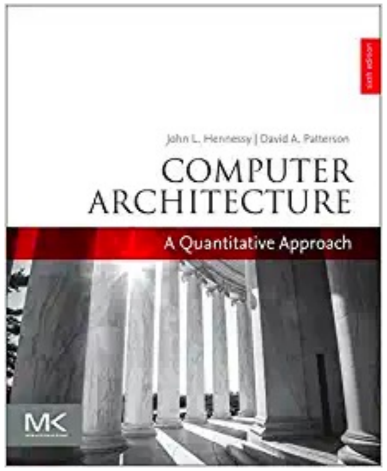


Chapter 3

Instruction-Level Parallelism and Its Exploitation

Part 1: A Brief Review

"Who's first?"
"America."
"Who's second?"
"Sir, there is no second."
-Dialog between two observers of the sailing race later named "The America's Cup" and run every few years -- the inspiration for John Cocke's naming of the IBM research processor as "America." This processor was the precursor to the RS/6000 series and the first superscalar microprocessor.

Acknowledgements

- Thanks to many sources for slide material

  © 1990 Morgan Kaufmann Publishers, © 2001-present Elsevier
  Computer Architecture: A Quantitative Approach by J. Hennessy & D. Patterson

  © 1994 Morgan Kaufmann Publishers, © 2001-present Elsevier
  Computer Organization and Design by D. Patterson & J. Hennessy

  © 2002 K. Asinovic & Arvind, MIT

  © 2002 J. Kubiatowicz, University of California at Berkeley

  © 2006, © 2010 No Starch Press for Inside the Machine by J. Stokes

  © 2007 W.-M. Hwu & D. Kirk, University of Illinois & NVIDIA

  © 2007-2010 J. Owens, University of California at Davis

  © 2010 CRC Press for Introduction to Concurrency in Programming Languages by M. Sottile, T. Mattson, and C. Rasmussen

