# Pipelined MIPS Processor

Dr. Igor Ivkovic

iivkovic@uwaterloo.ca

[with material from “Computer Organization and Design” by Patterson and Hennessy, and “Digital Design and Computer Architecture” by Harris and Harris, both published by Morgan Kaufmann]

# Objectives

- Introduction to MIPS Pipeline
- Different Hazard Types
- Handling Data Hazards
- Handling Control Hazards
- Handling Exceptions
- Advanced MIPS Pipeline Design

# Introduction to MIPS Pipeline /1

- **Pipelining Introduced:**
  - The goal is to improve the throughput of digital circuitry
  - We design a MIPS pipeline by dividing the single-cycle processor into five pipeline stages
  - In each stage, one instruction can run, so five instructions can run simultaneously
  - **Pipelining can ideally improve the throughput by five times**
  - Pipelining does introduce overhead, but unlike the multi-cycle processor, the performance advantage is significant
  - **As the result, most modern high-performance digital architectures are based on pipelining**

# Introduction to MIPS Pipeline /2

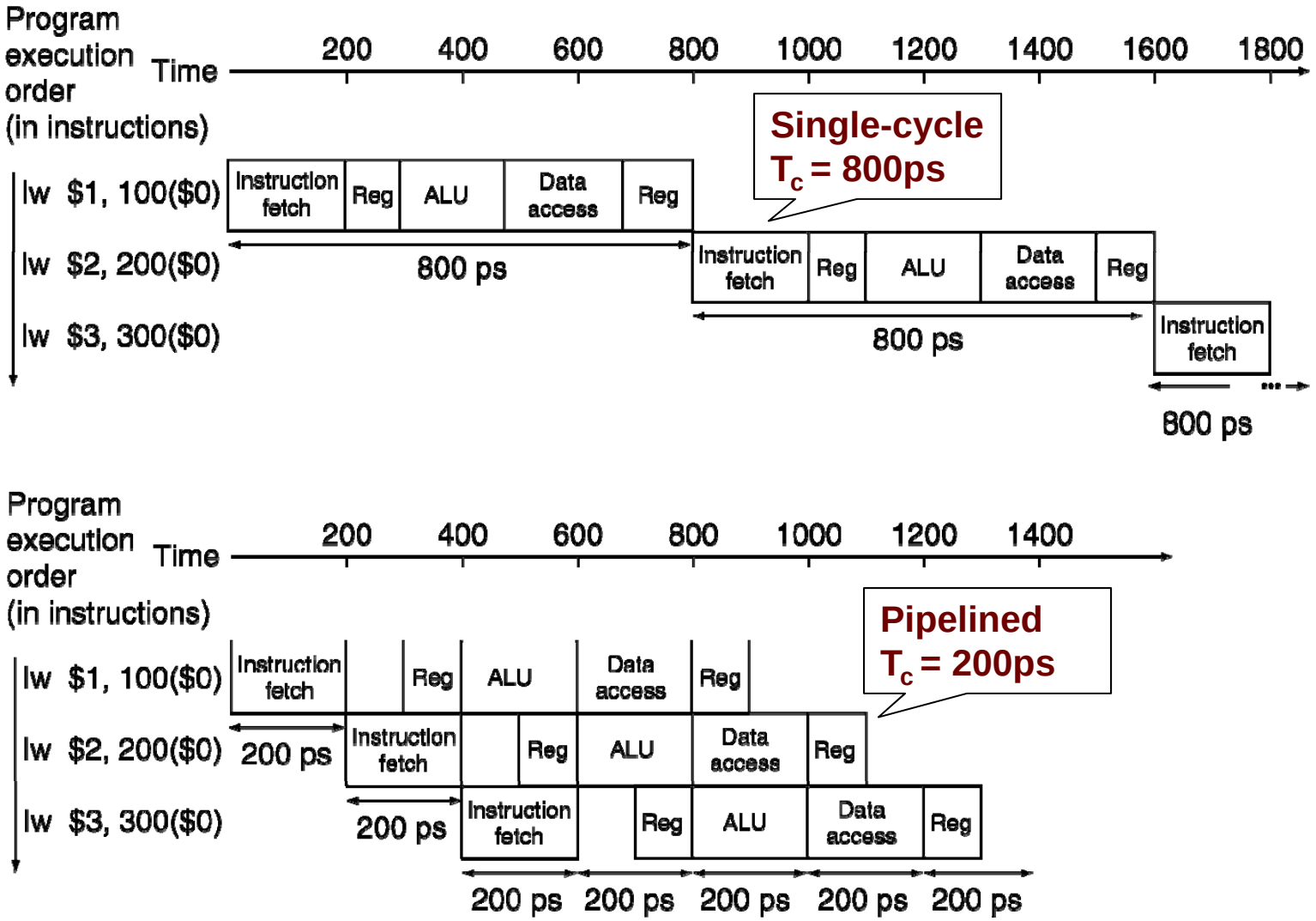
- **Pipelining Elaborated:**
  - Similar as in the multi-cycle processor, we focus on the operations that create the largest time delay
  - These operations include reading from and writing to memory, register file access, and ALU operations
  - Five stages, one step per stage
- **The stages are similar to the stages that multi-cycle processor goes through when executing `lw`:**
  - **Fetch (IF):** Instruction fetch from memory
  - **Decode (ID):** Instruction decode and register read
  - **Execute (EX):** Execute operation or calculate address
  - **Memory (MEM):** Access memory operand
  - **Writeback (WB):** Write result back to register

# Introduction to MIPS Pipeline /3

- **Five stages, one step per stage:**
  - **Fetch (IF):** Instruction fetch from memory
  - **Decode (ID):** Instruction decode and register read
  - **Execute (EX):** Execute operation or calculate address
  - **Memory (MEM):** Access memory operand
  - **Writeback (WB):** Write result back to register
- **Add pipeline registers between stages**
- **Pipeline performance:**
  - Assume time for stages is
    - 100ps for register read or write
    - 200ps for other stages
  - Let us compare pipelined datapath with single-cycle datapath

# Introduction to MIPS Pipeline /4

- **Pipeline Performance:**



# Introduction to MIPS Pipeline /5

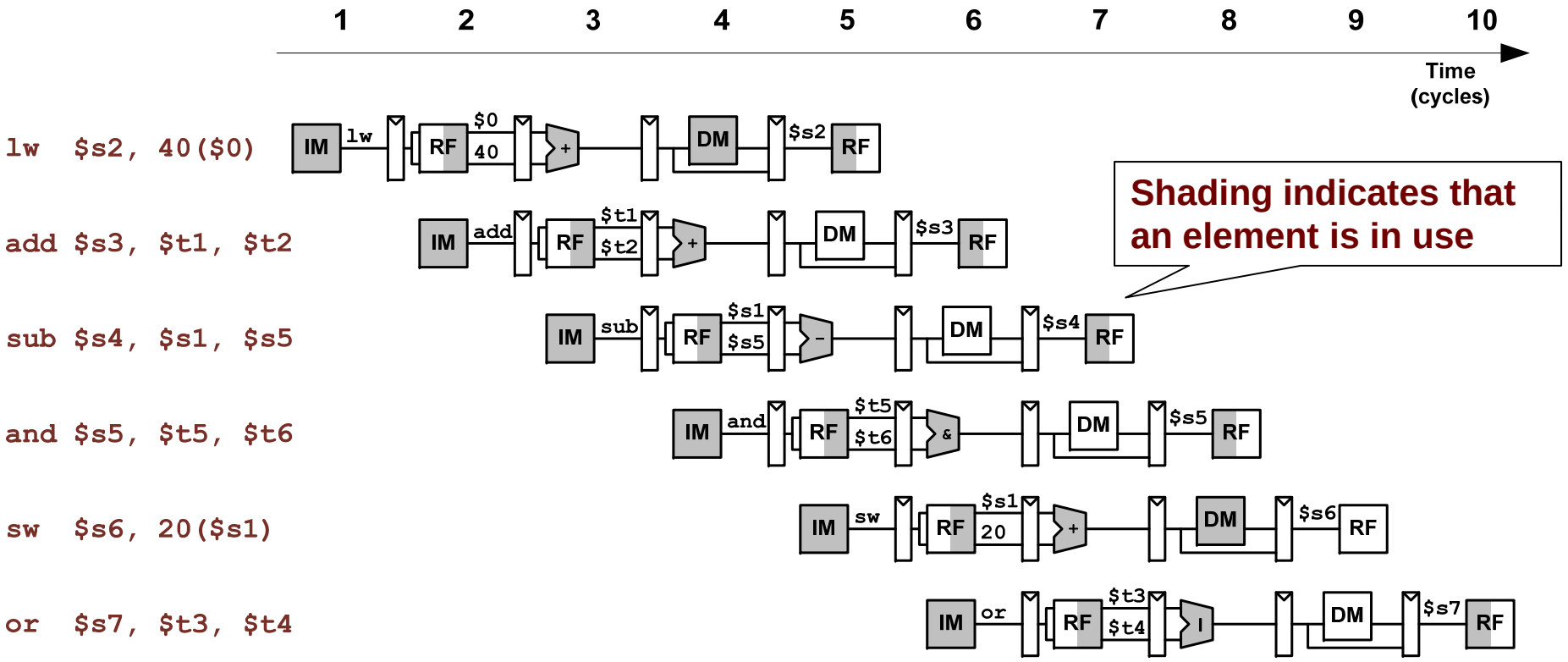
- **Pipeline Speedup:**
  - If all stages are balanced and take the same time, then

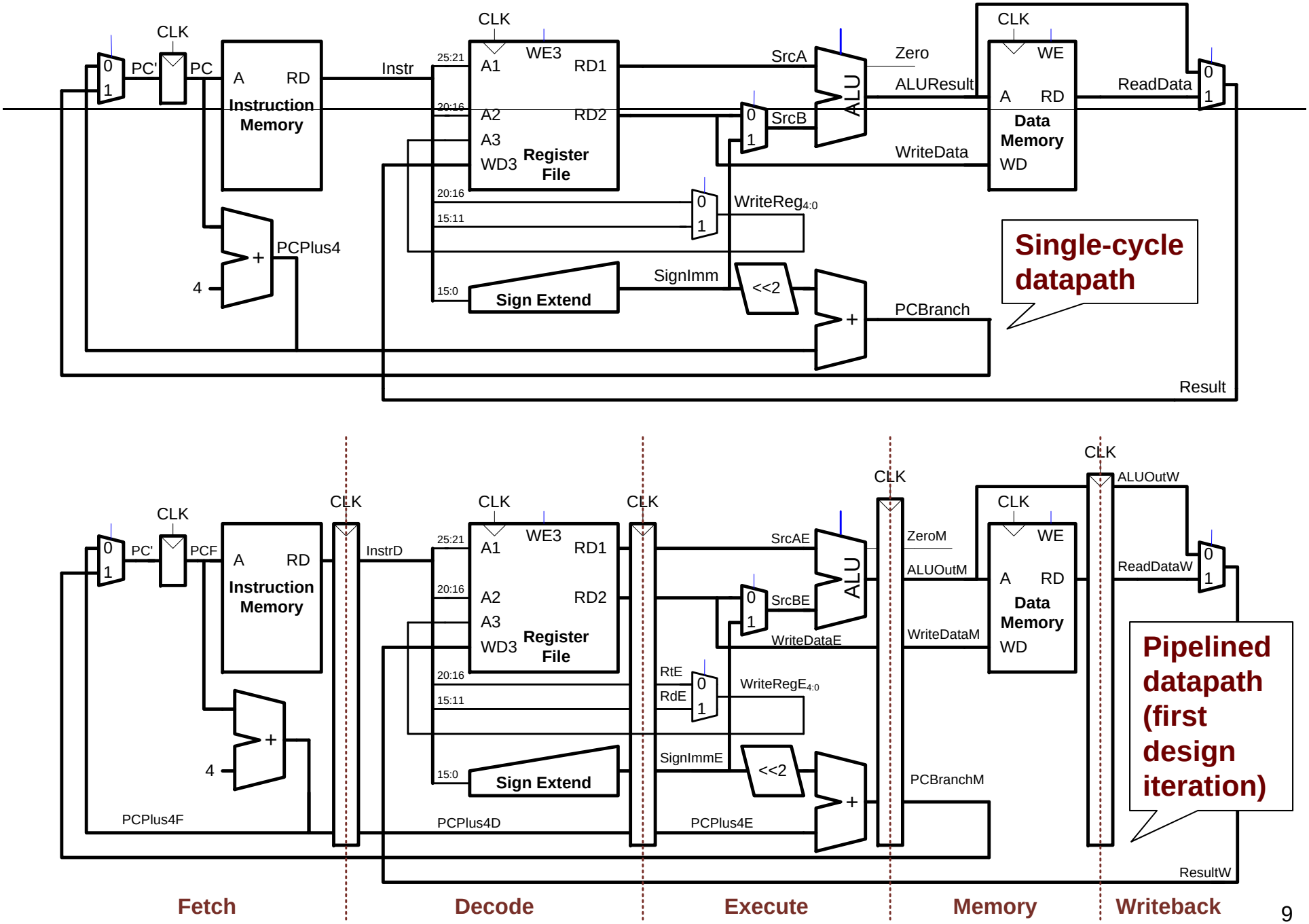
  **{time between instructions}$_{pipelined}$ =**

  **{time between instructions}$_{non\text{-}pipelined}$ / {number of stages}**
  - The speedup is due to the increased throughput
  - **If not balanced, the pipeline speedup is less**
  - Latency per each instruction does not decrease
- **MIPS ISA designed for pipelining:**
  - All instructions are 32-bits, so they are easier to fetch and decode in one cycle
  - MIPS allows decoding and reading of registers in one step
  - Memory operands are aligned, so memory access takes only one cycle

