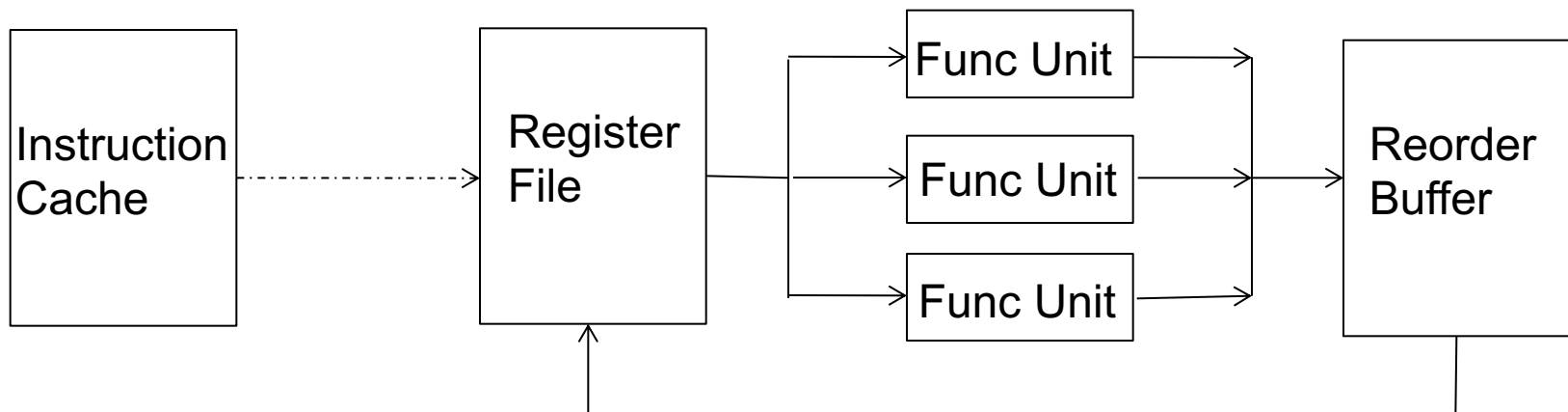
We will not cover these
See suggested lecture videos from Spring 2015

- Suggested reading

- 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 the 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



Reorder Buffer

- Buffers information about **all instructions** that are **decoded** but **not yet retired**/committed

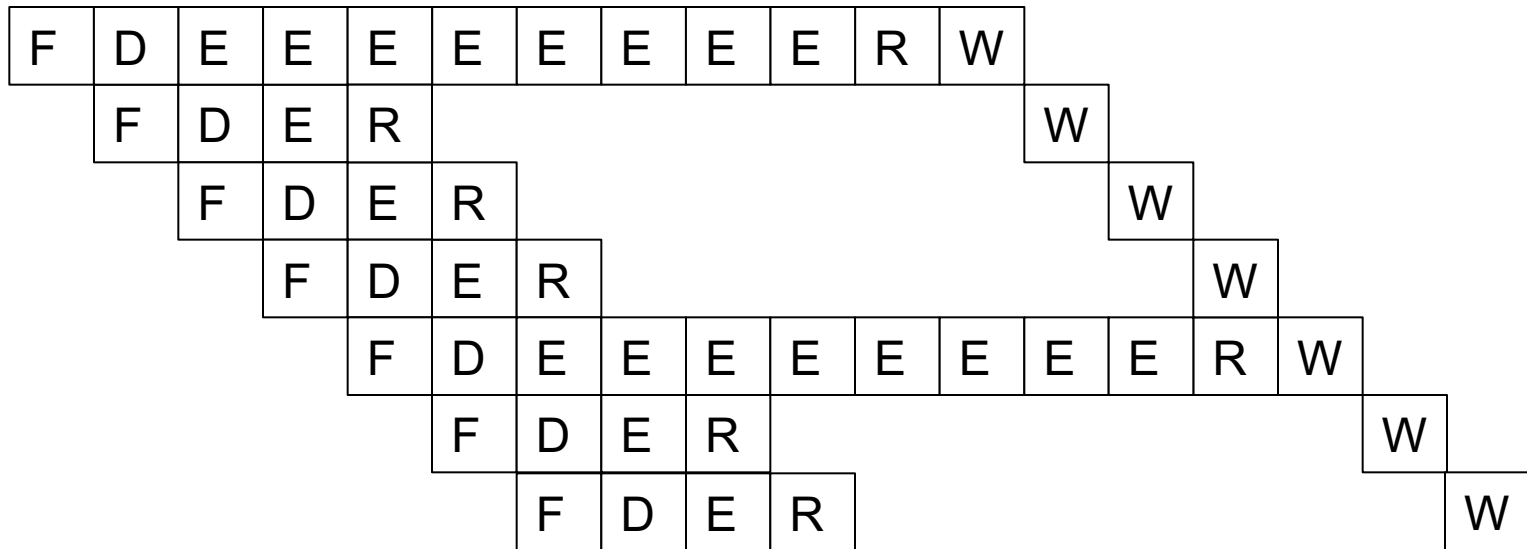
What's in a ROB Entry?