  © 2017, IBM POWER9 Processor Architecture by Sadasivam et al., IBM

  © 2016, © 2019 POWER9 Processor User's Manual, IBM

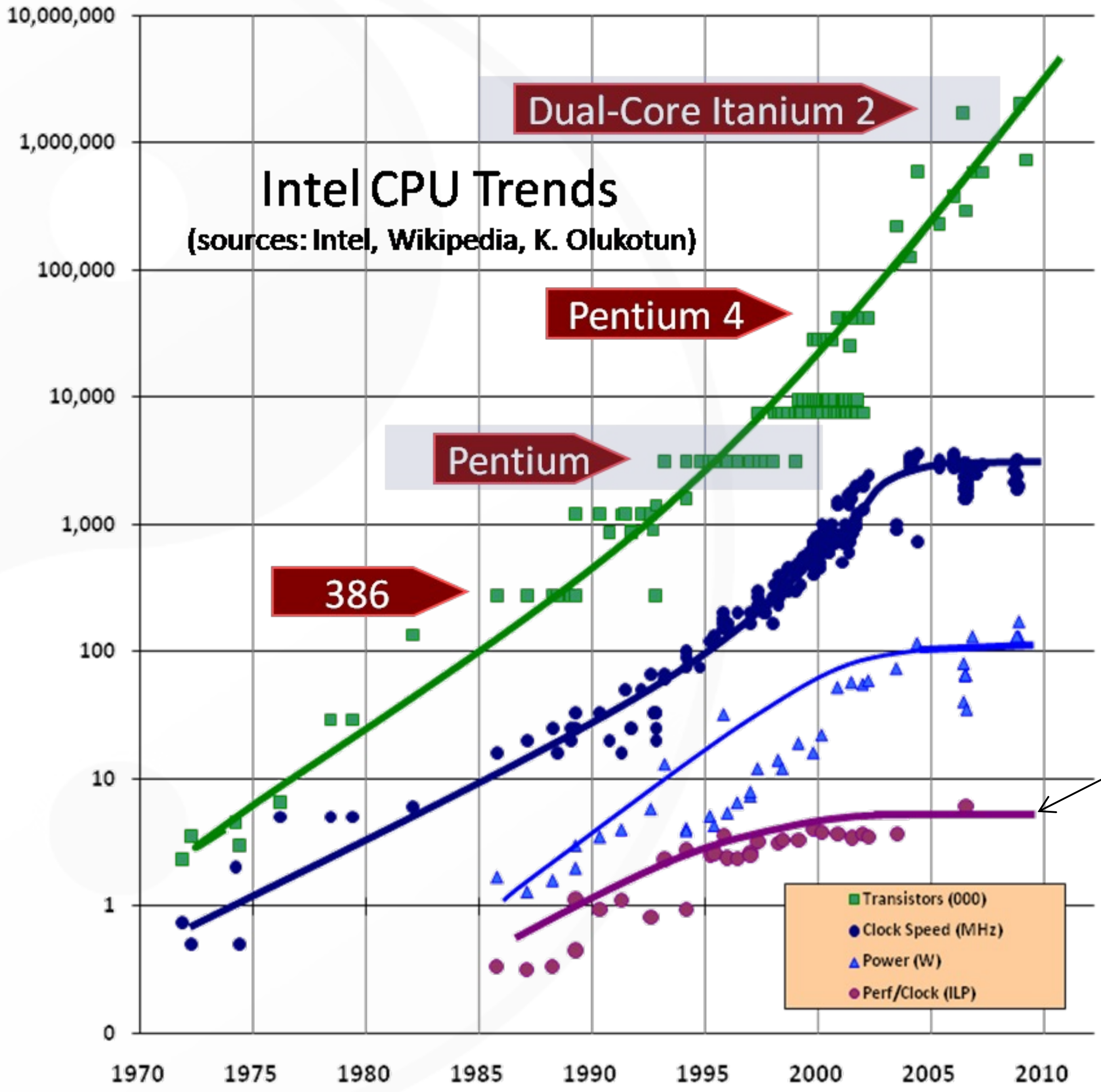
  © The OpenPOWER Foundation

  © 2022, W. Feng, Virginia Tech

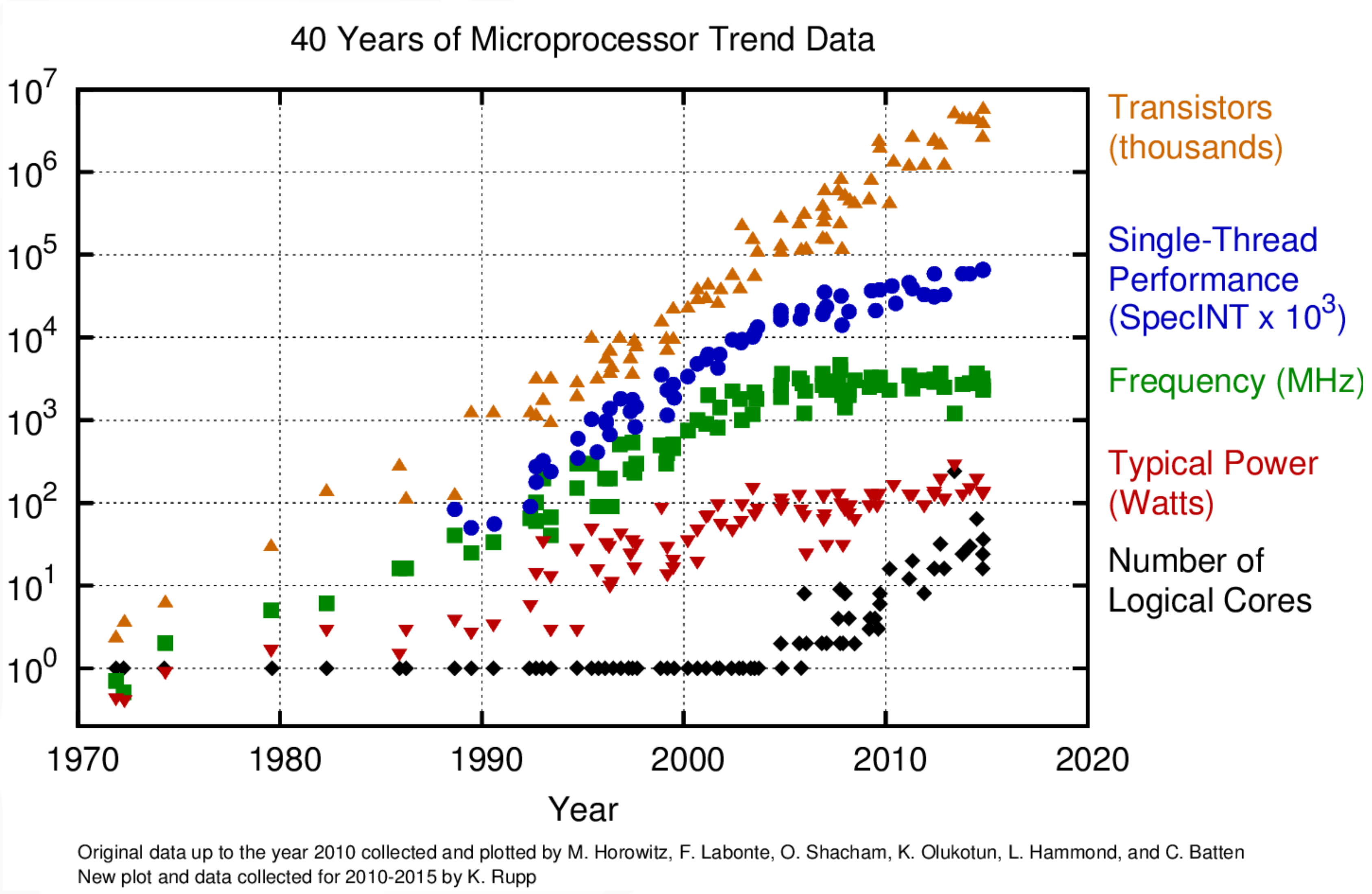
Introduction

- What is instruction-level parallelism (ILP)?
  - A measure of how many of the operations in a computer program can be performed simultaneously.
  - Pipelining → universal technique in 1985
    - Overlaps execution of instructions → exploits ILP
- How do processors extract ILP to get faster?
  - More parallelism (or more work per pipeline stage): fewer clocks/instruction [more instructions/cycle]
    - Get WIDER
  - Deeper pipeline stages: fewer gates/clock
    - Get DEEPER
  - Transistors get faster (Moore's Law): fewer ps/gate
    - Get FASTER
- What do processors do to extract ILP? Later …

Does a faster processor mean a faster computer?


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
🟩 Transistors (000)
● Clock Speed (MHz)
▲ Power (W)
● Perf/Clock (ILP)

ILP



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

https://www.karlrupp.net/wp-content/uploads/2015/06/40-years-processor-trend.png

Instruction-Level Parallelism (ILP)