# Introduction to MIPS Pipeline /6

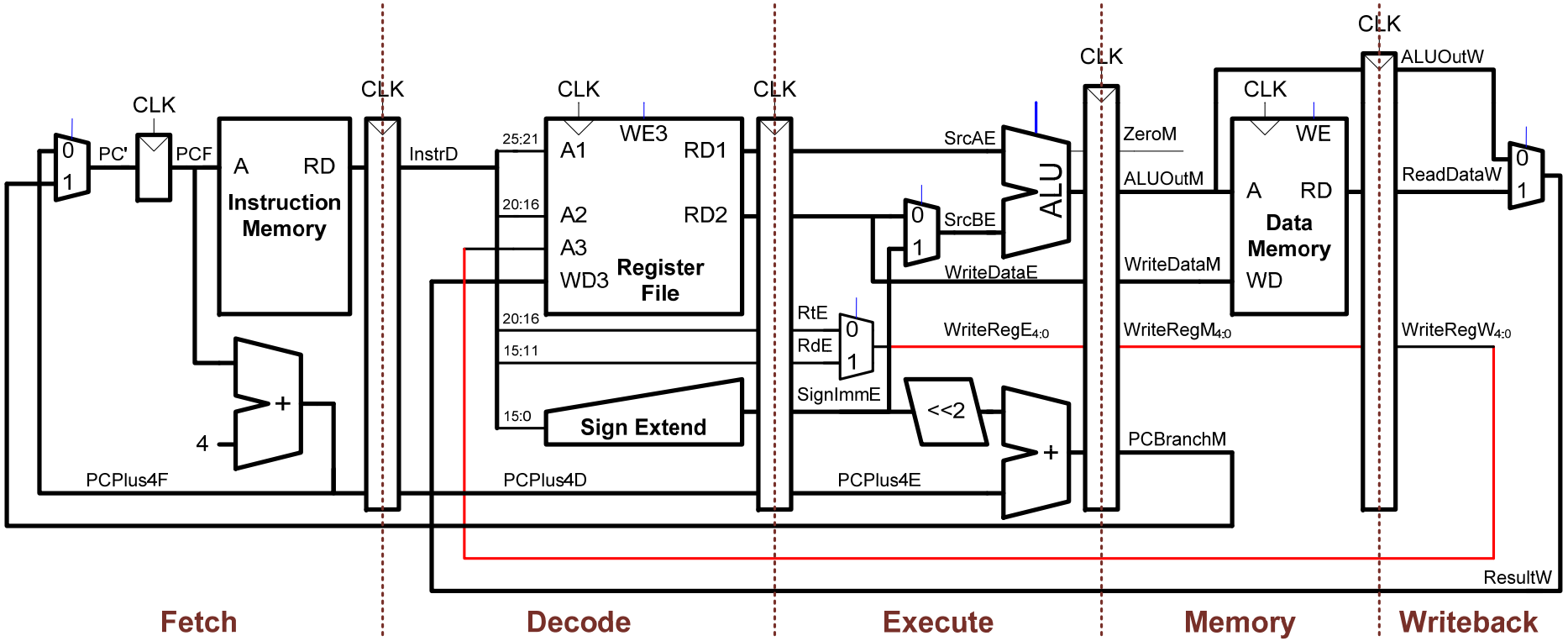
- **Pipelining Elaborated:**




CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
20:16
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
Register File
SrcA
Zero
ALU
ALUResult
SrcB
WriteData
CLK
WE
A
RD
Data Memory
WD
ReadData
20:16
15:11
WriteReg4:0
PCPlus4
4
15:0
Sign Extend
SignImm
<<2
PCBranch
Result
Single-cycle datapath
CLK
PC'
PCF
A
RD
Instruction Memory
InstrD
CLK
25:21
20:16
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
Register File
CLK
SrcAE
ZeroM
ALU
ALUOutM
SrcBE
WriteDataE
WriteDataM
CLK
WE
A
RD
Data Memory
WD
CLK
ALUOutW
ReadDataW
RtE
RdE
20:16
15:11
WriteRegE4:0
4
15:0
Sign Extend
SignImmE
<<2
PCBranchM
PCPlus4F
PCPlus4D
PCPlus4E
ResultW
Pipelined datapath (first design iteration)
Fetch
Decode
Execute
Memory
Writeback

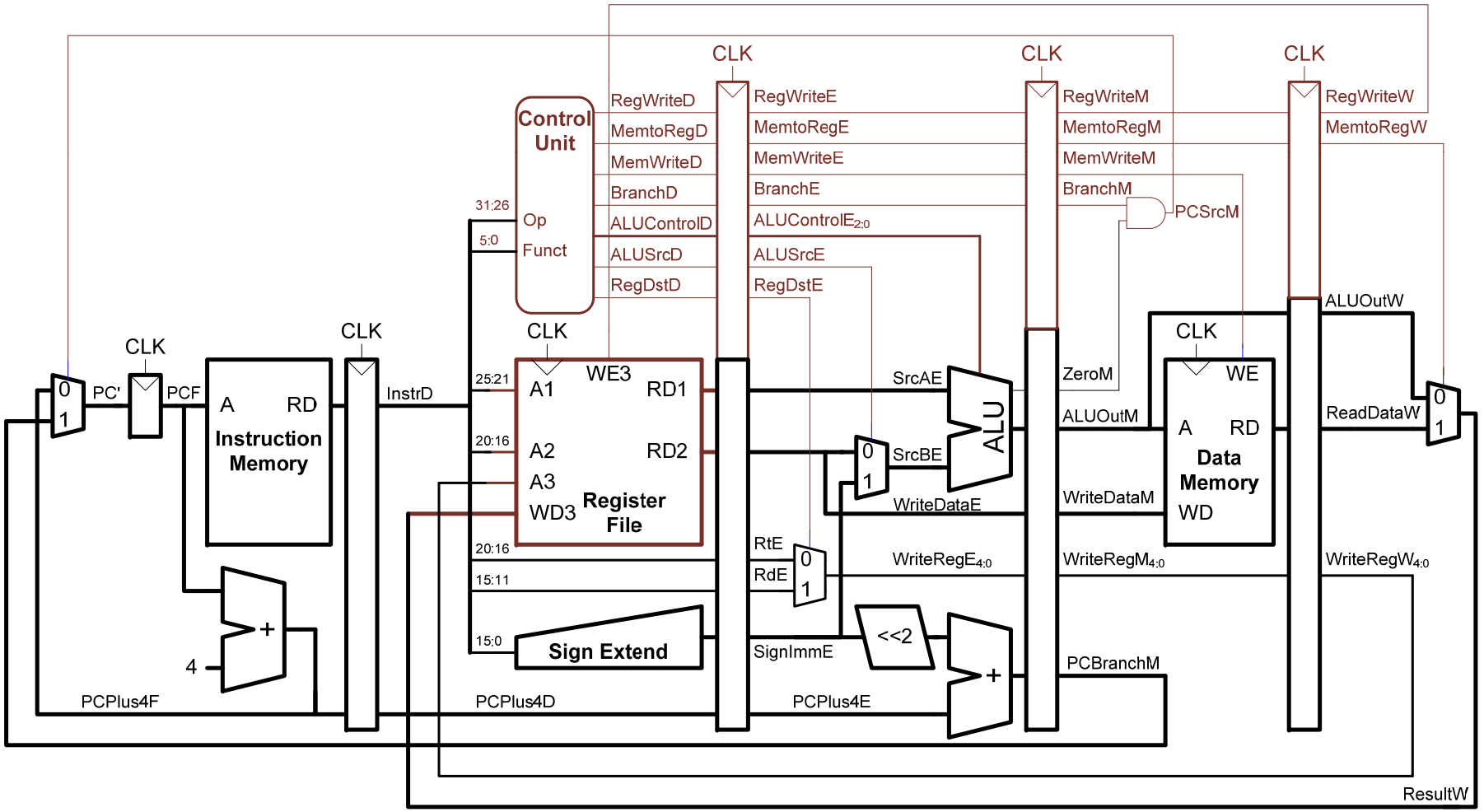
# Introduction to MIPS Pipeline /8

- **Corrected Pipelined Datapath:**
  - WriteReg must be carried through and arrive at the same time as the Result signal; otherwise, `lw` on cycle 5 writes to $s4



# Introduction to MIPS Pipeline /9

- **Pipelined Datapath Control Unit:**
  - Same control unit as single-cycle processor
  - Control delayed to proper pipeline stage

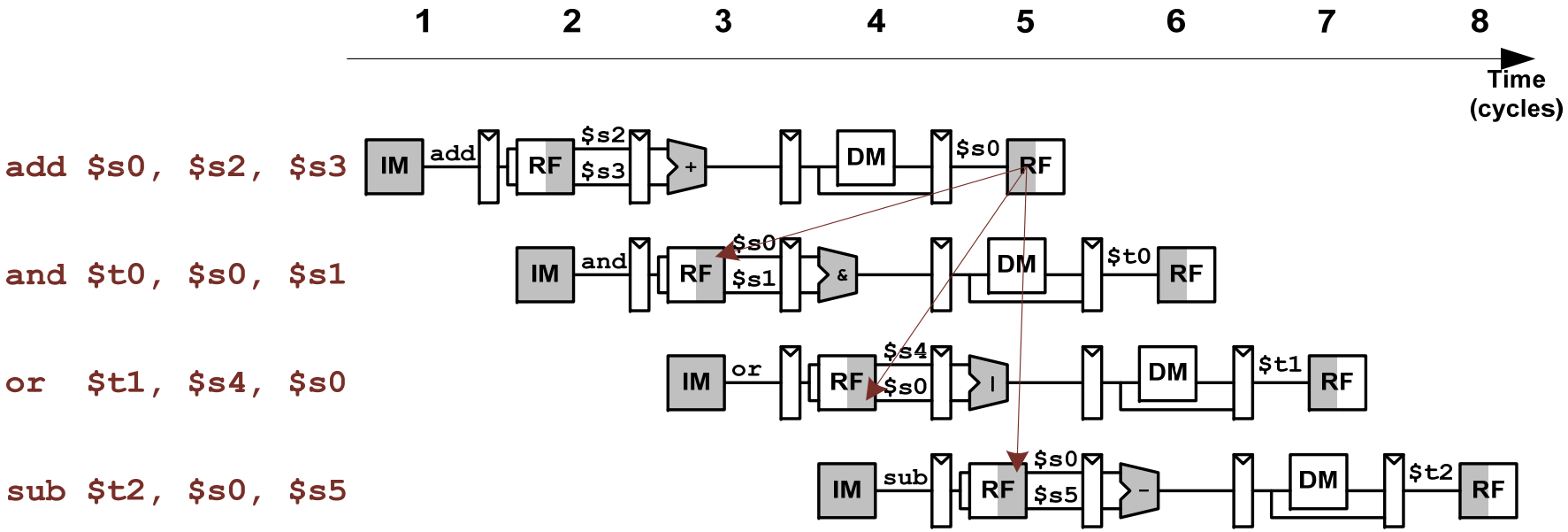


# Introduction to MIPS Pipeline /10

- **Hazard:**
  - A problem in execution where the subsequent instruction cannot execute
  - Occurs when an instruction depends on result from instruction that hasn't completed
- **Types of Hazards:**
  - **Structure Hazard:** A required architecture resource is busy
    - For instance, instruction fetch would have to stall for that cycle
    - **Addressed by separating Instruction and Data Memory into different units**
  - **Data Hazard:** Need to wait for previous instruction to complete its data read/write
  - **Control Hazard:** Next instruction not decided yet
    - Typically caused by branching
    - Predict outcome of branch and only stall if prediction is wrong

# Handling Data Hazards /1

- **Data Hazards:**
  - Register value not yet written back to register file
- **Read After Write (RAW) Hazard:**
  - Read occurs after the needed Write



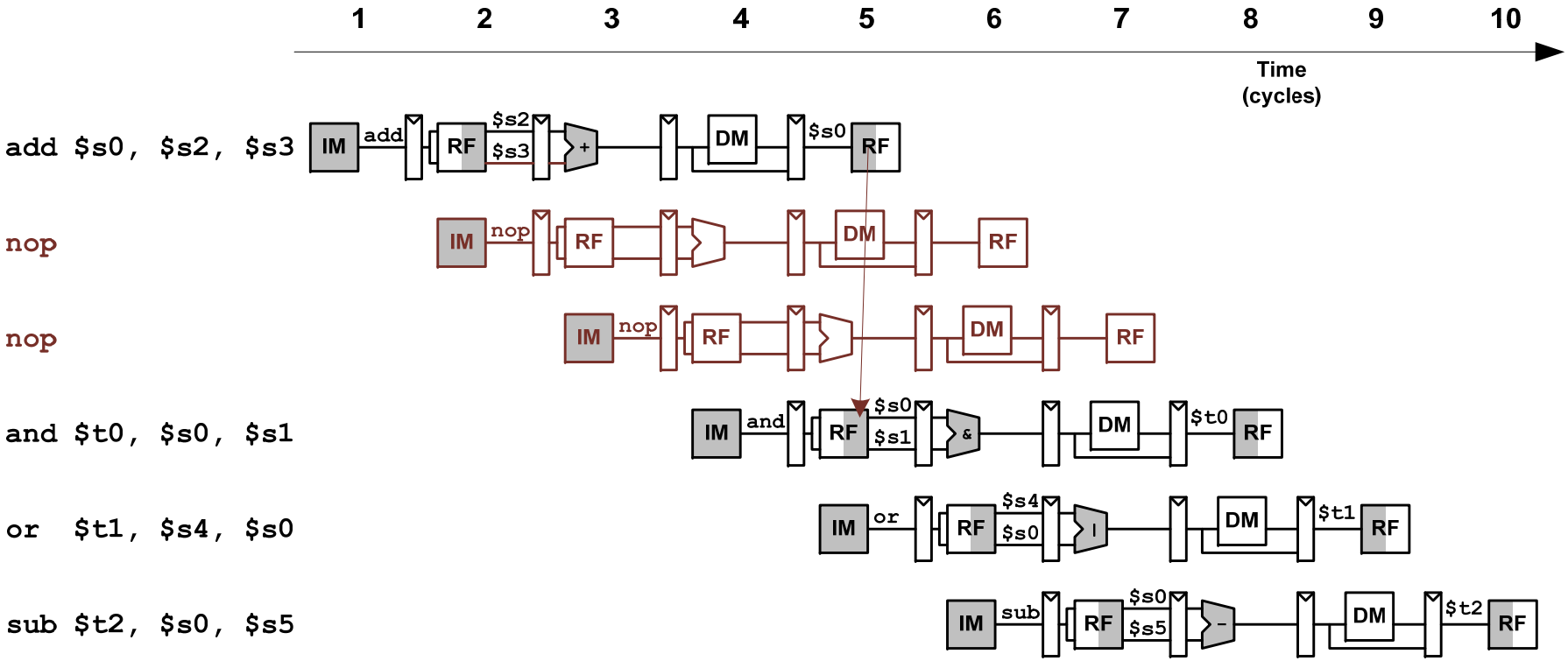

**`and` and `or` obtain incorrect values, but `sub` obtains the correct value**

# Handling Data Hazards /2

- **Handling Data Hazards:**
  1. Insert `nop` in code at the compile time
  2. Rearrange code at the compile time
  3. Forward data at run time
  4. Stall the processor at run time
- **Handle certain hazards through a special Hazard Unit with its own control signals**