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

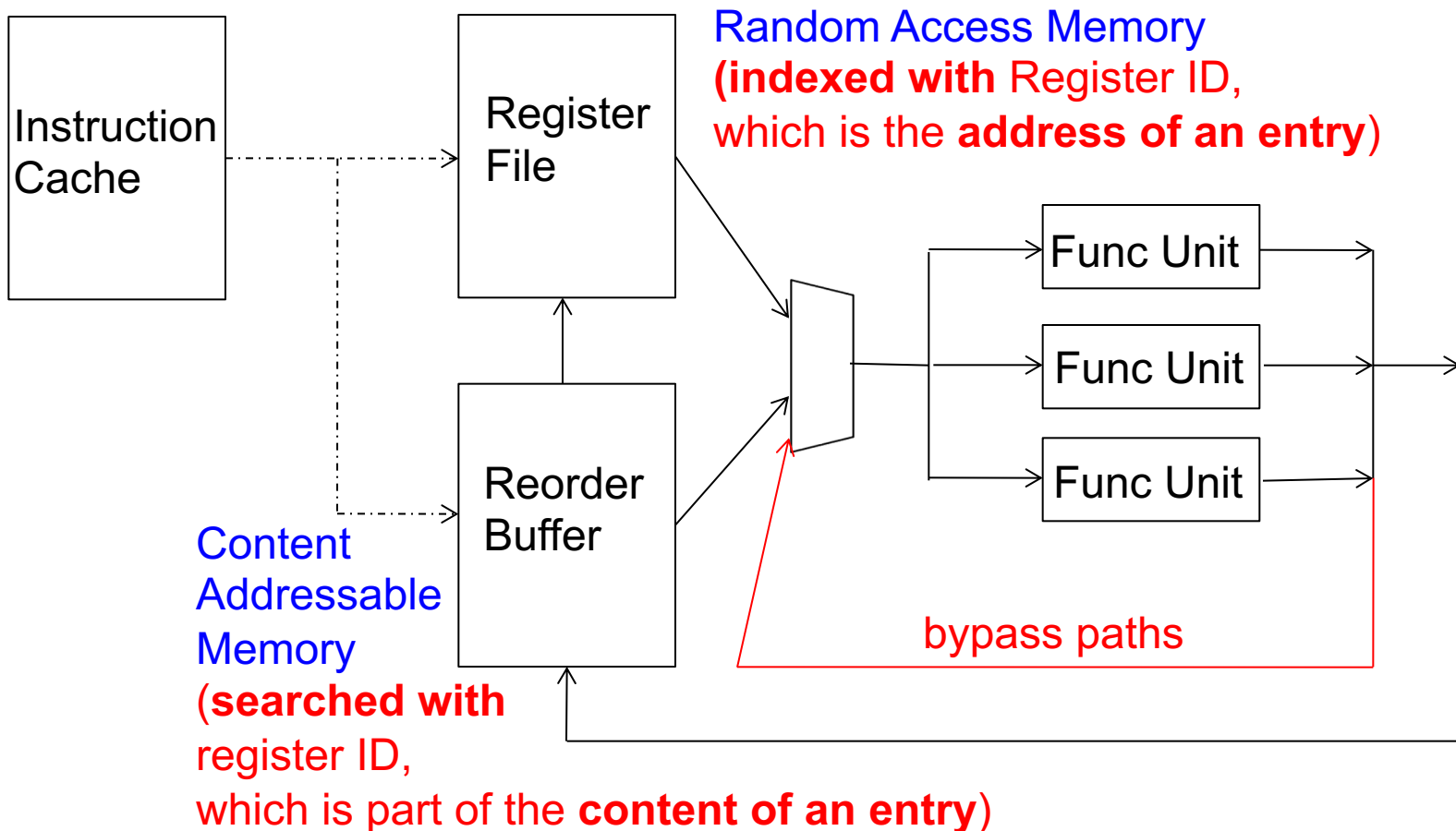
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- 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?

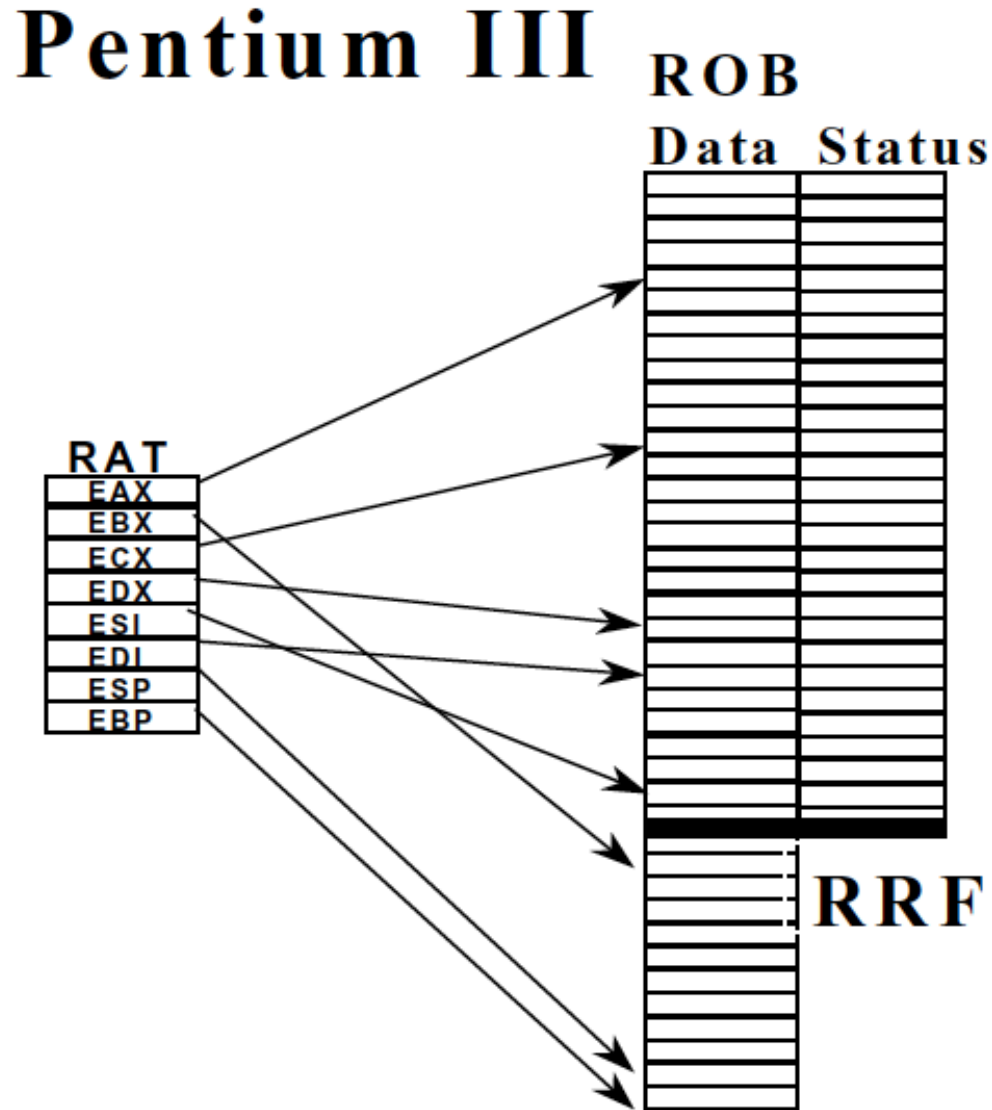
- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