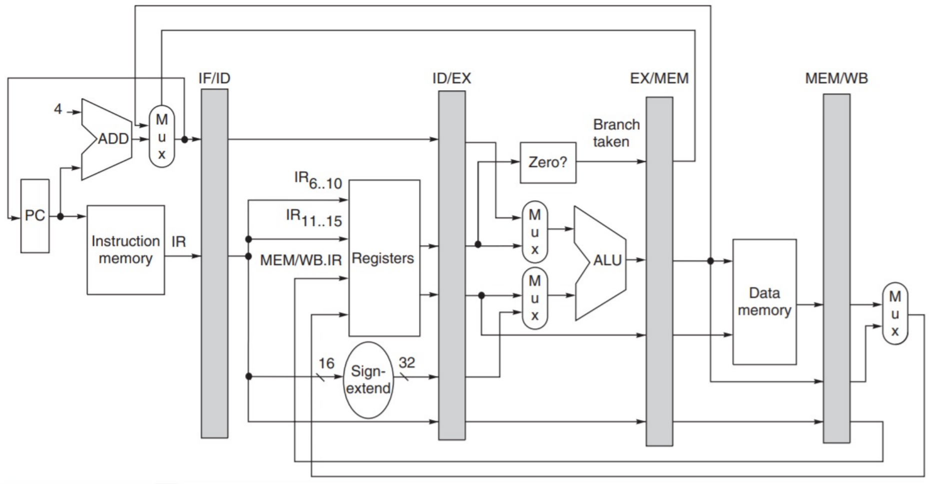
- Goal: *Minimize* CPI (or *maximize* IPC)
  - Pipeline CPI =

    Ideal pipeline CPI +

    Structural-hazard stalls +

    Data-hazard stalls +

    Control-hazard stalls

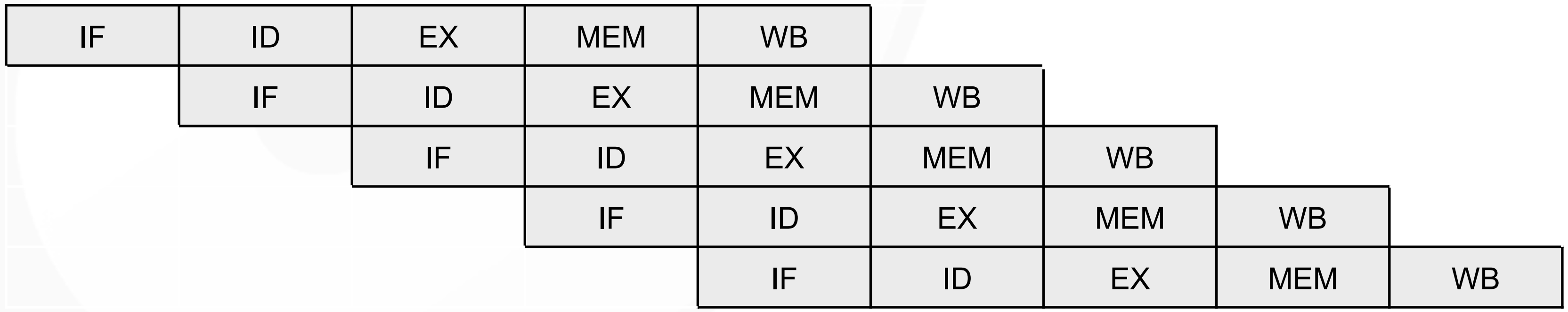


Recall

- CPI = cycles per instruction
  = number of clock cycles required to execute the program / number of instructions executed in running the program
- IPC = instructions per cycle = number of instructions executed while running a program / number of clock cycles required to execute the program

Extracting Yet More Performance

- How do processors extract ILP to get faster?
  - **More parallelism (or more work per pipeline stage): fewer clocks/instruction [more instructions/cycle]**
    - **Get WIDER → instruction width**
  - **Deeper pipeline stages: fewer gates/clock**
    - **Get DEEPER → instruction depth**
  - Transistors get faster (Moore's Law): fewer ps/gate
    - Get FASTER



Extracting Yet More Performance

- Architectural Options: GET WIDER or GET DEEPER
  - Fetch (and execute) more than one instruction at one time (expand every pipeline stage to accommodate multiple instructions) — *multiple-issue* → *superscalar or VLIW*
    - How does this help performance?
    - What does it impact in the performance equation?
  - Increase the depth of the pipeline to increase the clock rate — *superpipelining*
    - How does this help performance? (What does it impact in the performance equation?)

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Basic MIPS *Pipelined* Architecture