# Handling Data Hazards /3

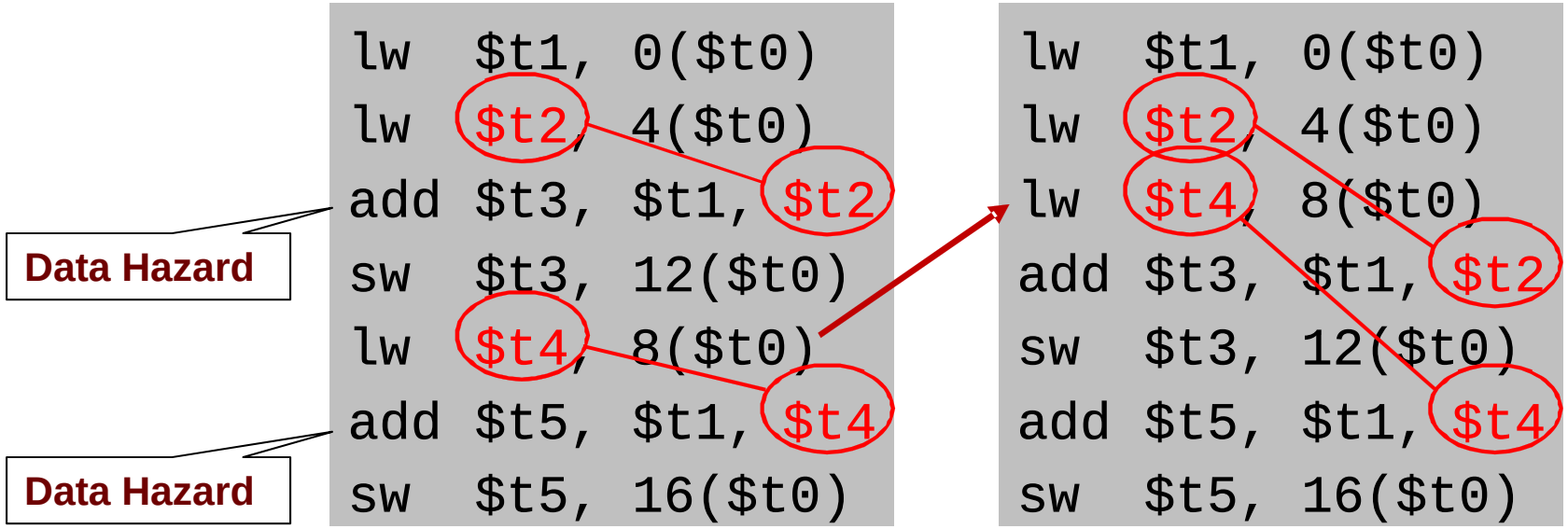
- **Compile-Time Hazard Elimination:**
  - Insert enough **`nop`** for result to be ready
  - No op(eration) or **`nop`** performs no operation (**`sll $0, $0, 0`**)



# Handling Data Hazards /4

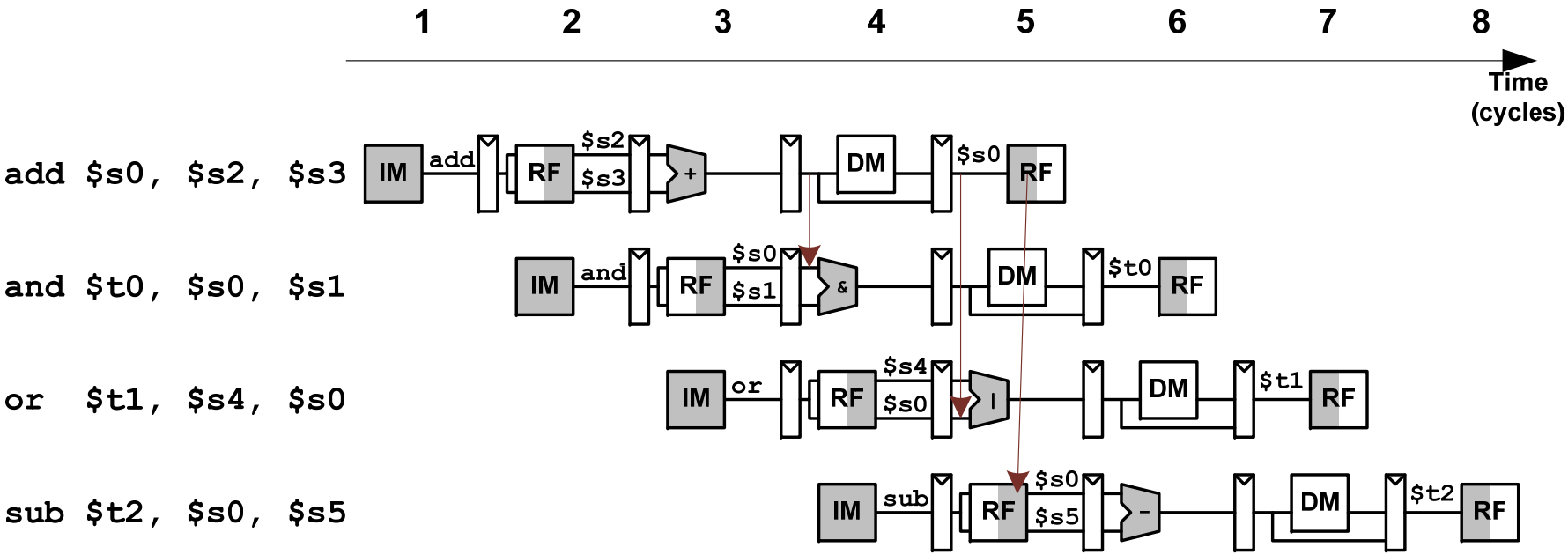
- **Compile-Time Hazard Elimination:**
  - Reorder code to avoid use of load result in the next instruction
  - Example: MIPS code for

```
A = B + E;
C = B + F;
```



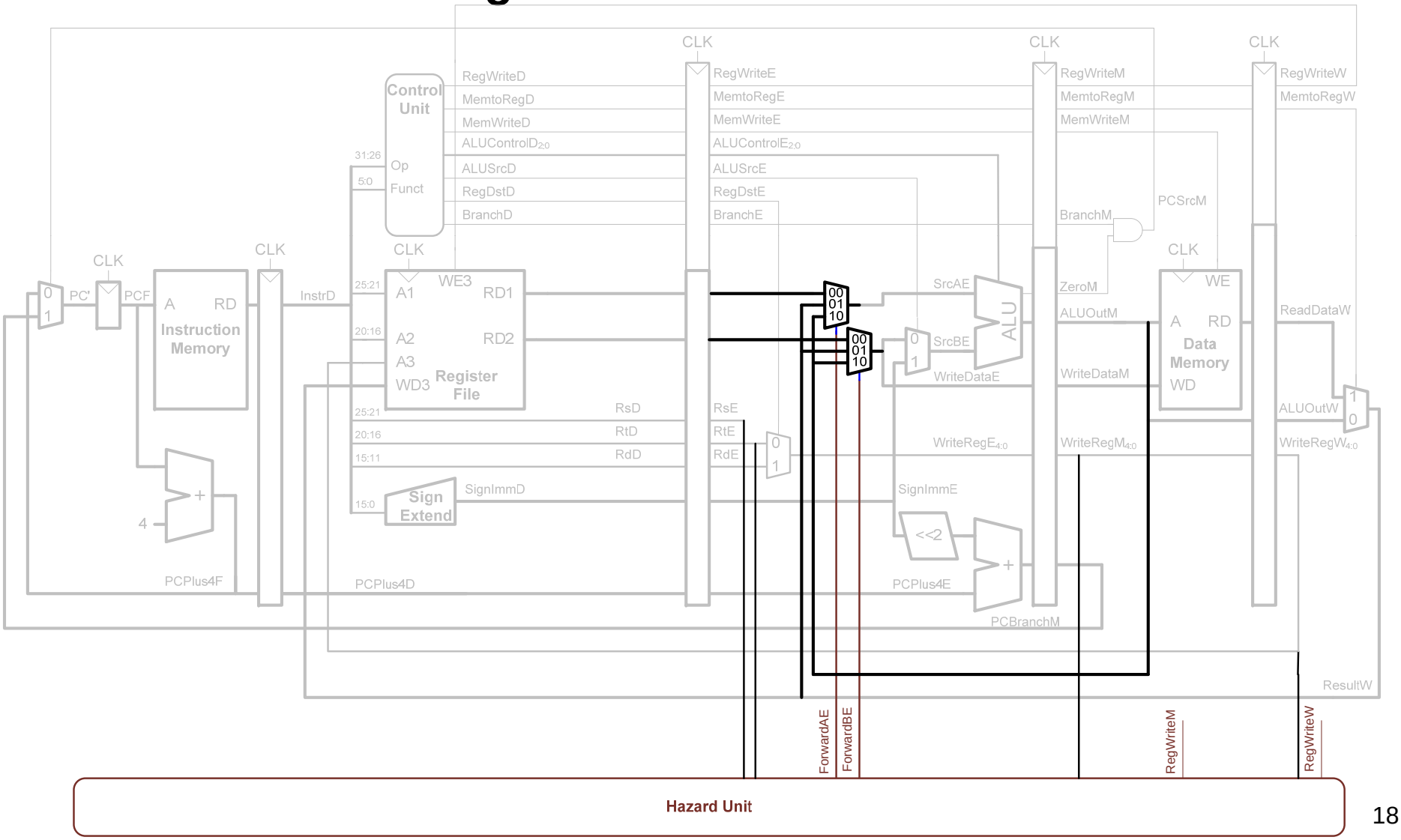
# Handling Data Hazards /5

- **Data Forwarding:**
  - Certain data hazards can be handled by forwarding (or bypassing) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage
  - Achieved by adding multiplexers in front of the ALU to select the operand from either the RF, the Memory or Writeback stage



# Handling Data Hazards /6

- **Data Forwarding:**



# Handling Data Hazards /7

- **Data Forwarding via Hazard Unit:**
  - **Forward to Execute stage from either: Memory stage or Writeback stage**
  - Forwarding logic for **`ForwardAE (SrcA)`**:

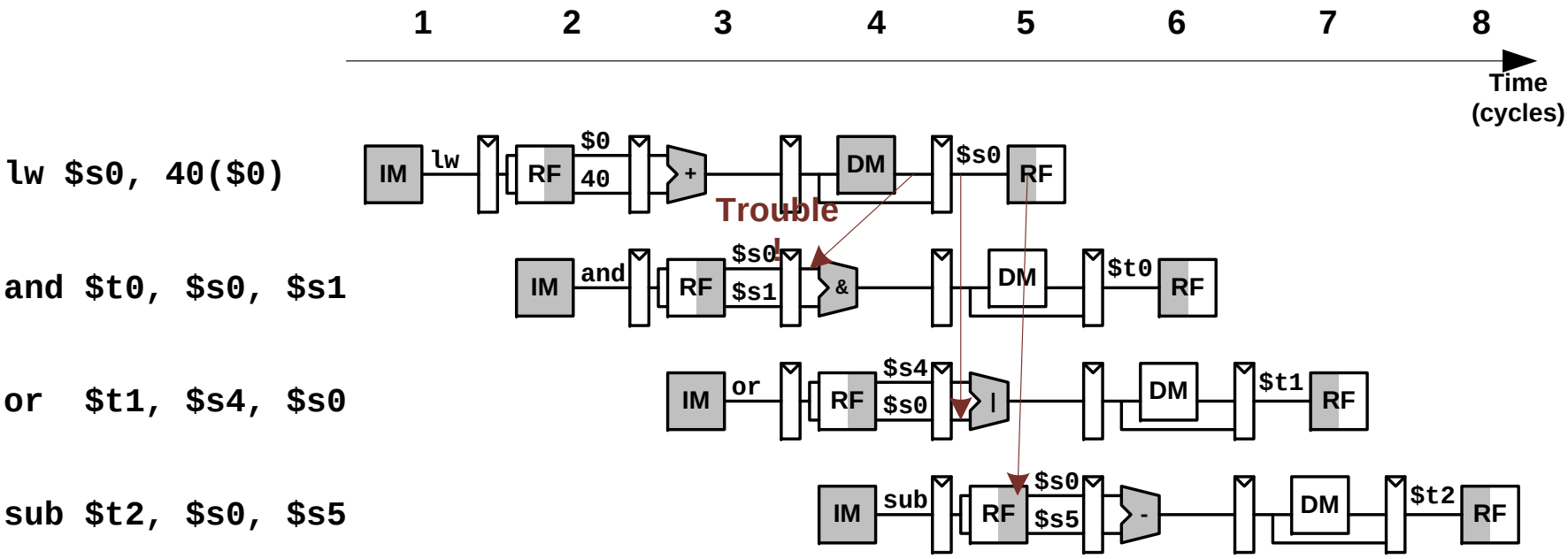
```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
    then      ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND
    RegWriteW)
    then      ForwardAE = 01
else          ForwardAE = 00
```