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 in Intel Pentium III



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


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

True (flow) dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW) -- **True**

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR) -- **Anti**

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



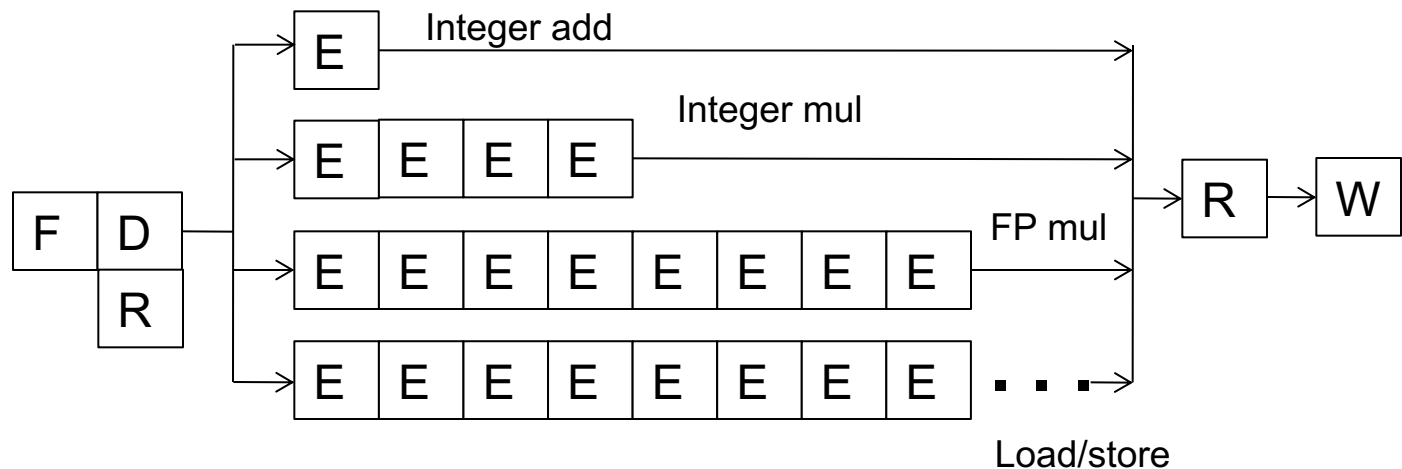
Write-after-Write
(WAW) -- **Output**

Renaming Example

- 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 R7, R8 → R12

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 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**



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

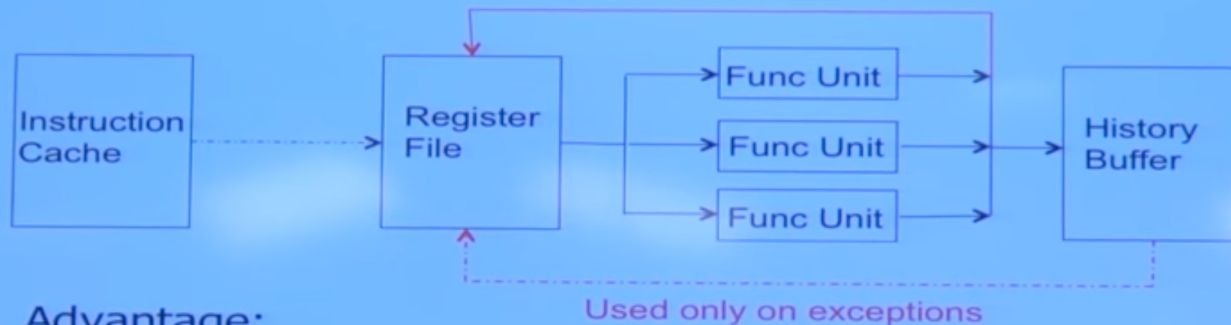
■ Other solutions aim to eliminate the disadvantages

- ❑ History buffer
- ❑ Future file
- ❑ Checkpointing

We will not cover these
See suggested lecture videos from Spring 2015

More on State Maintenance & Precise Exceptions

History Buffer



■ Advantage:

- Register file contains up-to-date values for incoming instructions
→ History buffer access not on critical path

■ Disadvantage:

- Need to read the old value of the destination register
- Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

29

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

11,990 views • Feb 12, 2015

77 0 SHARE SAVE ...