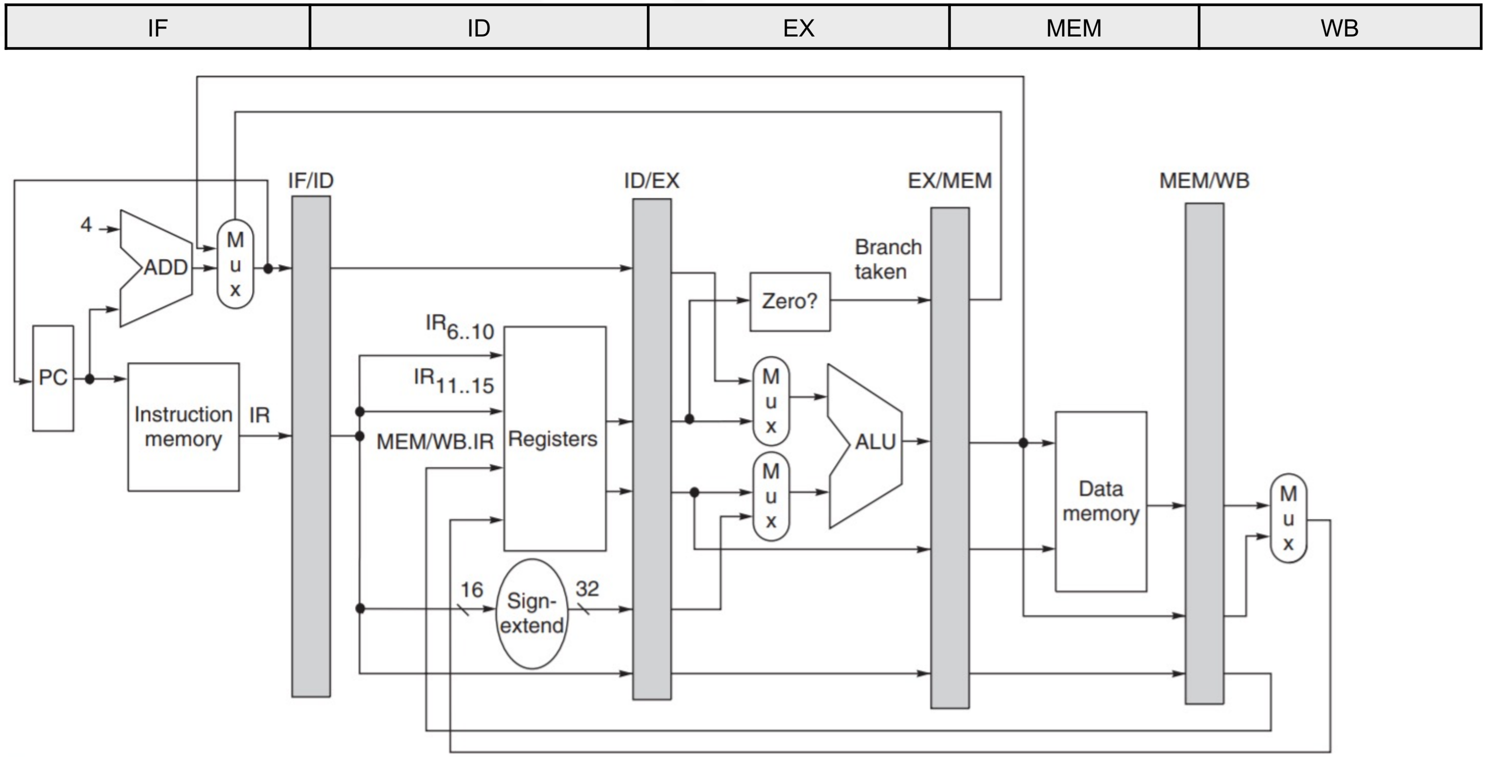
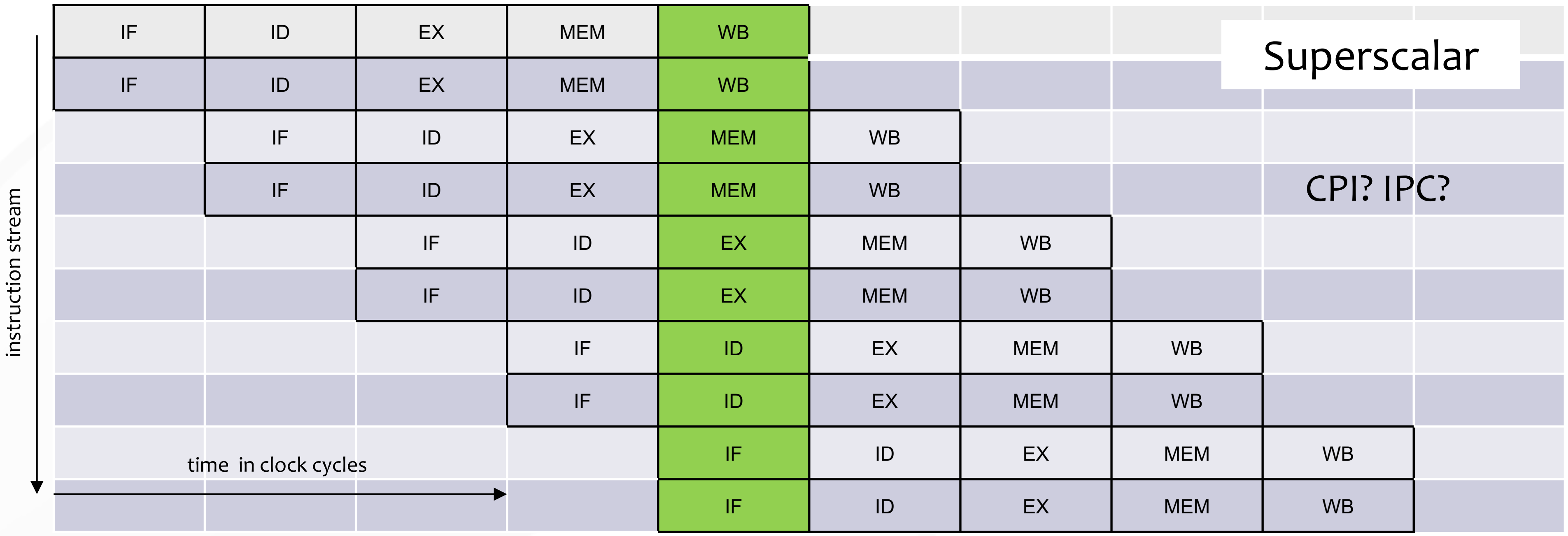


Figure C.22 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence, there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, since two instructions would try to write different values into the PC. Most of the data paths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline.