  - Forwarding logic for **`ForwardBE (SrcA):`**

```
if ((rtE != 0) AND (rtE == WriteRegM) AND RegWriteM)
    then      ForwardBE = 10
else if ((rtE != 0) AND (rtE == WriteRegW) AND
    RegWriteW)
    then      ForwardBE = 01
else          ForwardBE = 00
```

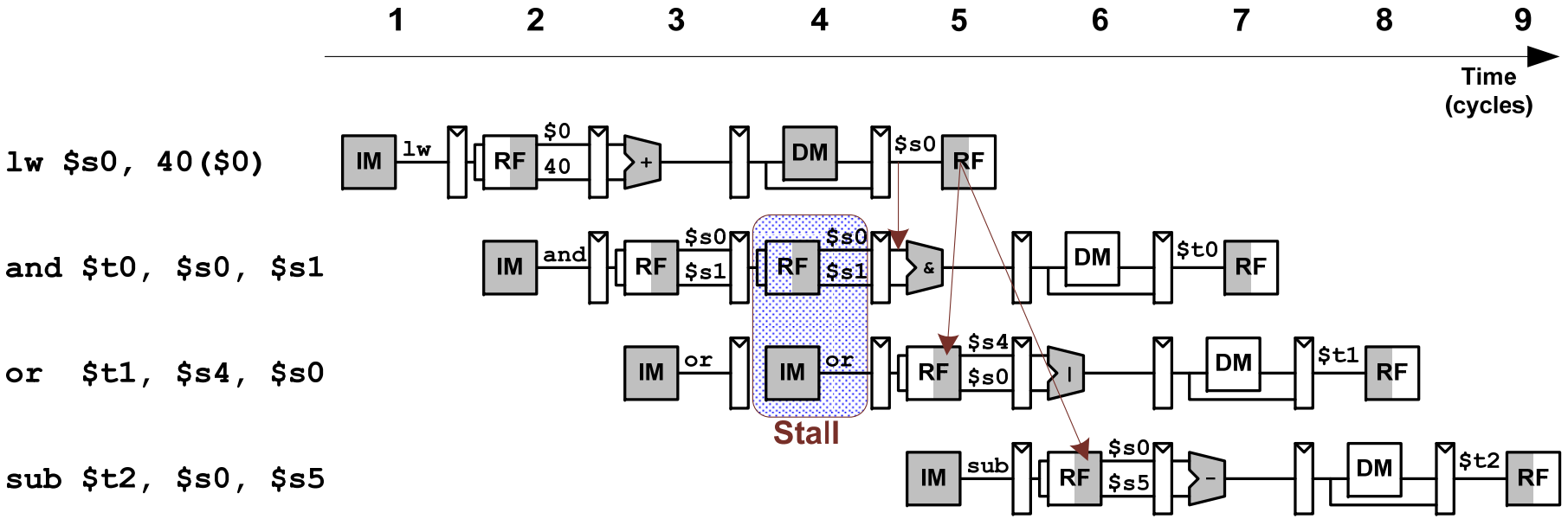
# Handling Data Hazards /8

- **Why stall at run time?**
  - The `lw` instruction experiences two-cycle latency
  - `lw` does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage



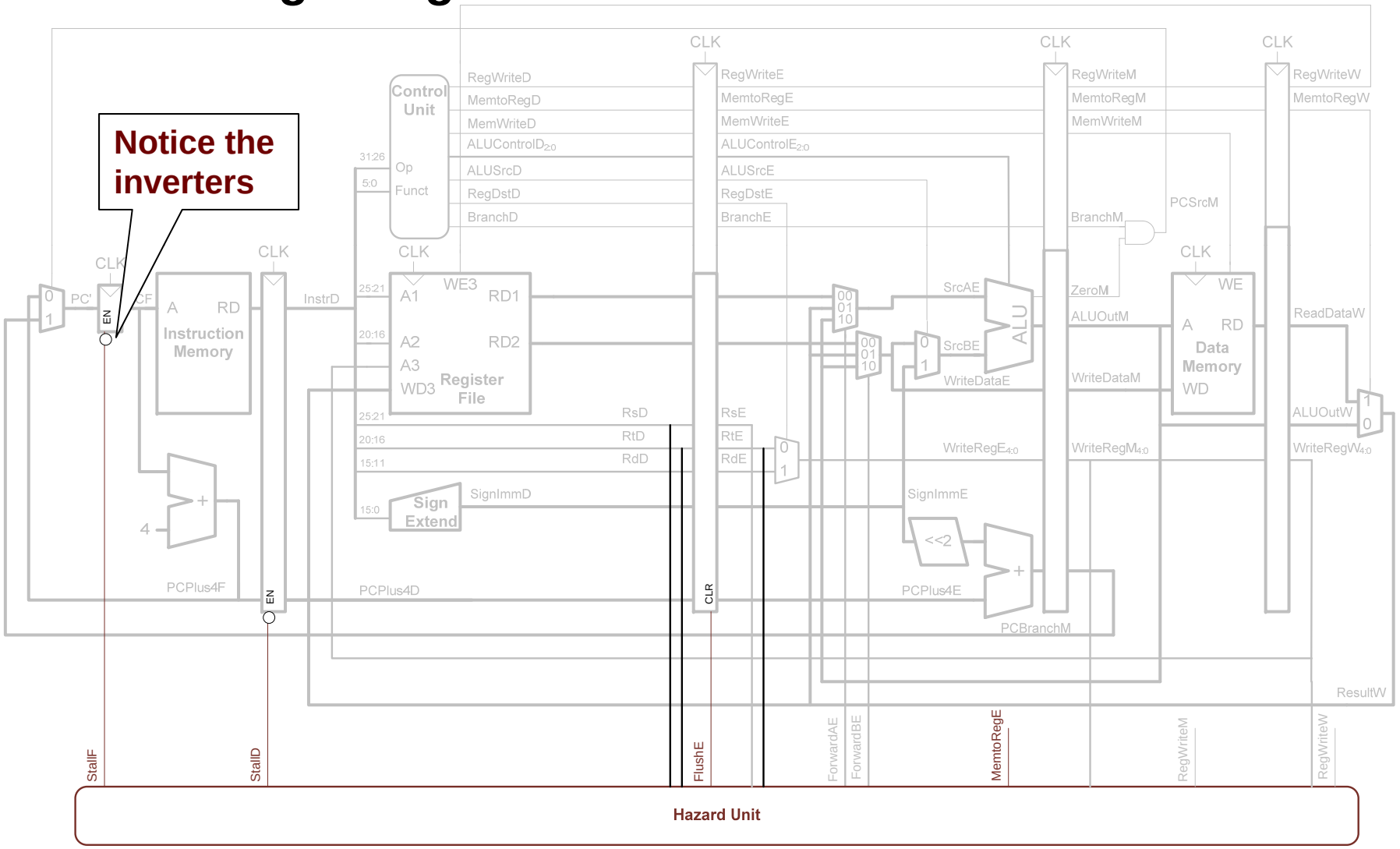
# Handling Data Hazards /9

- **Stalling:**
  - **Solution: Hold up the pipeline (called stalling) until the needed data is available**
  - The unused stage propagation through the pipeline is called a bubble, and it behaves like a `nop` instruction
  - **The bubble is introduced by zeroing out the EX stage control signals during a Decode stall, so that the bubble performs no action and changes no architectural state**



# Handling Data Hazards /10

- **Stalling using the Hazard Unit:**



# Handling Data Hazards /11

- **Stall by disabling the pipeline register, so that the contents do not change**
  - The pipeline register directly after the stalled stage must be cleared to prevent invalid information from moving forward
  - Stalls typically degrade the pipeline performance, so should be used sparingly
- **Stalling logic is relatively simple:**
  - Stall if dealing with lw and the destination register from EX is one of the operands in the ID stage
  - **Logic encoding of Stall bits that serve as EN bits:**

```
lwstall = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE
StallF = StallD = FlushE = lwstall
```

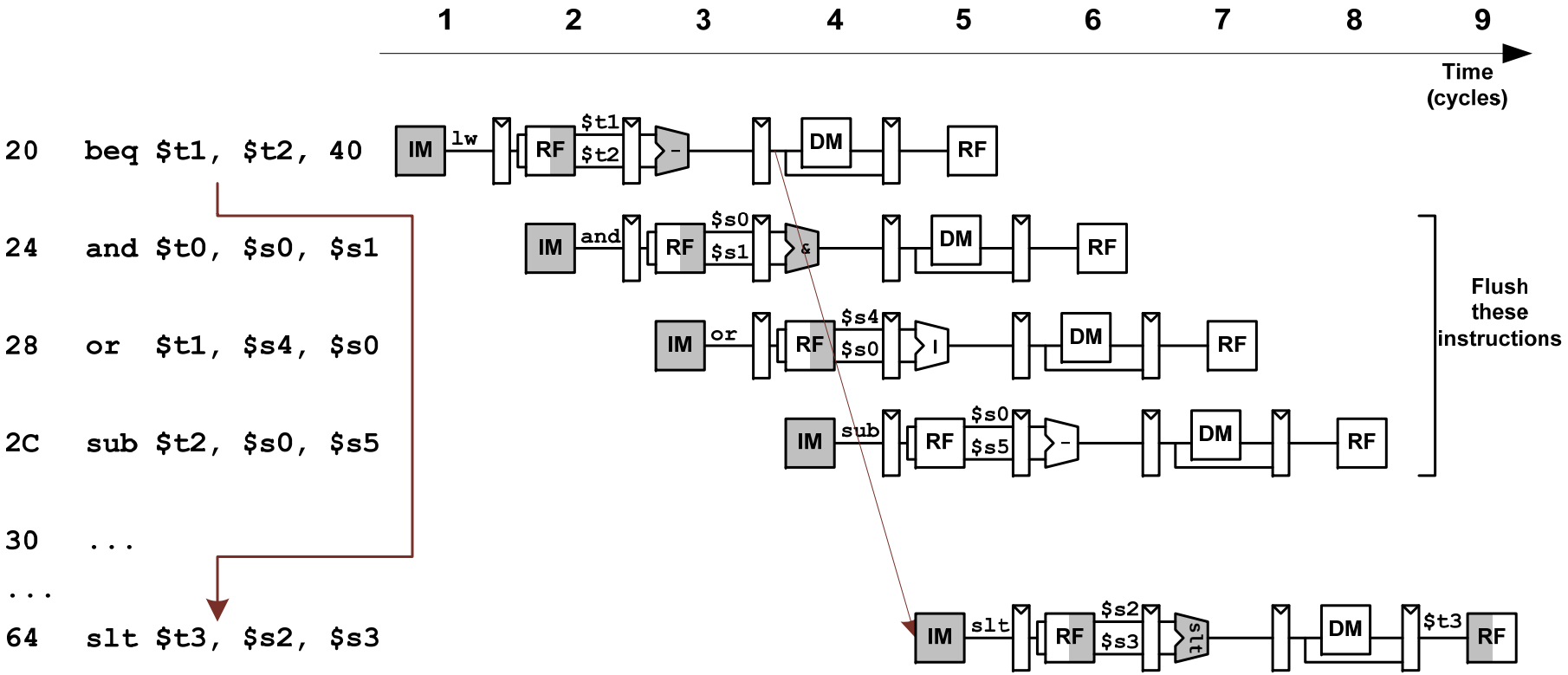
  - **`FlushE`** is used to clear the contents of the EX stage register

# Handling Control Hazards /1

- **`beq`-related Control Hazards**:
  - Branch value not determined until the fourth stage of pipeline
  - Instructions after branch fetched before branch occurs
  - **These instructions must be flushed if branch happens**
- **Could we stall the pipeline?**
  - Possibly yes, but we would have to stall for three cycles, which would be a significant performance penalty
- **Instead, predict whether the branch will be taken**
  - Start executing immediately based on the prediction
  - If the prediction turns out to be wrong, throw out the loaded instructions (called instruction flush) and continue execution