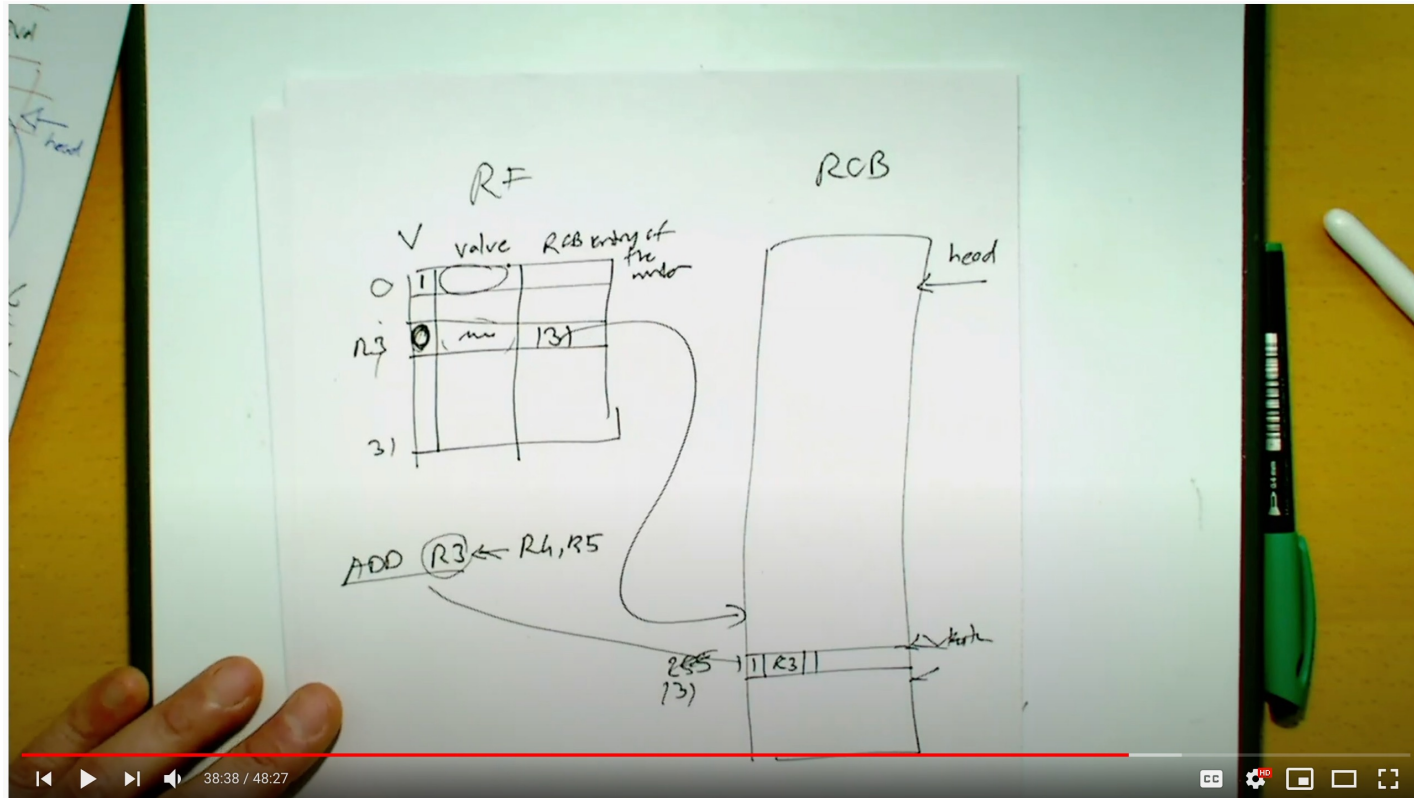
Carnegie Mellon Computer Architecture
23K subscribers

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

ANALYTICS

EDIT VIDEO

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

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

ANALYTICS

EDIT VIDEO

Lectures on State Maintenance & Recovery

- **Computer Architecture, Spring 2015, Lecture 11**
 - Precise Exceptions, State Maintenance/Recovery (CMU, Spring 2015)
 - <https://www.youtube.com/watch?v=nMfbtzWizDA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=13>
- **Digital Design & Computer Architecture, Spring 2019, Lecture 15a**
 - Reorder Buffer (ETH Zurich, Spring 2019)
 - <https://www.youtube.com/watch?v=9yo3yhUijQs&list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&index=17>

Suggested Readings for the Interested

- Smith and Plezskun, “**Implementing Precise Interrupts in Pipelined Processors**,” IEEE Trans on Computers 1988 and ISCA 1985.
- Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proceedings of the IEEE, 1995
- Hwu and Patt, “**Checkpoint Repair for Out-of-order Execution Machines**,” ISCA 1987.
- Backup Slides

Backup Slides on Precise Exceptions

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

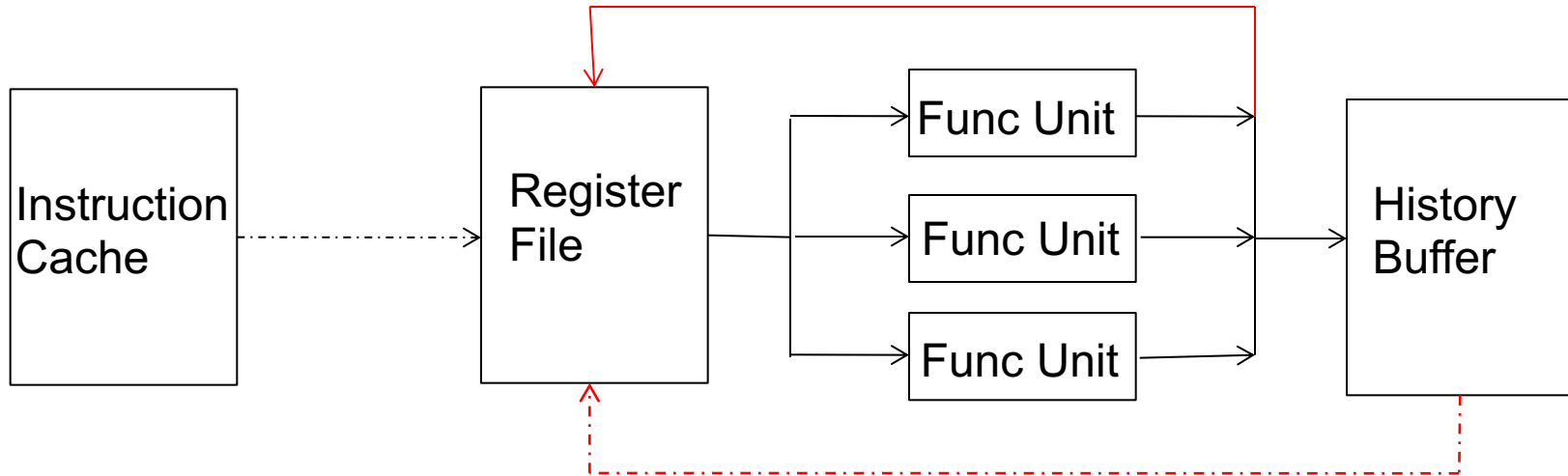
■ Other solutions aim to eliminate the disadvantages