Superscalar

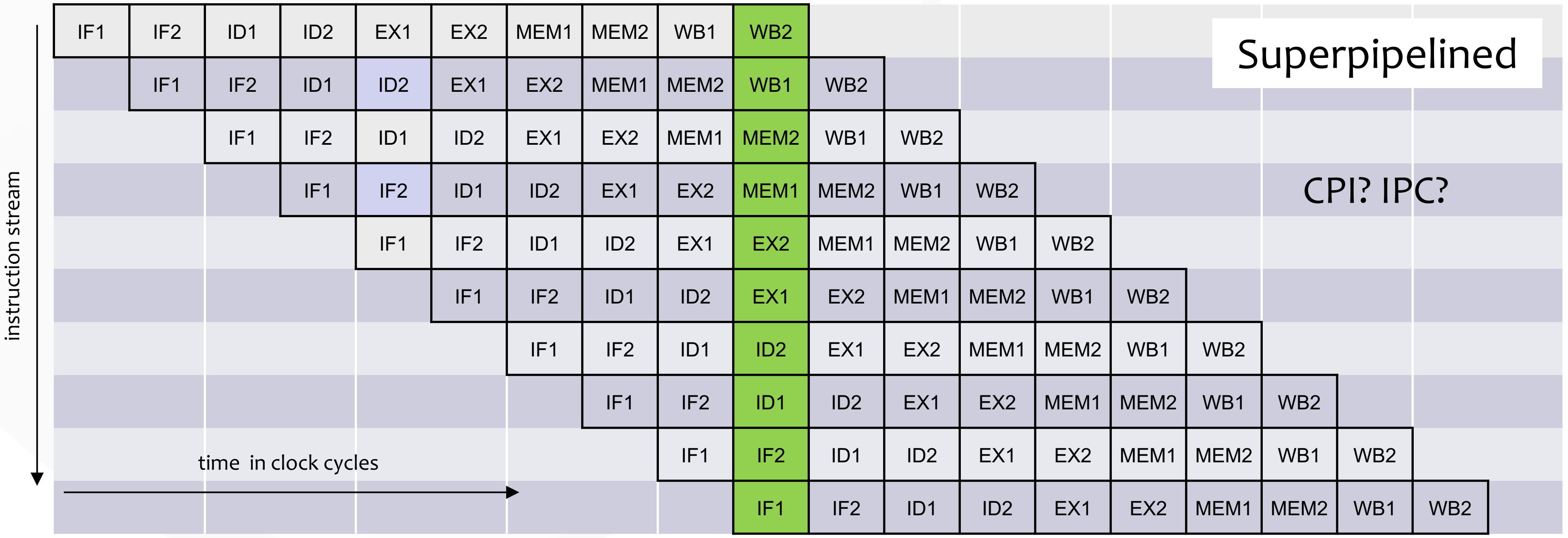
CPI? IPC?



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Superpipelined

CPI? IPC?



IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2									
	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2								
		IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2							
			IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2						
				IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2					
					IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2				
						IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2			
							IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2		
								IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2	
									IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2

N. Jouppi, "Superscalar vs. Superpipelined Machines," ACM SIGARCH Computer Architecture News, 16(3):71-80, June 1988.

Superscalar Processors

- Launching multiple instructions per stage allows the instruction execution rate, CPI, to be less than 1.
  - So, let's use the reciprocal (IPC: instructions per clock cycle)
    - Example
      - CPU: 3-GHz, 4-way multiple-issue processor
      - Peak Execution Rate: 12-billion instructions per second
      - Best-Case CPI of 0.25 → Best-Case IPC of 4
  - If the datapath has a five-stage pipeline, how many instructions are active in the pipeline at any given time?
  - How might this lead to difficulties?

Superpipelined Processors

- Increase the depth of the pipeline leading to more instructions “in flight” at once …
  - The higher the degree of superpipelining …
    - the more forwarding/hazard hardware needed
    - the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time),
    - the bigger the clock skew issues (i.e., because of faster and faster clocks)
  - We know there are limits to this (6–8 FO4 delays).

Superpipelined vs. Superscalar

- Superpipelined (SP) processors
  - Longer instruction latency (in terms of cycles) than superscalar (SS) processors which can degrade performance in the presence of true dependencies
  - Key: Improving throughput at the expense of latency!
- Superscalar processors
  - More susceptible to resource conflicts—but we can fix this with hardware (to a point).

Instruction vs Machine Parallelism

- Instruction-Level Parallelism (ILP) of a Program
  - In Theory
    - A measure of the average # of instructions in a program that, in theory, a processor could execute at the same time
  - In Practice
    - A function of the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- Take Away
  - **ILP is traditionally *"extracting parallelism from a <u>single</u> <u>instruction stream</u> working on a <u>single stream of data</u>"***

(Recall Flynn's Classification ...)

Instruction vs Machine Parallelism

- Machine Parallelism of a Processor
  - A measure of the ability of a processor to take advantage of the ILP of the program
  - In Theory (or "In the Limit")
    - A perfect machine with infinite machine parallelism can achieve the ILP of a program.
  - In Practice
    - A function of the number of instructions that can be fetched and executed at the same time

- ***To achieve high performance, need both ILP and machine parallelism.***

First more on instruction parallelism, then machine parallelism.

Points of Contemplation

- Why is ILP a good idea?
- If you were designing a computer system …
  - Why choose ILP instead of multiple processors/cores?

- What kind of code has lots of ILP?
- What kind of code has little ILP?
- *What is the reality?*

Matrix Multiplication

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

"Assembly Code" for y0

```
y0 = m00*x0 + m01*x1 + m02*x2 + m03*x3
```

```
t0 = m00 * x0
t1 = m01 * x1
t2 = m02 * x2
t3 = m03 * x3
t4 = t0 + t1
t5 = t2 + t3
y0 = t4 + t5
```