# Handling Control Hazards /2

- **Instructions are flushed when the branch is taken:**

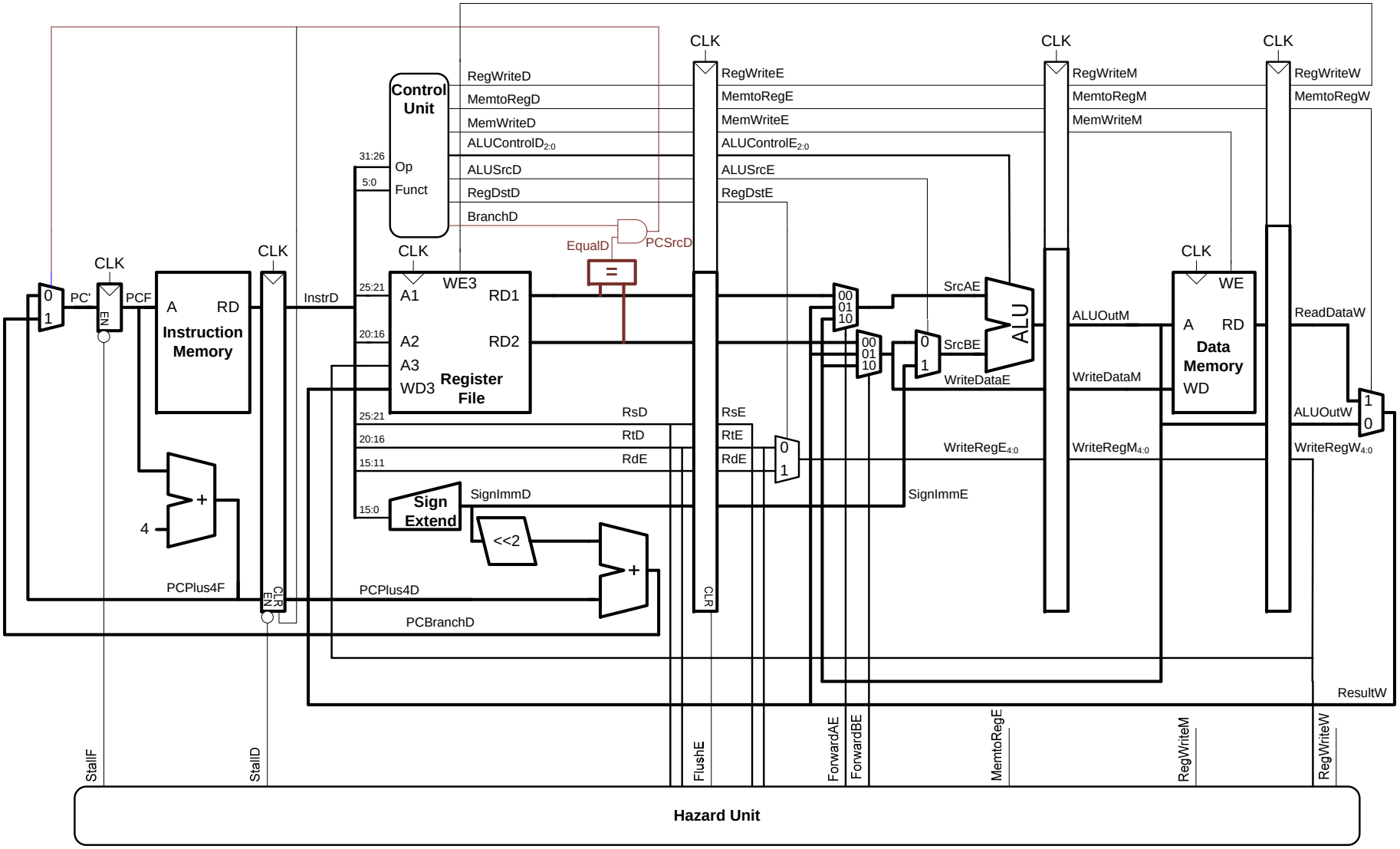


# Handling Control Hazards /3

- **Branch Misprediction Penalty:**
  - Number of instruction flushed when branch is taken
  - May be reduced by determining the branch value earlier
- **Note that deciding whether the branch is taken simply requires an equality comparator**
  - This can be implemented much faster than using a subtraction and zero detection via ALU
  - Adding an equality comparator is also referred to as Early Branch Resolution
  - **By introducing the equality comparator into the ID stage, the branch misprediction penalty can be reduced to one (from three)**

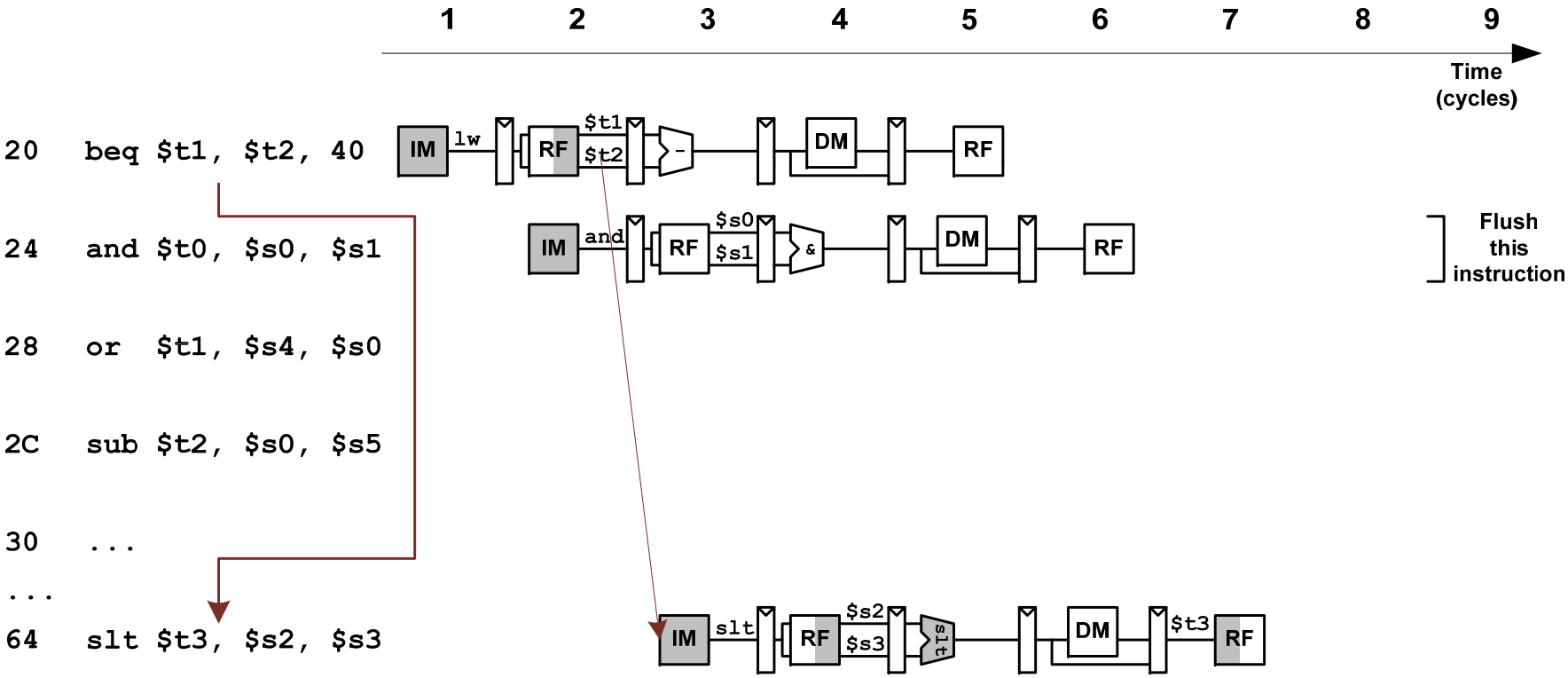
# Handling Control Hazards /4

- **Early Branch Resolution**:



# Handling Control Hazards /5

- **Early Branch Resolution**:
  - Lowered the number of instructions that need to be flushed

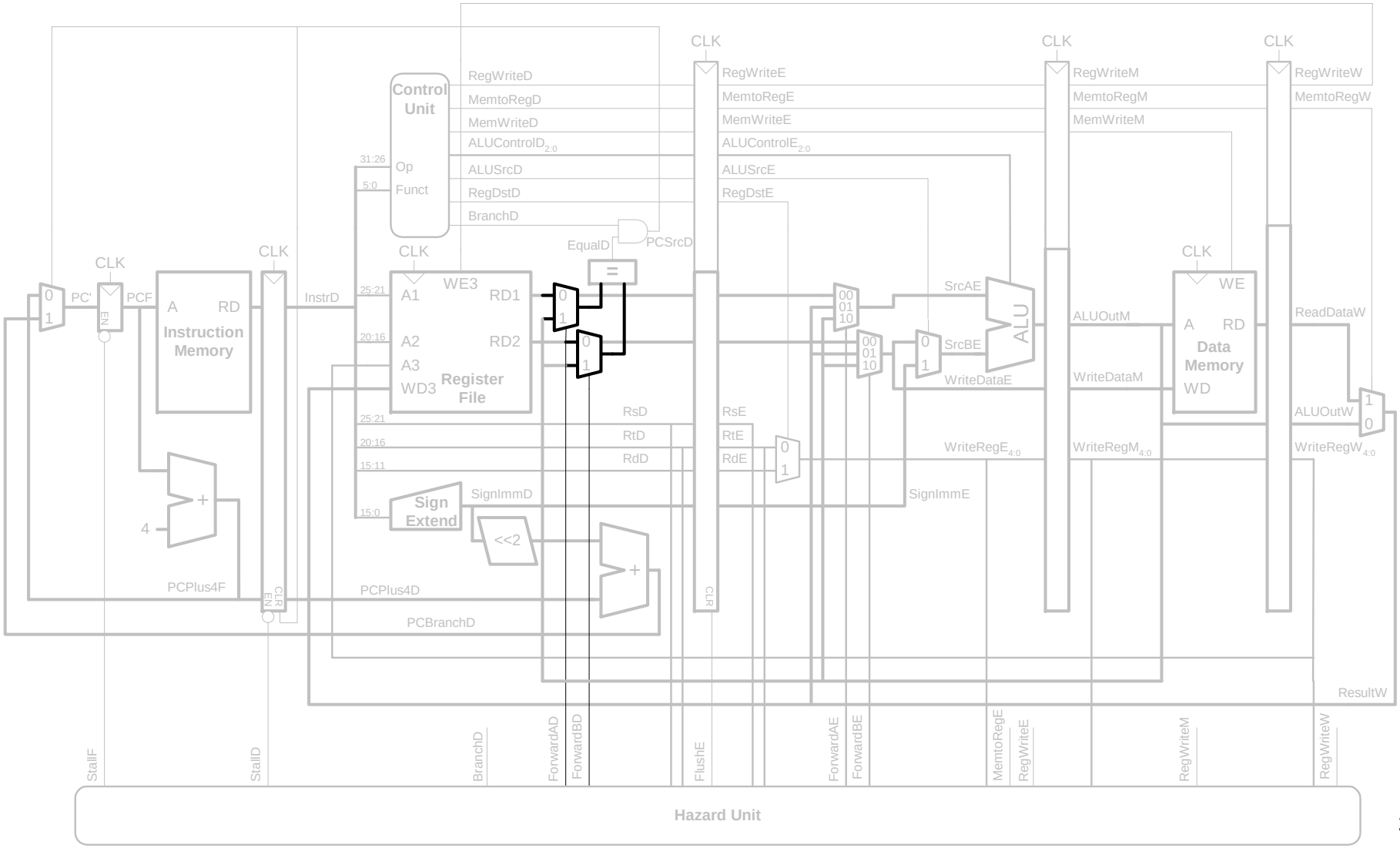


# Handling Control Hazards /6

- **Handling Data and Control Hazards Together:**
  - Early branch resolution introduces a RAW hazard
  - That is, if one of the source operands for the branching is being written by the previous instruction, it may not be ready yet
  - For the WB stage, the hazard will not apply since the write will occur in the first part of the cycle and read in the second
  - For the MEM stage of an ALU instruction, we could forward the needed value to the new equality comparator via multiplexors
    - Not using the memory unit, so we can freely forward the value
  - For the EX stage of an ALU instruction or MEM stage of a lw instruction, we will need to delay (stall) the pipeline until the result is ready
    - Using the matching element in each option, so we need to stall
  - **The special kind of stall that is introduced here is called the branch stall**

# Handling Control Hazards /7

- **Handling Data and Control Hazards Together:**



# Handling Control Hazards /8

- **Handling Data and Control Hazards Together:**
  - **Forwarding Logic:**

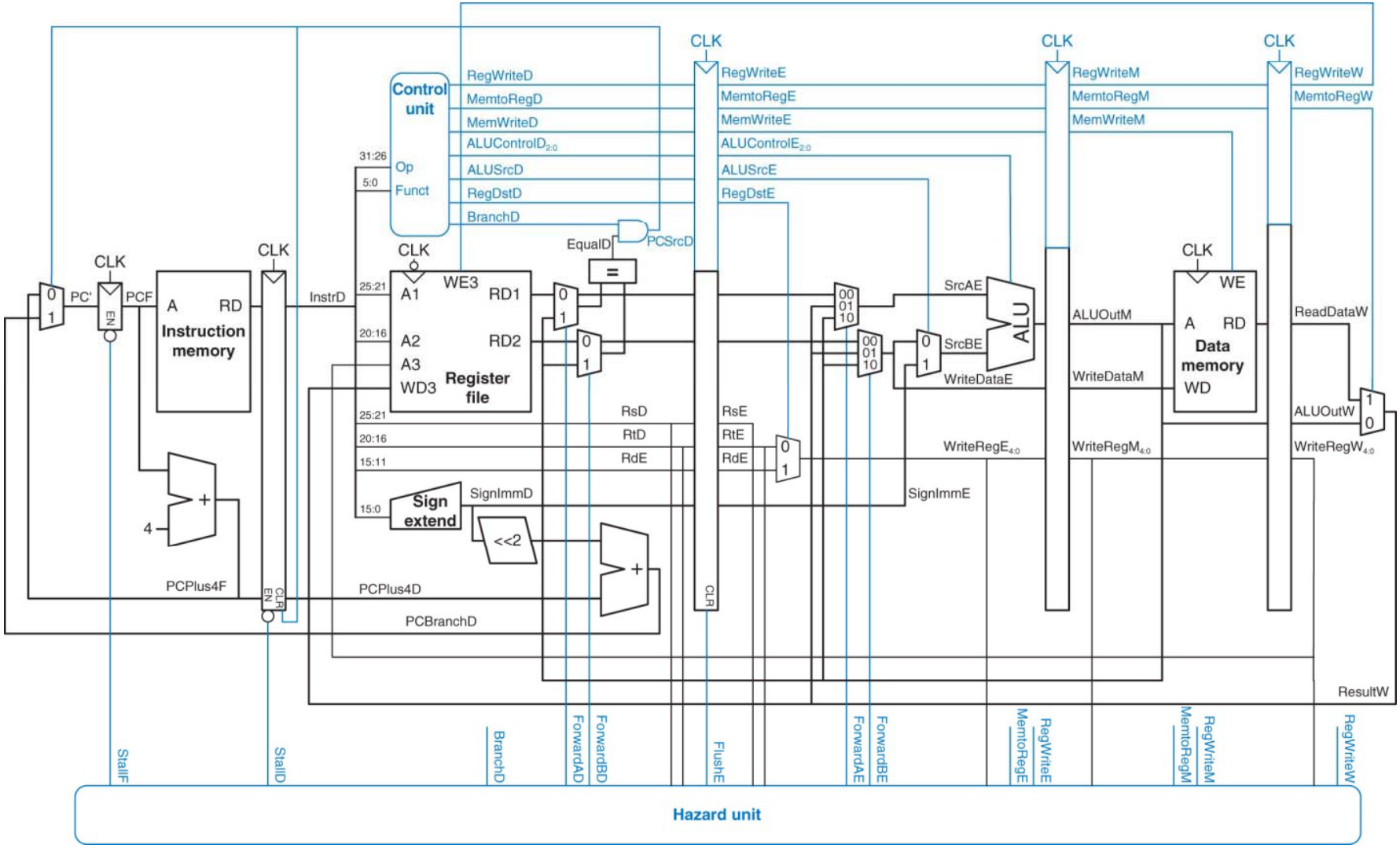
```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND
       RegWriteM

ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND
       RegWriteM
```