- ❑ History buffer
- ❑ Future file
- ❑ Checkpointing

Solution II: History Buffer (HB)

- Idea: Update the register file when instruction completes, but UNDO UPDATES when an exception occurs
- When instruction is decoded, it reserves an HB entry
- When the instruction completes, it stores the old value of its destination in the HB
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head

History Buffer



Used only on exceptions

■ Advantage:

- ❑ Register file contains up-to-date values for incoming instructions
→ History buffer access not on critical path

■ Disadvantage:

- ❑ Need to read the old value of the destination register
- ❑ Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

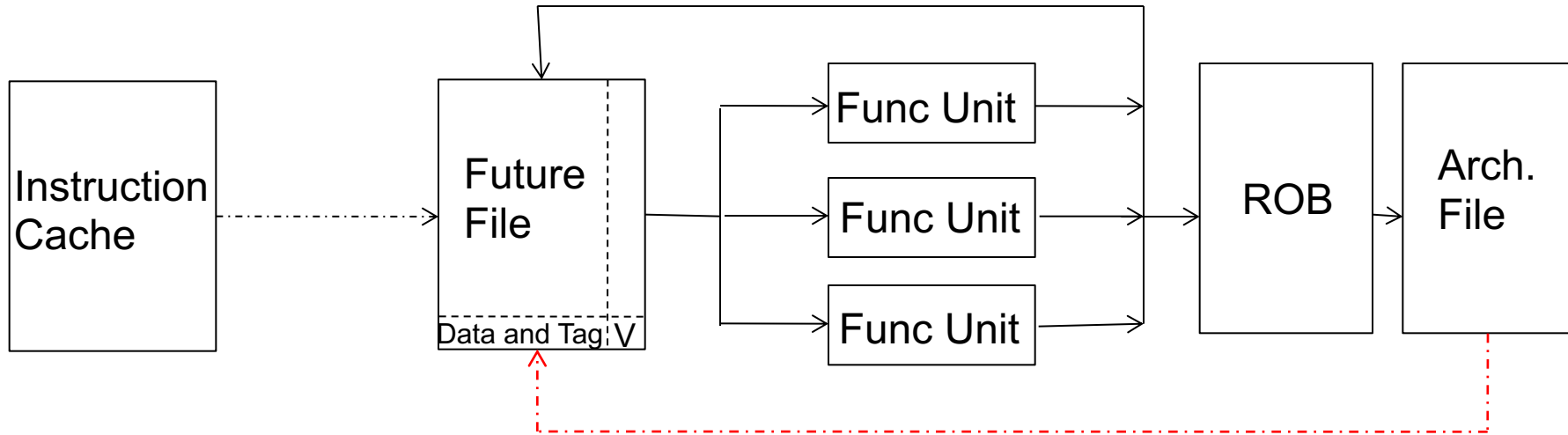
Comparison of Two Approaches

- Reorder buffer
 - ❑ Pessimistic register file update
 - ❑ Update only with non-speculative values (in program order)
 - ❑ Leads to complexity/delay in accessing the new values
- History buffer
 - ❑ Optimistic register file update
 - ❑ Update immediately, but log the old value for recovery
 - ❑ Leads to complexity/delay in logging old values
- Can we get the best of both worlds?
 - ❑ Principle: Heterogeneity
 - ❑ Idea: Have both types of register files

Solution III: Future File (FF) + ROB

- Idea: Keep two register files (speculative and architectural)
 - Arch reg file: Updated in program order for precise exceptions
 - Use a reorder buffer to ensure in-order updates
 - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)
- Future file is used for fast access to latest register values (speculative state)
 - Frontend register file
- Architectural file is used for state recovery on exceptions (architectural state)
 - Backend register file

Future File



■ Advantage

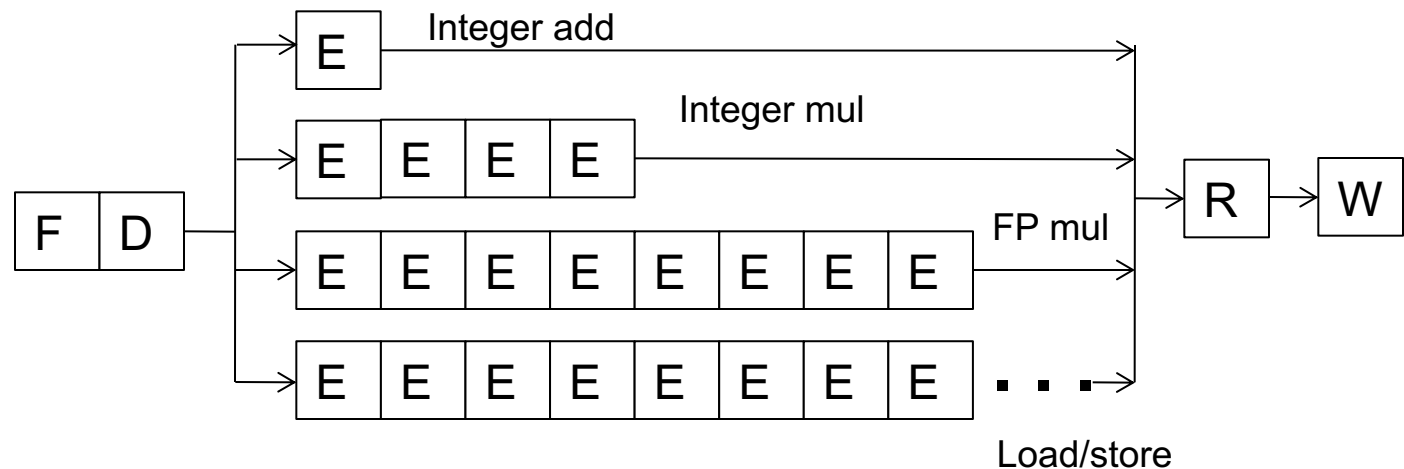
- ❑ No need to read the new values from the ROB (no CAM or indirection) or the old value of destination register

■ Disadvantage

- ❑ Multiple register files
- ❑ Need to copy arch. reg. file to future file on an exception