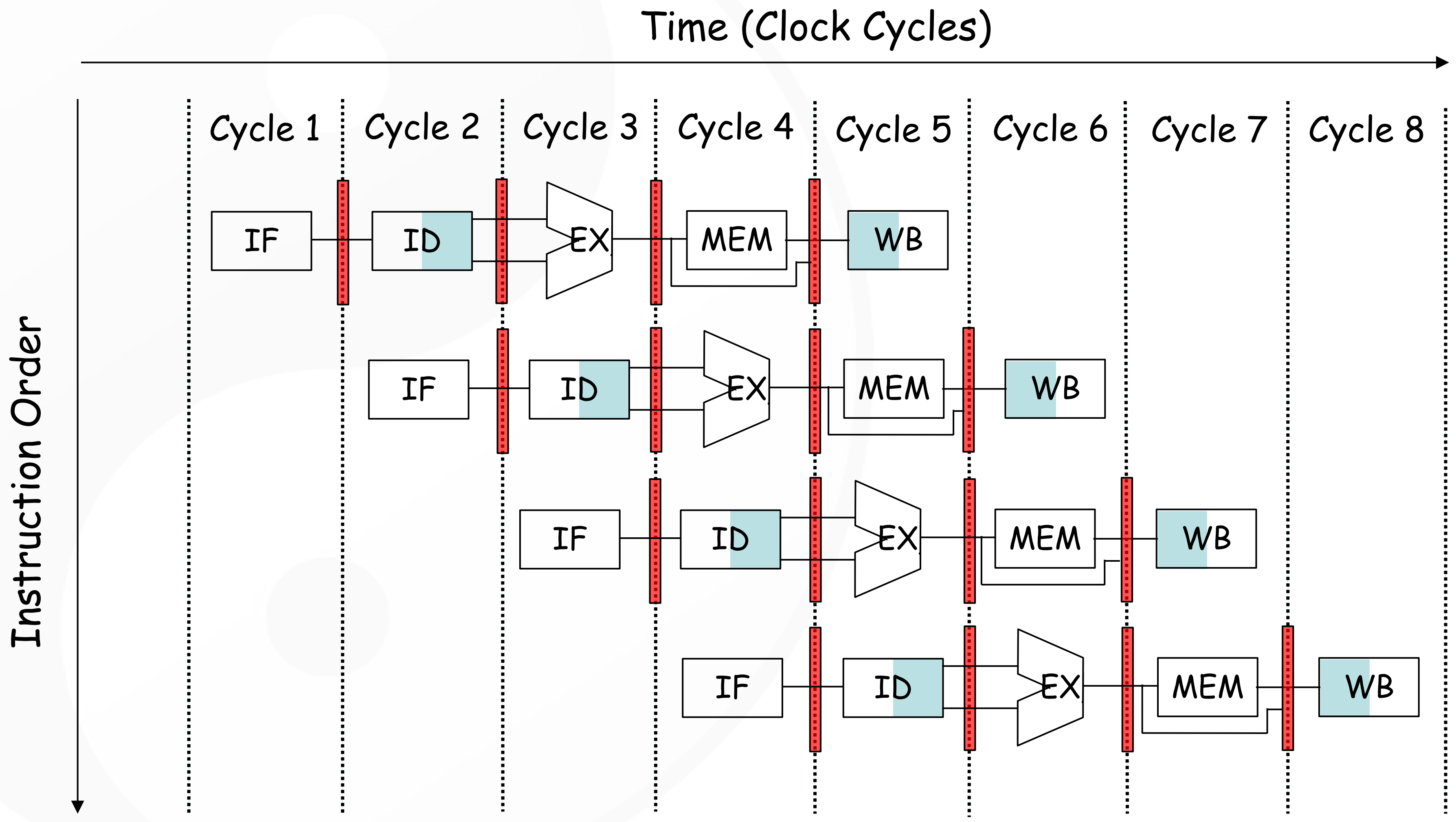
In this case …
what will the machine parallelism be
relative to the instruction-level parallelism?

- What is the ILP for one product (y0)?
- What is the ILP for the entire matrix?
- What is the ILP for doing 100 matrices in parallel?

Basics of a RISC Instruction Set

- Key Properties
  - All operations on data apply to REGISTERS (typically the *entire* register, i.e., 32 or 64 bits per register)
  - Only operations that affect memory?
    - LOAD and STORE
  - Instruction formats?
    - FEW (in contrast to CISC where there are many)
      - ALU instructions
      - LOAD and STORE instructions
      - BRANCH and JUMP instructions
    - FIXED-SIZE (in general)
      - 32 bit or 64 bit, depending on architecture