  - **Stalling Logic:**

```
branchstall = (BranchD AND RegWriteE AND
      (WriteRegE == rsD OR WriteRegE == rtD))
           OR
        (BranchD AND MemtoRegM AND
       (WriteRegM == rsD OR WriteRegM == rtD))
StallF = StallD = FlushE = lwstall OR branchstall
```

# Handling Control Hazards /9

- **Final Pipelined MIPS Datapath Design:**



# Handling Control Hazards /10

- **Branch prediction to improve average CPI:**
  - Ideal pipelined processor: CPI = 1
  - Branch misprediction increases the average CPI
- **Static branch prediction:**
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Else, predict not taken
- **Dynamic branch prediction:**
  - **Keep the history of the last several hundred branches in the branch target buffer, and record:**
    - Branch destination
    - Whether the branch was taken
  - Good prediction reduces fraction of branches requiring a flush

# Handling Control Hazards /11

- **Branch Prediction Example:**

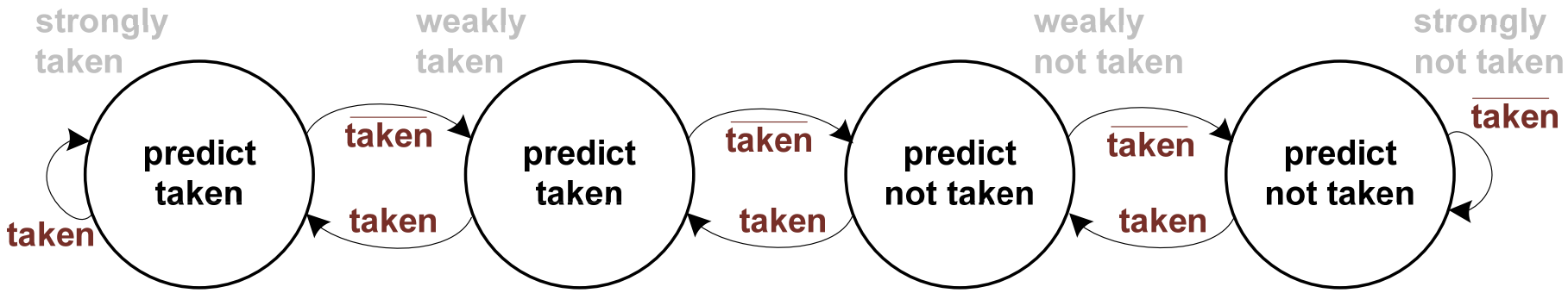
```
add  $s1, $0, $0       # sum = 0
add  $s0, $0, $0       # i   = 0
addi $t0, $0, 10       # $t0 = 10
for:
  beq  $s0, $t0, done   # if i == 10, branch
  add  $s1, $s1, $s0    # sum = sum + i
  addi $s0, $s0, 1      # increment i
  j    for
done:
```

**The loop continues until i == 10**

- **Simple one-bit dynamic branch predictor:**
  - Check if the last branch was taken
  - Repeat the same action from the previous cycle
  - **One-bit predictor is simple to design, but it mispredicts the first and the last branch of each loop**

# Handling Control Hazards /12

- **Two-bit branch predictor:**
  - Introduces four states: strongly taken, weakly taken, weakly not taken, and strongly not taken
  - Changes the next action after two mispredictions, or two correct predicitions, respectively
  - **Two-bit predictor mispredicts the last branch of a loop**



# Pipelined Performance Example /1

- **Example:**
  - 40% of loads used by the next instruction
  - 25% of branches mispredicted
  - All jumps flush the next instruction
  - Also, 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions are executed
- **What is the average CPI?**
  - Load/Branch CPI = 1 when no stalling, 2 when stalling
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$
  - Average CPI =

    (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) =

    **Average CPI = 1.15**

# Pipelined Performance Example /2

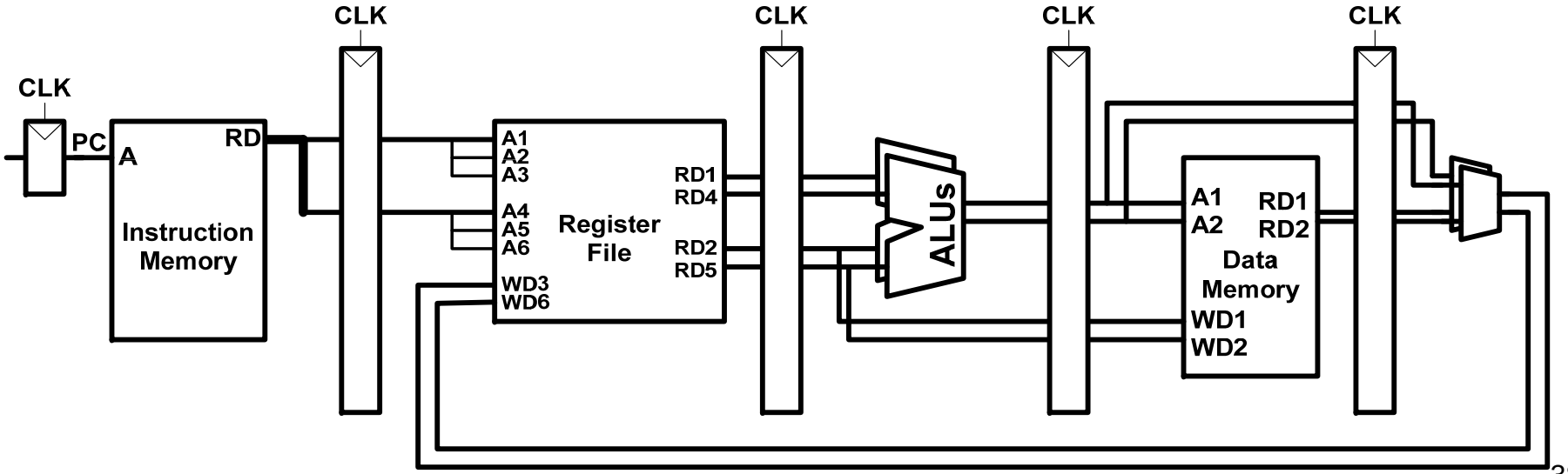
- **Sample values for the clock-cycle times ($T_c$):**
  - Single-cycle processor $T_c$: 925 ps
  - Single-cycle processor CPI: 1
  - Multi-cycle processor $T_c$: 325 ps
  - Multi-cycle processor CPI: 4.04
  - **Pipelined processor $T_c$: 550ps**
  - **Pipelined processor CPI: 1.15**
- **So which of the three processor designs will perform faster when executing 100 billion gcc instructions?**
  - Single-cycle processor execution time = 92.5 seconds
  - Multi-cycle processor execution time = 131.3 seconds
  - Pipelined processor execution time =

    # of instructions × CPI × $T_c$ = $(100 \times 10^9) \times 1.15 \times 550 \times 10^{-12}$

    **= 63.25 seconds**

# Advanced MIPS Pipeline Design /1

- **Superscalar Datapath:**
  - Multiple copies of the pipeline datapath execute multiple instructions at once
  - The datapath fetches two instructions at one time
  - It contains a register file with six input register ports, two ALUs, and a data memory unit with two input address ports
  - **Dependencies between instructions may make it tricky to issue multiple instructions at once**