In-Order Pipeline with Future File and Reorder Buffer

- **Decode (D)**: Access future file, 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 **and future file**
- **Retirement/Commit (W)**: Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline, **copy architectural file to future file**, and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**

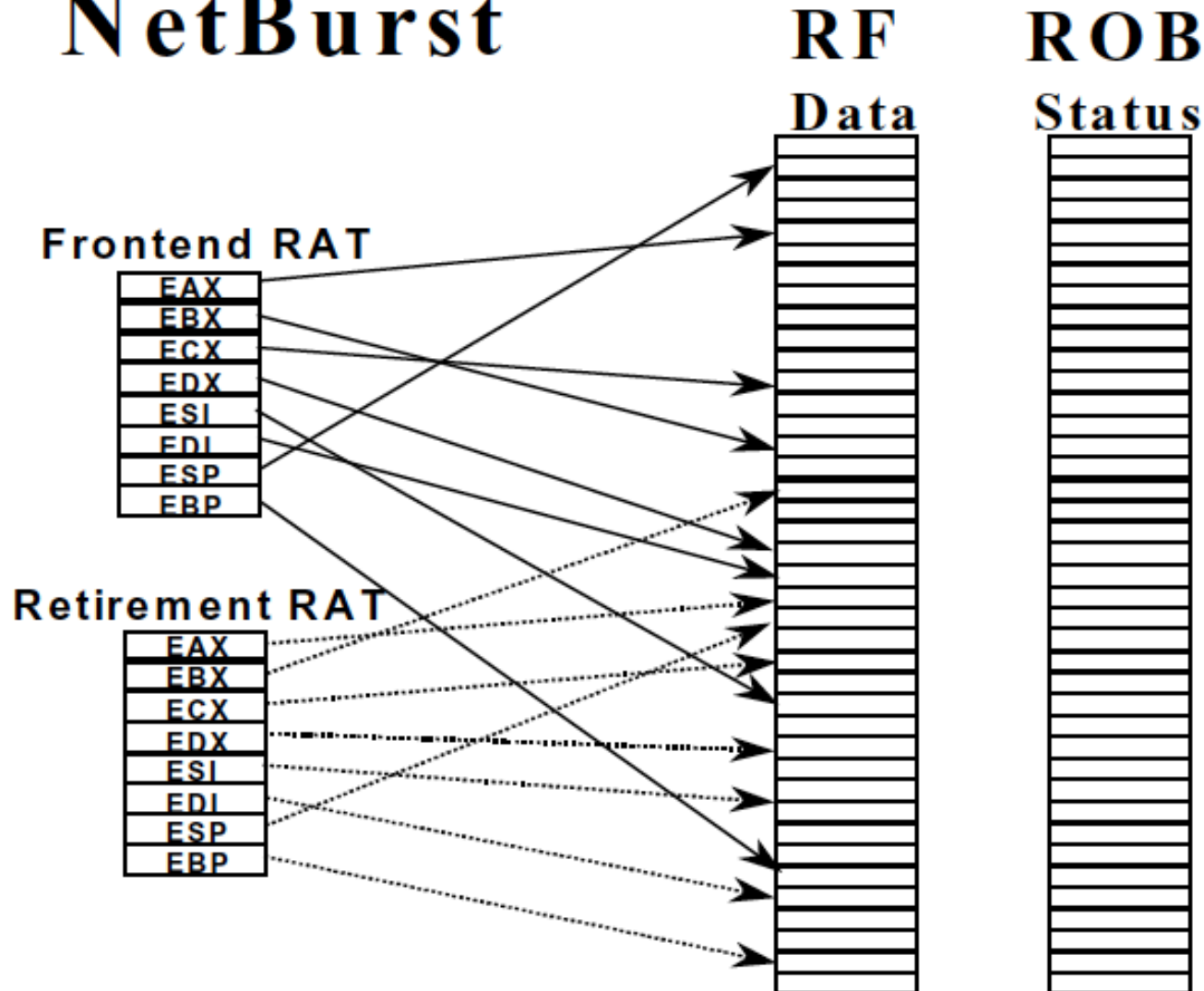


Can We Reduce the Overhead of Two Register Files?

- Idea: Use indirection, i.e., pointers to data in frontend and retirement
 - Have a single storage that stores register data values
 - Keep two register maps (speculative and architectural); also called register alias tables (RATs)
- Future map used for fast access to latest register values (speculative state)
 - Frontend register map
- Architectural map is used for state recovery on exceptions (architectural state)
 - Backend register map

Future Map in Intel Pentium 4

NetBurst



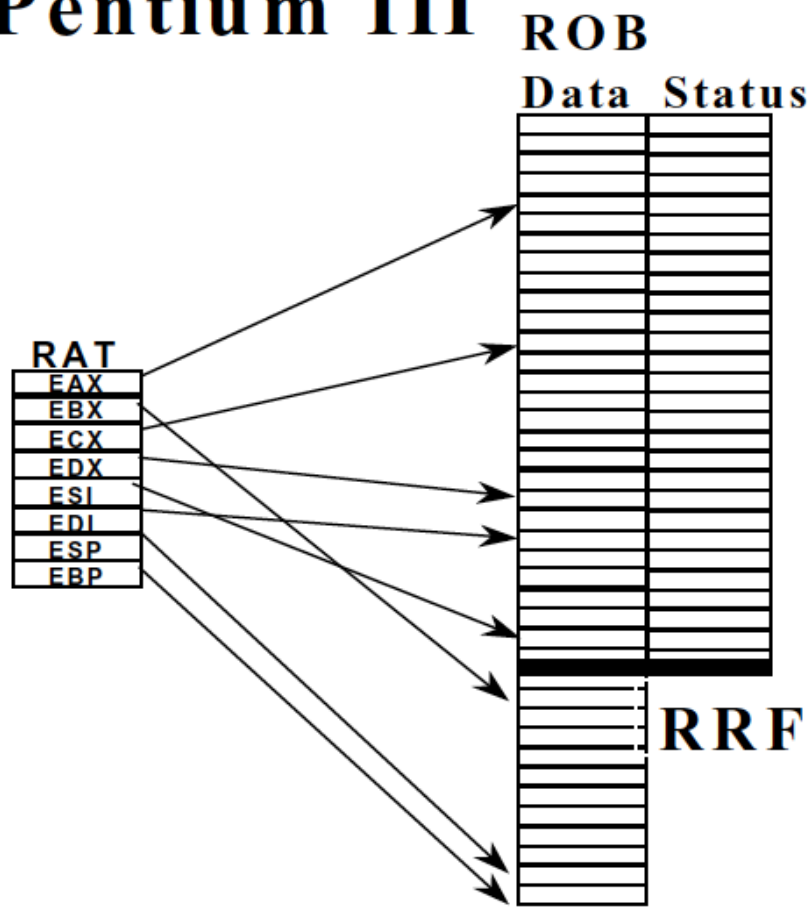
Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Many modern processors are similar:

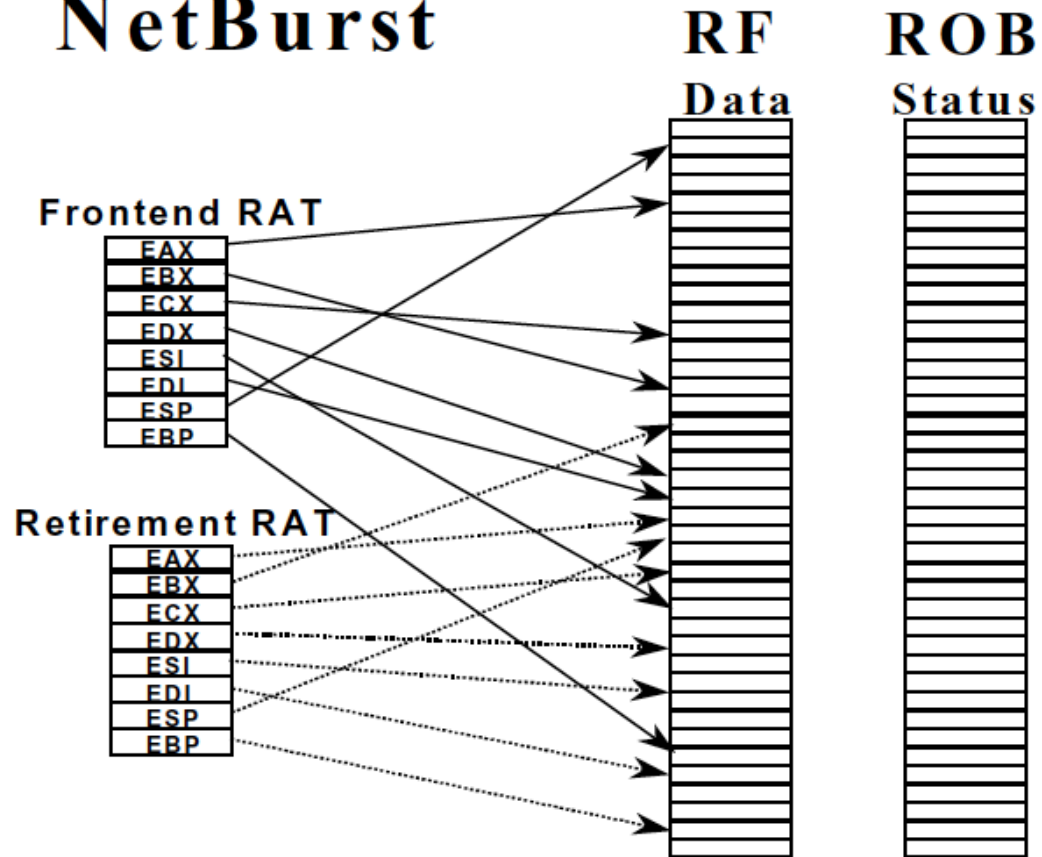
- MIPS R10K
- Alpha 21264

Reorder Buffer vs. Future Map Comparison

Pentium III



NetBurst



Before We Get to Checkpointing ...

- Let's cover what happens on exceptions
- And branch mispredictions

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
 - ❑ Recovers architectural state (register file, IP, and memory)
 - ❑ Flushes all younger instructions in the pipeline
 - ❑ Saves IP and registers (as specified by the ISA)
 - ❑ Redirects the fetch engine to the exception handling routine
 - Vectored exceptions

Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an “exception”
 - Except it is not visible to software (i.e., it is microarchitectural)
- What about branch misprediction recovery?
 - Similar to exception handling except can be initiated before the branch is the oldest instruction (not architectural)
 - All three state recovery methods can be used
- Difference between exceptions and branch mispredictions?
 - Branch mispredictions are much more common
 - need fast state recovery to minimize performance impact of mispredictions

How Fast Is State Recovery?

- Latency of state recovery affects
 - Exception service latency
 - Interrupt service latency
 - Latency to supply the correct data to instructions fetched after a branch misprediction

- Which ones above need to be fast?

- How do the three state maintenance methods fare in terms of recovery latency?
 - Reorder buffer
 - History buffer
 - Future file

Branch State Recovery Actions and Latency

- Reorder Buffer
 - ❑ Flush instructions in pipeline younger than the branch
 - ❑ Finish all instructions in the reorder buffer

- History buffer
 - ❑ Flush instructions in pipeline younger than the branch
 - ❑ Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values one by one into the register file

- Future file
 - ❑ Wait until branch is the oldest instruction in the machine
 - ❑ Copy arch. reg. file to future file
 - ❑ Flush entire pipeline

Can We Do Better?

- Goal: Restore the frontend state (future file) such that the correct next instruction after the branch can execute right away after the branch misprediction is resolved
- Idea: Checkpoint the frontend register state/map at the time a branch is decoded and keep the checkpointed state updated with results of instructions older than the branch
 - Upon branch misprediction, restore the checkpoint associated with the branch
- Hwu and Patt, “Checkpoint Repair for Out-of-order Execution Machines,” ISCA 1987.