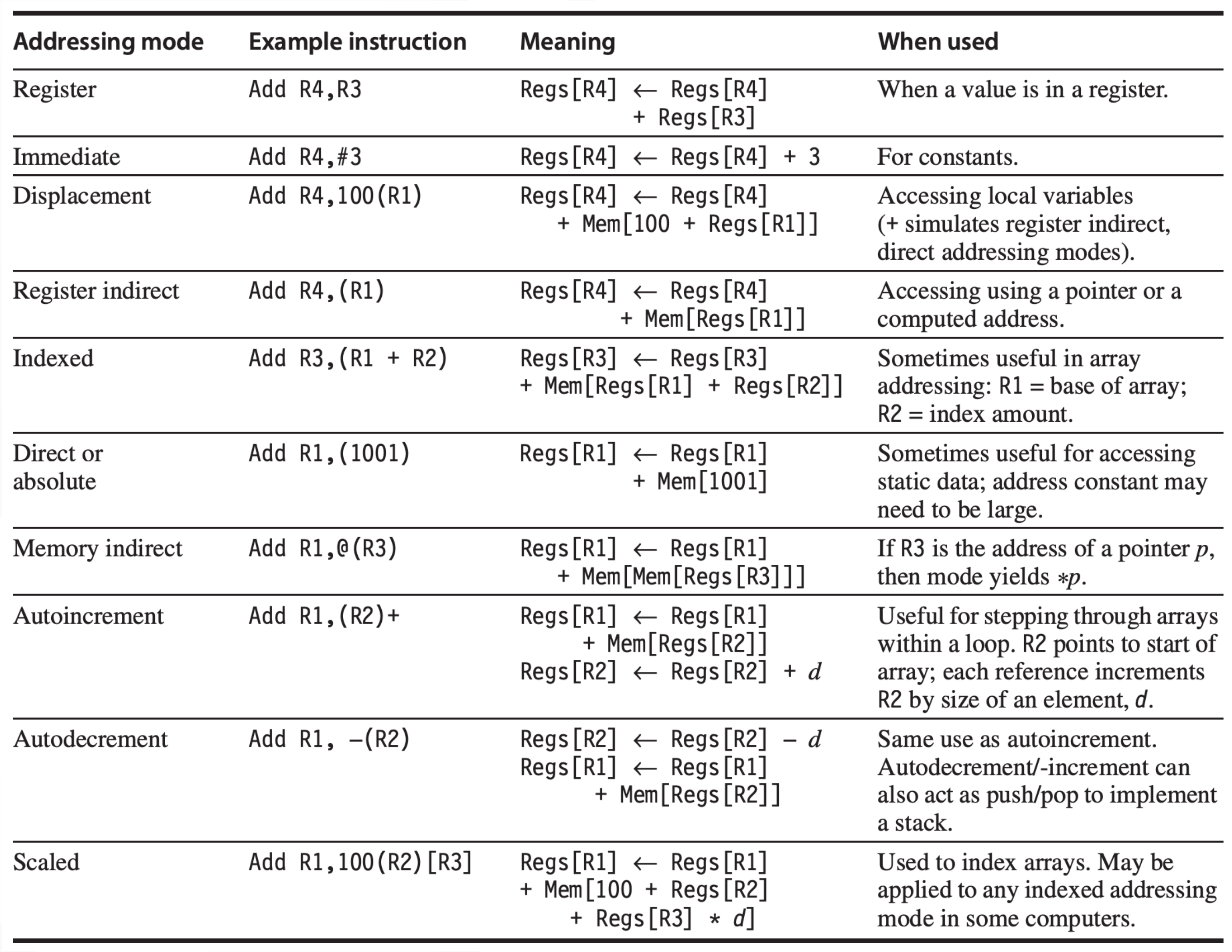
Pipelined Processor



Overview of MIPS64 ISA
(see Appendix)

- Registers
  - 32 64-bit general-purpose registers (GPRs)
  - 32 floating-pt registers (FPRs), holding 32 SP or 32 DP values
    - (When holding a SP number, the other half of the FPR is unused.)
- Data Types → Integer and Floating Point
  - 8-bit bytes; 16-bit half words, 32-bit words, 64-bit double-words
  - 32-bit single-precision (SP); 64-bit double-precision (DP)
- Addressing Modes → ONLY THREE natively supported!
  - Register
  - Immediate (16-bit field)
  - Displacement (16-bit field)
    - Register-indirect addressing: 0 in displacement field
    - Absolute addressing: R0 as the base register

Addressing Modes of Past ISAs (see Fig. A.6)



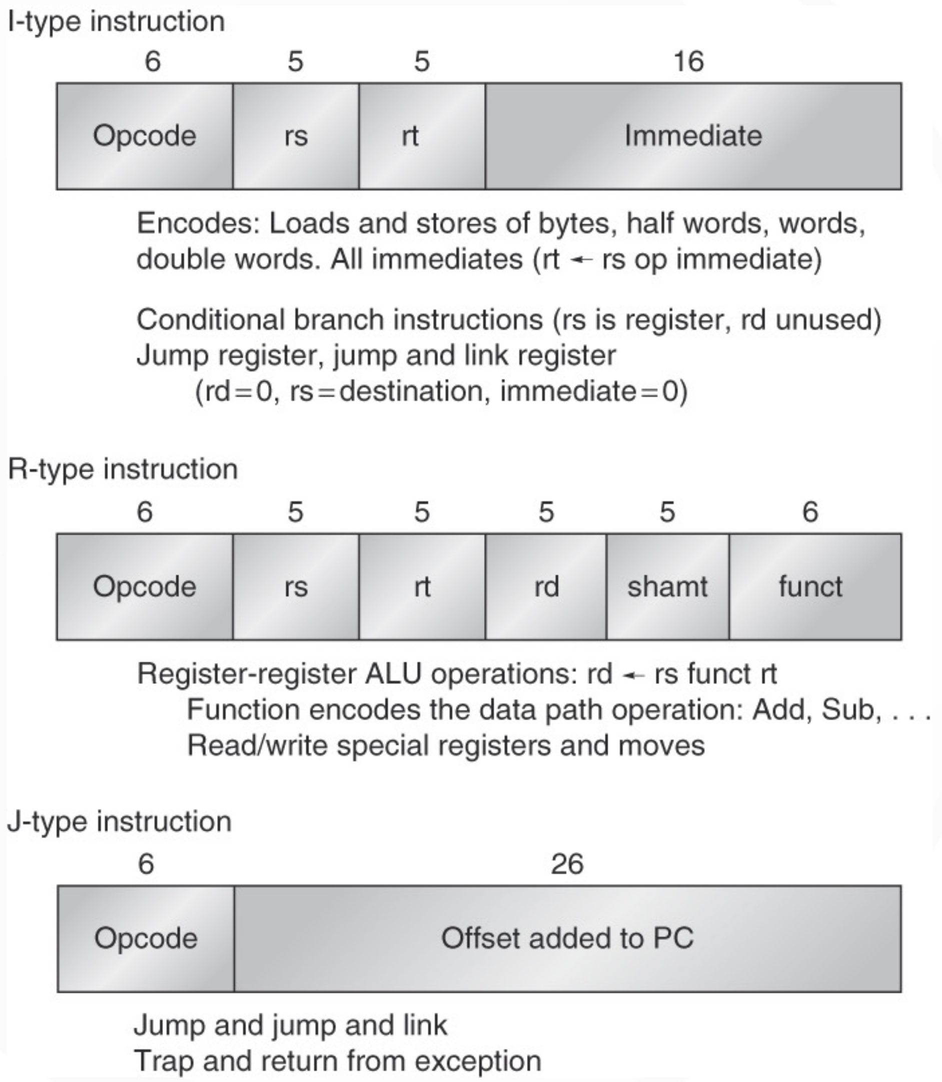
Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] +` *d*	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1, –(R2)`	`Regs[R2] ← Regs[R2] –` *d* `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] *` *d*`]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Encoding an Instruction Set

Balancing Act of Many Competing Forces

- Desire to have as many registers and addressing modes as possible.
- Impact of the size of the register and addressing mode fields (on average instruction size, and hence, on average program size).
- Desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.
  - Ideally, instructions should be in multiples of bytes (or words) rather than arbitrary bit length.