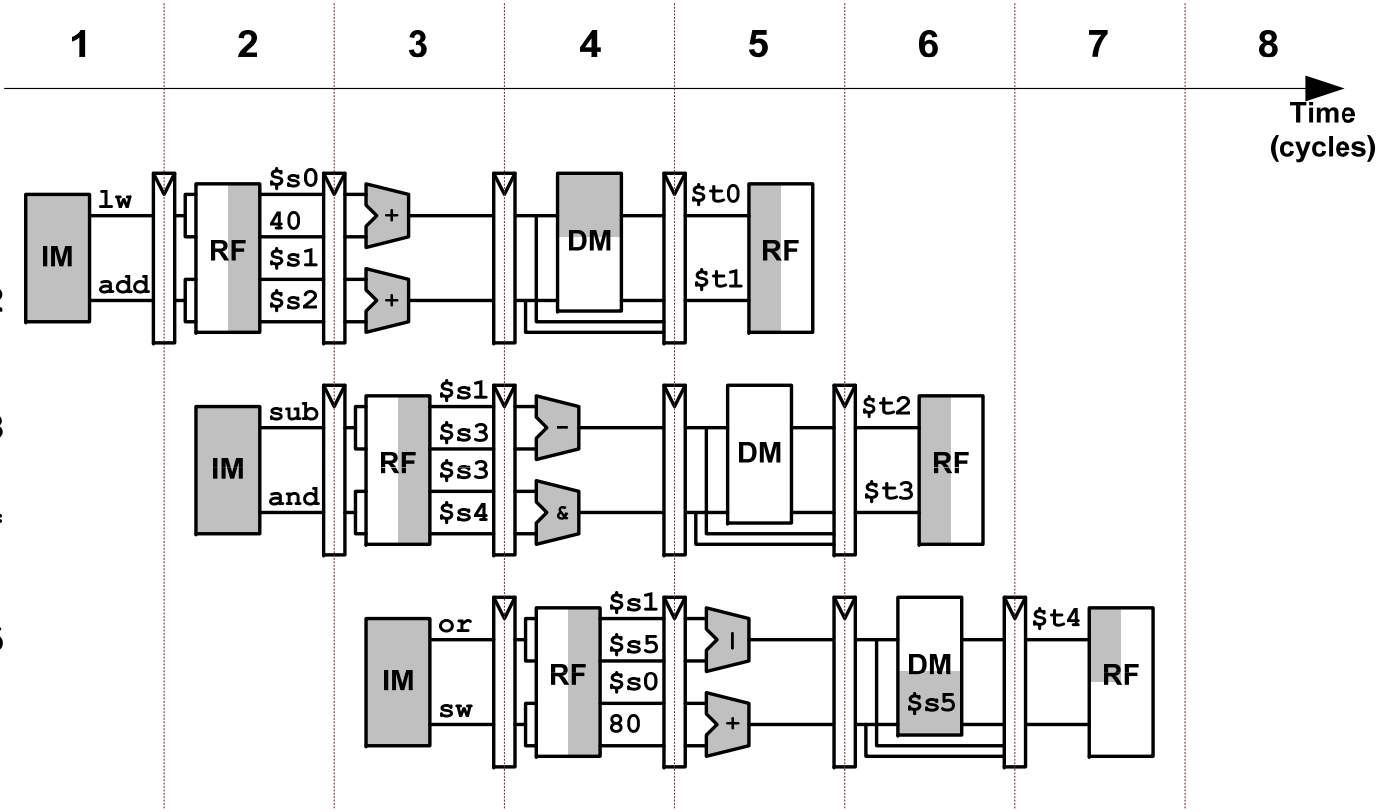
# Advanced MIPS Pipeline Design /2

- **Superscalar Datapath Example:**

IPC value refers to the number of instructions issued per cycle

Desired IPC: 2
Actual IPC: 2

```
lw  $t0, 40($s0)
add $t1, $s1, $s2

sub $t2, $s1, $s3
and $t3, $s3, $s4

or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

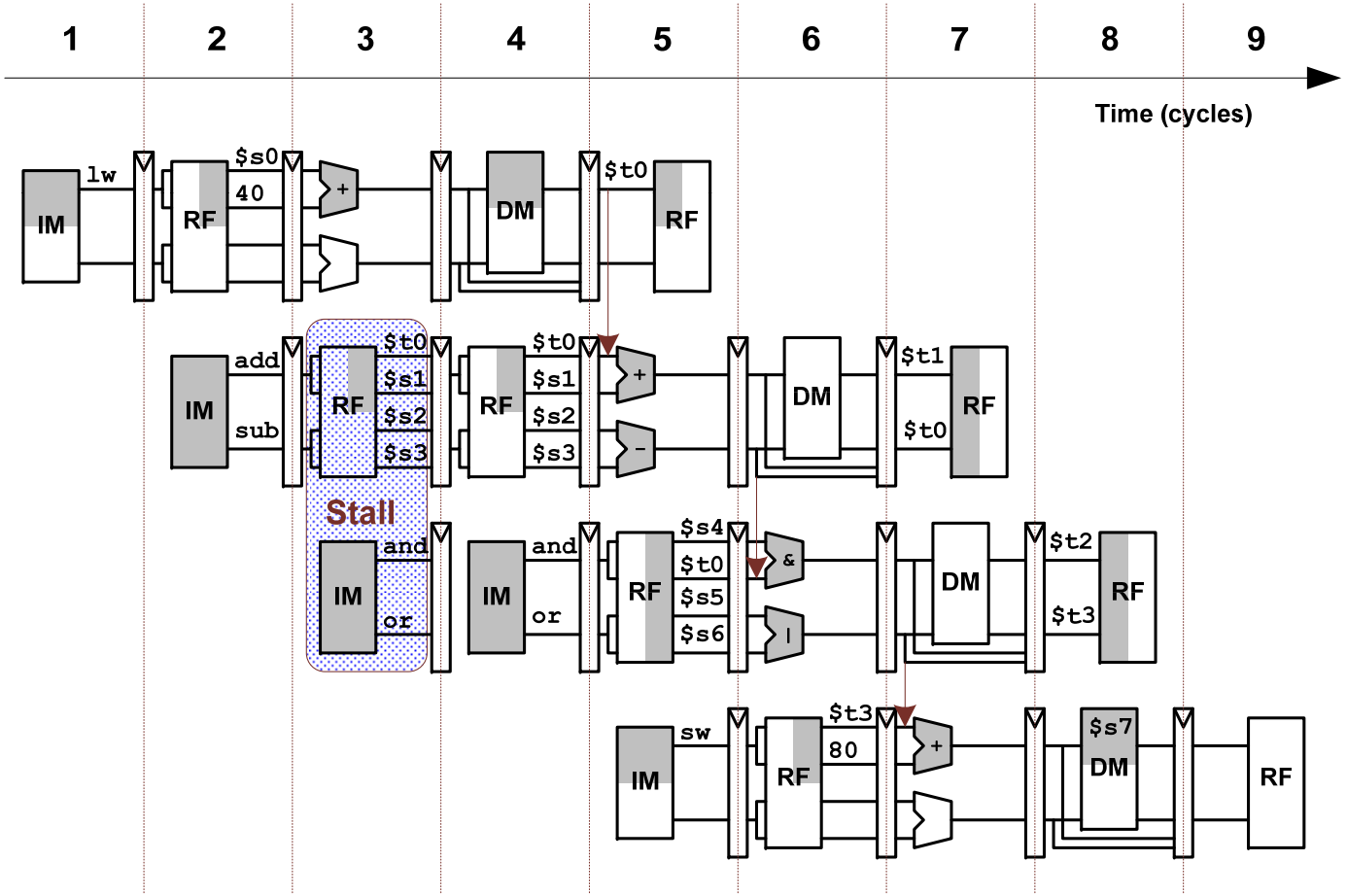


# Advanced MIPS Pipeline Design /3

- **Superscalar Datapath Example with Dependencies:**

**Desired IPC: 2**
**Actual IPC:**
**6/5 = 1.17**

```
lw  $t0, 40($s0)

add $t1, $t0, $s1
sub $t0, $s2, $s3

and $t2, $s4, $t0
or  $t3, $s5, $s6

sw  $s7, 80($t3)
```



# Advanced MIPS Pipeline Design /4

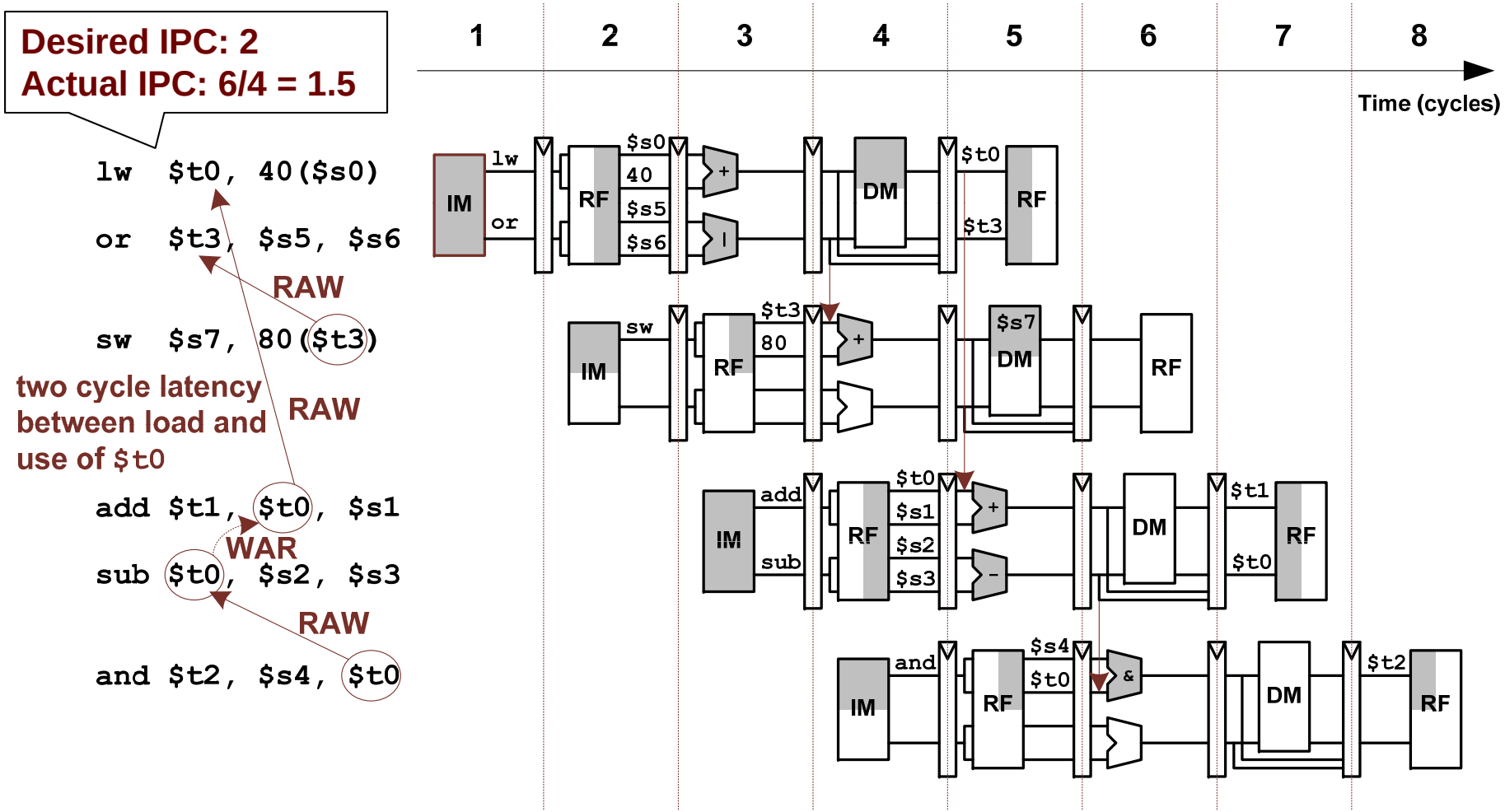
- **Deep Pipelining:**
  - Split the pipeline into more stages
  - Each stage requires less logic to run, so it runs faster
  - It is typical that 10 to 20 stages are used
    - We have covered the classical 5-stage MIPS pipeline
  - The maximum number of pipelines is limited by additional dependencies between instructions, sequencing overhead, increased power requirements, and increased cost
- **Additional stages introduce additional hazards**
  - Increasing the number of stages increases the number of additional dependencies between instructions, which can lead to additional data and control hazards
  - Some hazards can be resolved by forwarding, but some may require stalls, which increases the average CPI

# Advanced MIPS Pipeline Design /5

- **Out-of-Order Processor:**
  - Looks ahead across multiple instructions
  - Issues as many instructions as possible at once
  - Issues instructions out of order as long as all dependencies are not violated
- **Dependencies to consider:**
  - **RAW (read after write):** One instruction writes, later instruction reads a register before the first instruction finished writing it
    - Occurs in the classical MIPS pipeline; essential hazard
  - **WAR (write after read):** One instruction reads, later instruction writes over a value before the intended instruction has read it
    - Also called anti-dependence; non-essential hazard
  - **WAW (write after write):** One instruction writes, later instruction is moved forward so it writes over the written value
    - Also called output dependence; non-essential hazard

# Advanced MIPS Pipeline Design /6

- **Out-of-Order Processor Example:**



# Advanced MIPS Pipeline Design /7

- **Out-of-Order Processor Implementation:**
  - **Use a table (called scoreboard) that keeps track of:**
    - Instructions waiting to issue
    - Available functional units
    - Dependencies
  - The processor executes the maximum number of instructions based on the current status of the scoreboard
- **Instruction level parallelism (ILP):**
  - Theoretical ILP can be quite large for architectures with many architectural elements and very accurate branch predictors
  - **In practice however, the number of instructions that can be issued simultaneously is on average less than 3**

# Advanced MIPS Pipeline Design /8

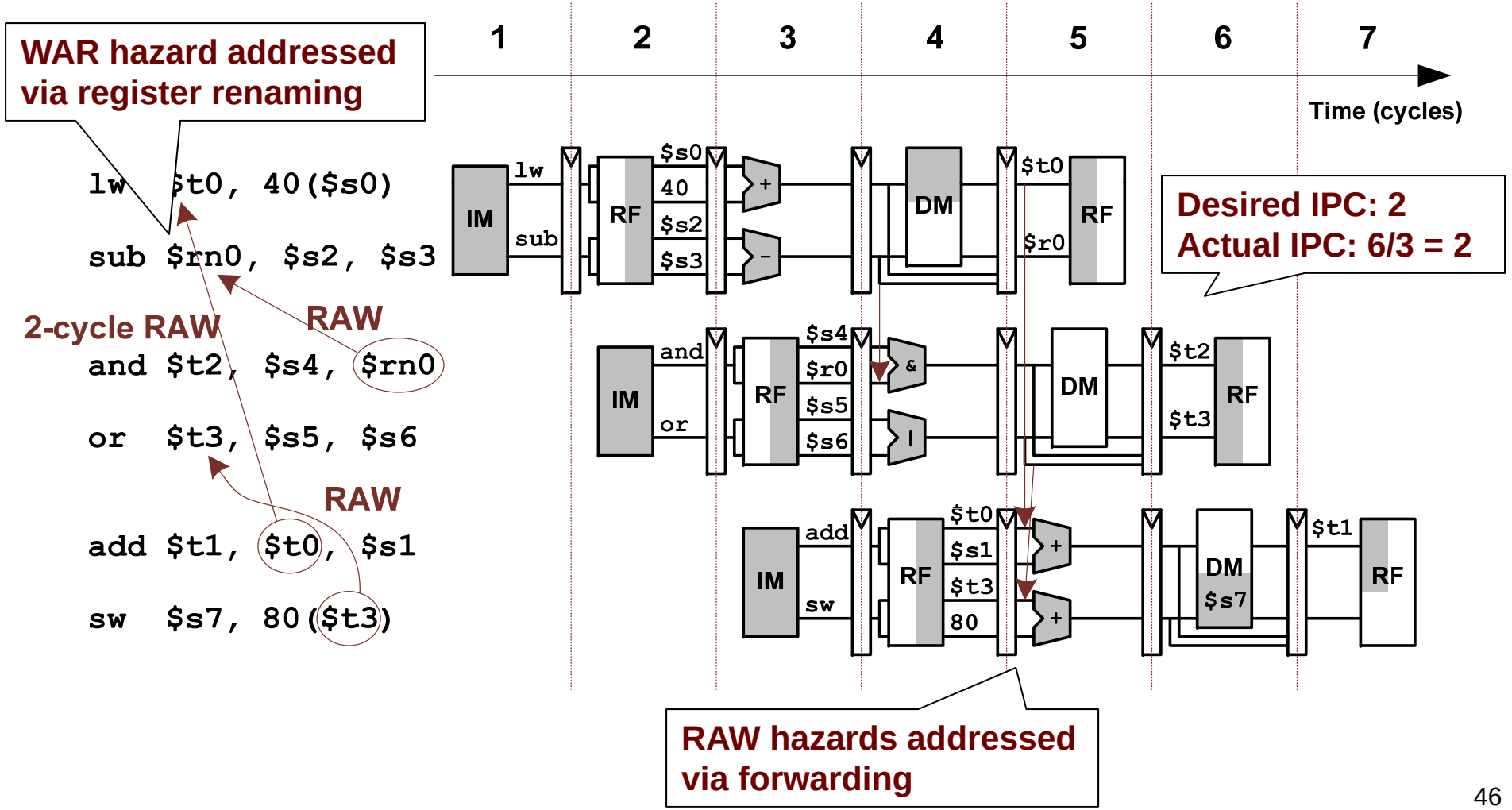
- **Out-of-Order Processor Register Renaming:**
  - Add renaming registers to the datapath
  - For instance, the datapath may include additional renaming registers called $rn0 - $rn19
  - **These registers cannot be used directly, but can be used to resolve hazards**
  - For instance, if $t0 was used in two conflicting instructions, $t0 could be renamed to $rn0 in one of the two instructions
- **Register Renaming Example:**

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

**WAR hazard may be introduced**

# Advanced MIPS Pipeline Design /9

- **Out-of-Order Processor Register Renaming:**



# Advanced MIPS Pipeline Design /10

- **Process:**
  - A program running on a computer
  - Multiple processes can run at once
  - For example, surfing Web, playing music, writing a paper
- **Thread:**
  - A unit of program execution
  - Each process has multiple threads
  - For example, a word processor may have threads for typing, spell checking, printing
- **Multiprocessors:**
  - Multiple processors (cores) with a method for communication between them (more on this later in the course)

# Advanced MIPS Pipeline Design /11

- **Threads in a conventional processor:**
  - One thread runs at one time
  - When one thread stalls (for example, waiting for memory):
    - Architectural state of that thread is stored
    - Architectural state of a waiting thread is loaded into the processor and that thread is then run
    - **This process is called context switching**
  - Appears to a user like threads are running simultaneously

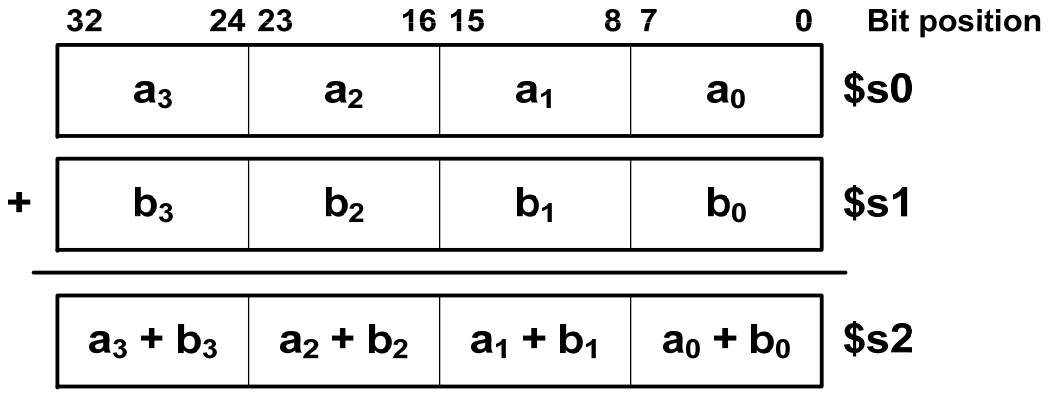
# Advanced MIPS Pipeline Design /12

- **Multithreading:**
  - Multiple copies of the architectural state
  - Multiple threads active at once:
    - When one thread stalls, another runs immediately
    - If one thread cannot keep all execution units busy, another thread can use them
  - **Does not increase instruction-level parallelism (ILP) of a single thread, but it does increase throughput**
  - Intel refers to this process as “hyper-threading”

# Advanced MIPS Pipeline Design /13

- **Single Instruction Multiple Data (SIMD):**
  - One SIMD instruction acts on multiple pieces of data at once
    - Typically implemented as a coprocessor (e.g., ARM core)
  - Common application: graphics
  - Perform short arithmetic operations
    - Also called packed arithmetic
- **For example, add four 8-bit elements:**
  - Modify the ALU to ignore carry-outs between packets