Checkpointing

- When a branch is decoded
 - Make a copy of the future file/map and associate it with the branch
- When an instruction produces a register value
 - All future file/map checkpoints that are younger than the instruction are updated with the value
- When a branch misprediction is detected
 - Restore the checkpointed future file/map for the mispredicted branch when the branch misprediction is resolved
 - Flush instructions in pipeline younger than the branch
 - Deallocate checkpoints younger than the branch

Checkpointing

■ Advantages

- Correct frontend register state available right after checkpoint restoration → Low state recovery latency
- ...

■ Disadvantages

- Storage overhead
- Complexity in managing checkpoints
- ...

Many Modern Processors Use Checkpointing

- MIPS R10000
- Alpha 21264
- Pentium 4

- Yeager, “The MIPS R10000 Superscalar Microprocessor,” IEEE Micro, April 1996

- Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro, March-April 1999.

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

Summary: Maintaining Precise State

- Reorder buffer
- History buffer
- Future register file
- Checkpointing
- Readings
 - Smith and Plezskun, “[Implementing Precise Interrupts in Pipelined Processors](#),” IEEE Trans on Computers 1988 and ISCA 1985.
 - Hwu and Patt, “[Checkpoint Repair for Out-of-order Execution Machines](#),” ISCA 1987.

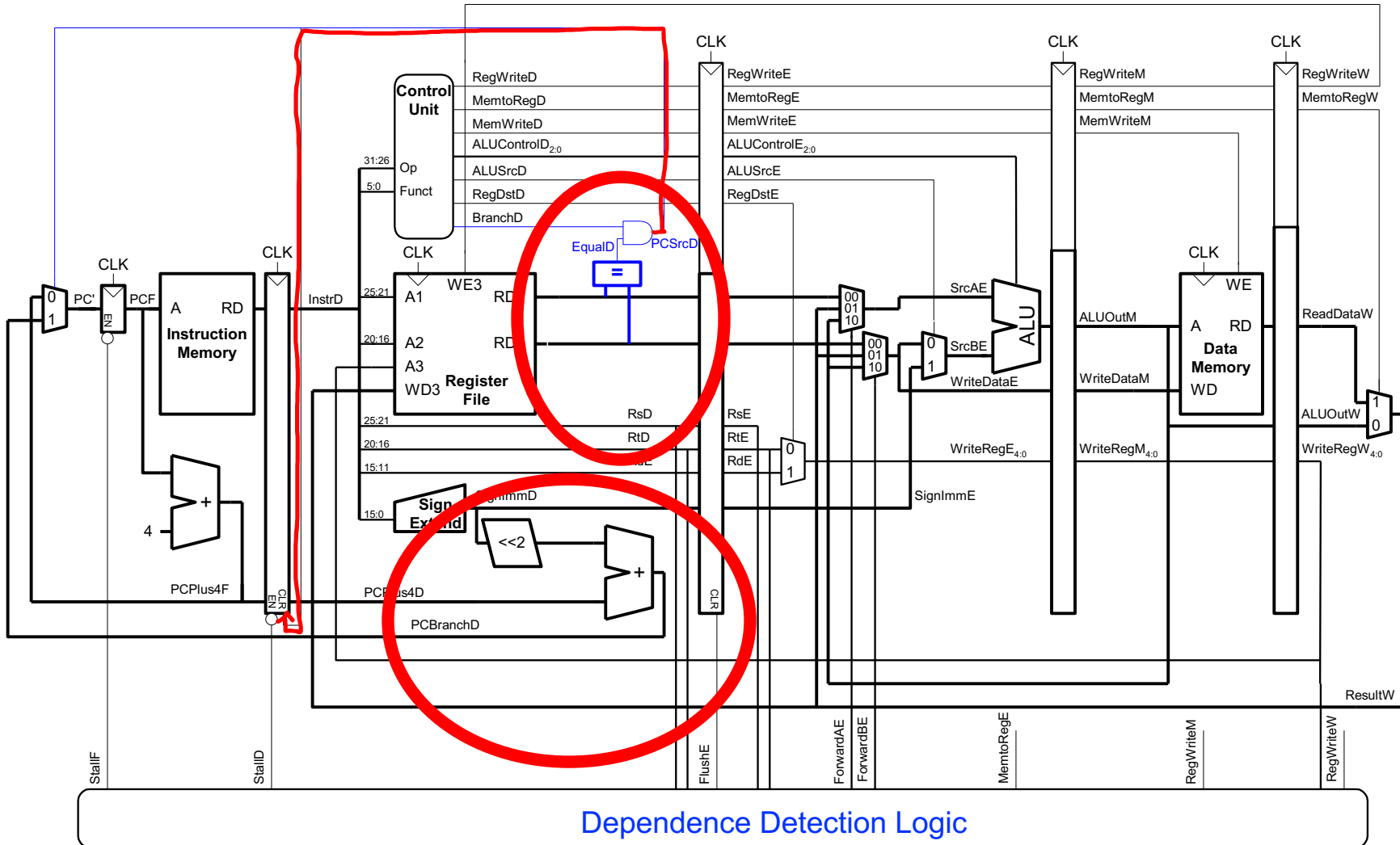
Registers versus Memory

- So far, we considered mainly registers as part of state
- What about memory?
- What are the fundamental differences between registers and memory?
 - Register dependences known statically – memory dependences determined dynamically
 - Register state is small – memory state is large
 - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)

Maintaining Speculative Memory State: Stores

- Handling out-of-order completion of memory operations
 - UNDOing a memory write more difficult than UNDOing a register write. Why?
 - One idea: Keep store address/data in reorder buffer
 - How does a load instruction find its data?
 - Store/write buffer: Similar to reorder buffer, but used only for store instructions
 - Program-order list of un-committed store operations
 - When store is decoded: Allocate a store buffer entry
 - When store address and data become available: Record in store buffer entry
 - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data
- We will get back to this!

Pipeline with Early Branch Resolution



Need to calculate branch target and condition in the Decode Stage