```
padd8 $s2, $s0, $s1
```

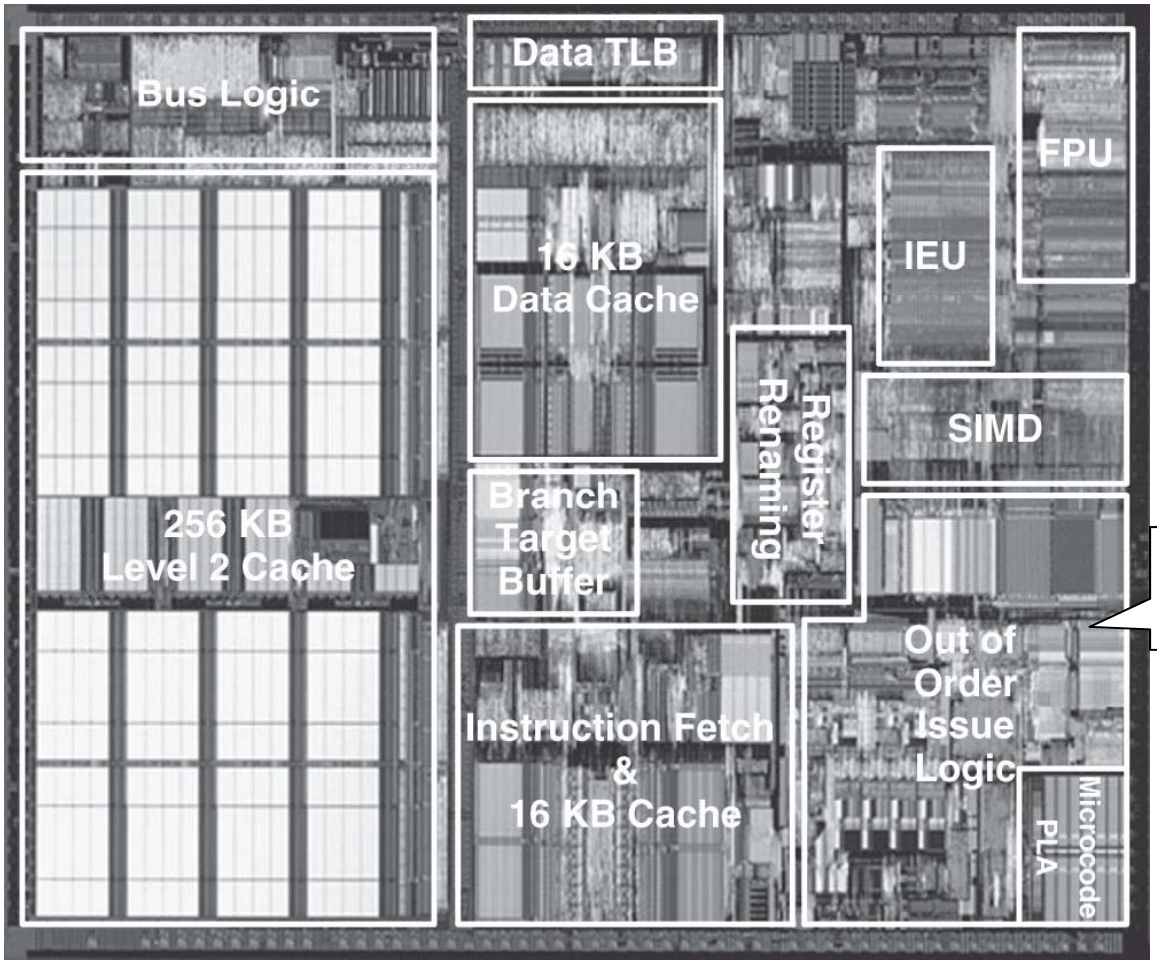


# Advanced MIPS Pipeline Design /14

- **More on Single Instruction Multiple Data (SIMD):**
  - Quite useful when dealing with data elements that are smaller than 32 bits
  - For instance, a pixel may store 8-bit information about a red, green, or blue component at one time
  - Using an entire 32-bit value to perform 8-bit arithmetic would be wasteful (the upper 24 bits would go unused)
  - **Instead, package four 8-bit packets into a 32-bit word, and perform arithmetic on multiple values at one time**
  - Double-value words (64-bit words) can be even more helpful since those can be used to package eight 8-bit data elements, or four 16-bit data elements
  - SIMD instructions can also be used in floating-point arithmetic

# Visualizing Computer Architectures /1

- **Pentium III Microprocessor:**



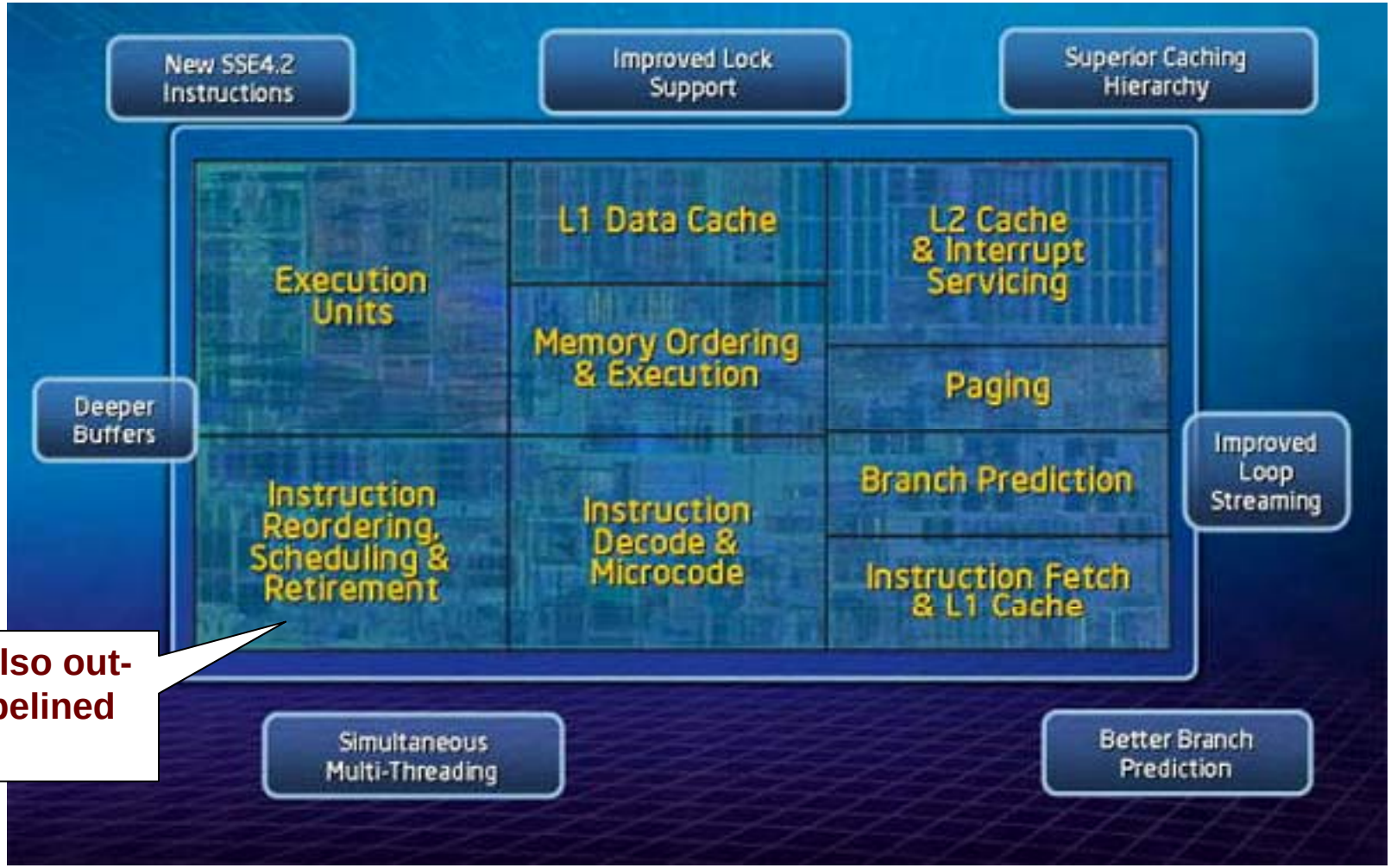

**P III was an out-of-order pipelined processor**

Source: "Digital Design and Computer Architecture" by Harris and Harris, Chapter 7

(read the textbook for other processor architectures and their descriptions)

# Visualizing Computer Architectures /2

- **Core i7 Microprocessor Core:**

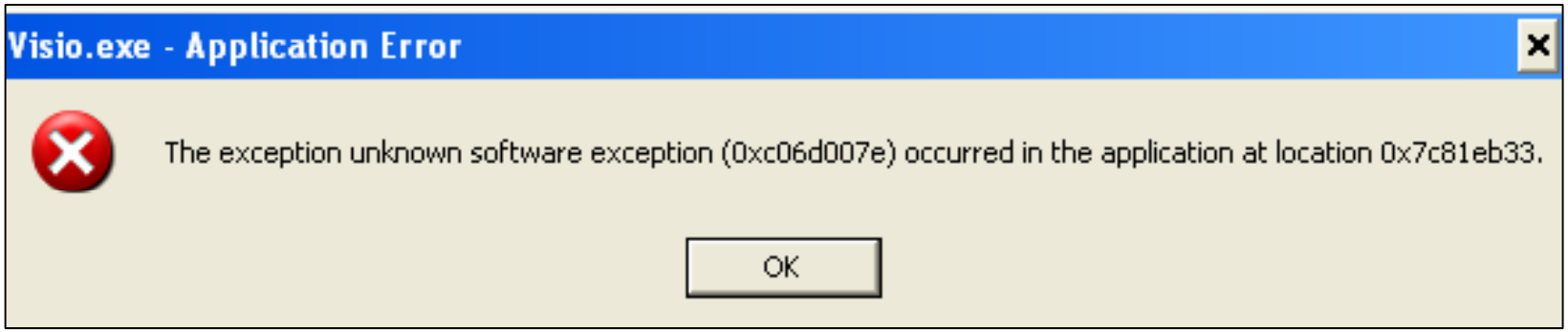



**i7 Core is also out-of-order pipelined processor**

Source: Intel Core i7 Processor Technical documents

# Handling Exceptions /1

- **What are Exceptions?**
  - Unscheduled function call to an exception handler
- **Caused by:**
  - Hardware (also called interrupt), such as keyboard interrupts
  - Software (also called traps), such as undefined instruction
- **When exception occurs, the processor:**
  - Records cause of exception (Cause register)
  - Jumps to exception handler
  - Returns to program (EPC register)



# Handling Exceptions /2

- **Exception registers are not part of the register file**
  - **`Cause Register`**
    - Records cause of exception
    - Coprocessor 0, register 13
  - **`EPC (Exception PC) Register`**
    - Records PC where exception occurred
    - Coprocessor 0, register 14
- **Move from Coprocessor 0**
  - **`mfc0 $t0, Cause`**
  - Moves contents of **`Cause`** into **`$t0`**

**mfc0**

| 010000 | 00000 | $t0 (8) | Cause (13) | 00000000000 |
|---|---|---|---|---|
| 31:26 | 25:21 | 20:16 | 15:11 | 10:0 |

# Handling Exceptions /3

- **Exception Causes:**

| Exception | Cause |
|---|---|
| Hardware Interrupt | 0x00000000 |
| System Call | 0x00000020 |
| Breakpoint / Divide by 0 | 0x00000024 |
| **Undefined Instruction** | 0x00000028 |
| **Arithmetic Overflow** | 0x00000030 |

# Food for Thought

- **Download and Read Assignment #3 Specifications**
- **Read:**
  - Chapter 6 from the Course Notes
    - Review the material discussed in the lecture notes in more detail
    - Our course schedule follows the material in the Course Notes
  - Recommended: Chapter 7 from the Harris and Harris textbook

  OR
  - Recommended: Chapter 4 from the course textbook