Instruction Layout for MIPS Pipelined Processor



Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

Jump and jump and link
Trap and return from exception

This is for the MIPS32 ISA.

Resources for MIPS64 ISA

- https://www.mips.com/products/architectures/mips64/
- https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00083-2B-MIPS64INT-AFP-06.01.pdf

Basic Five-Stage Pipelined MIPS Architecture

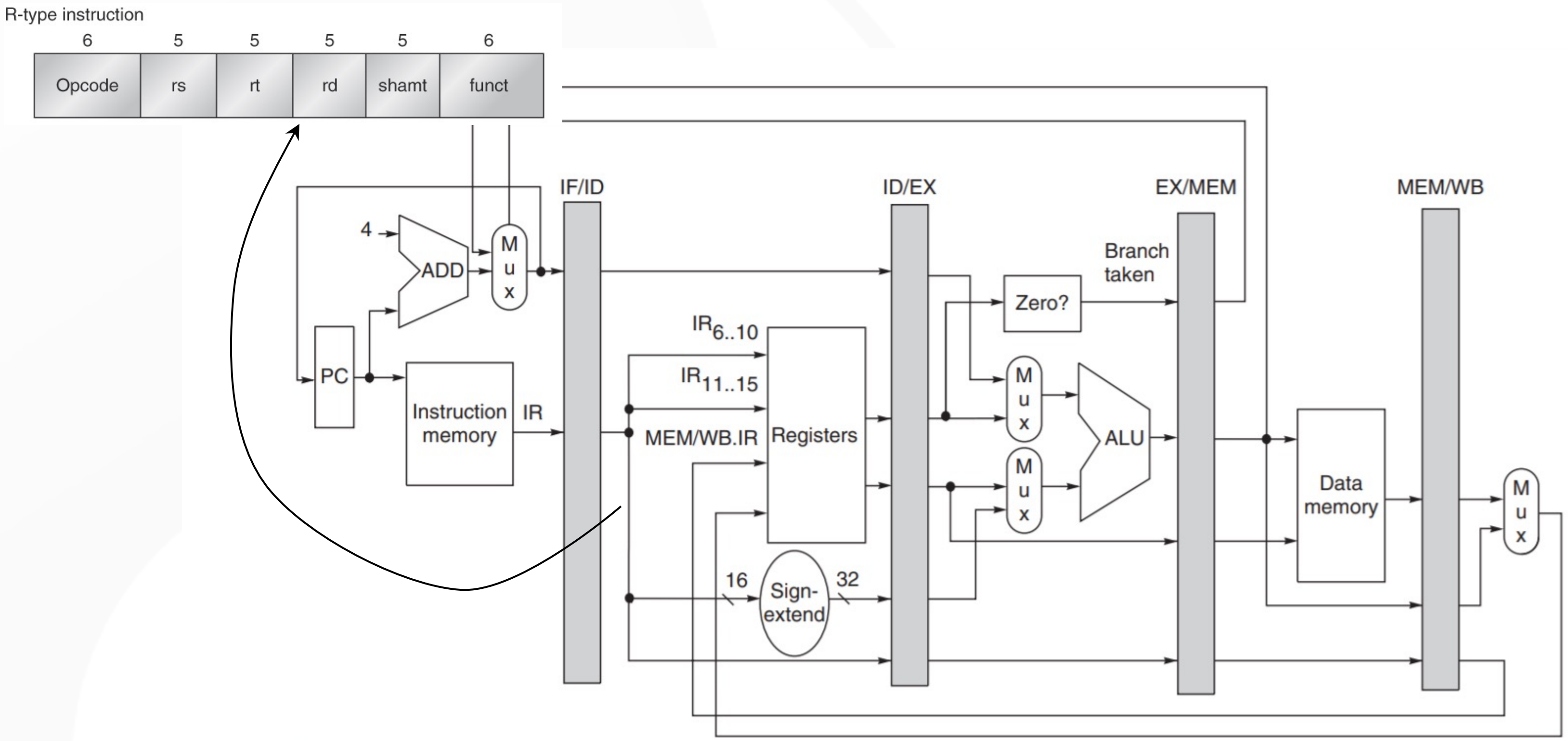
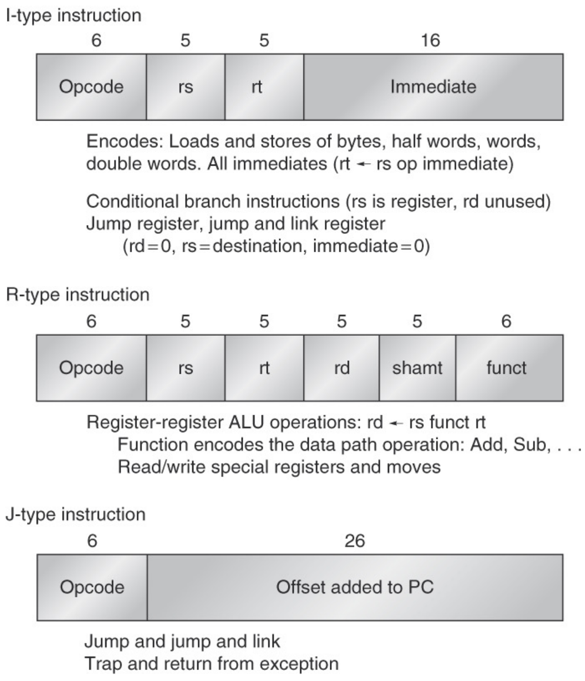


Figure C.22 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence, there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, since two instructions would try to write different values into the PC. Most of the data paths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline.

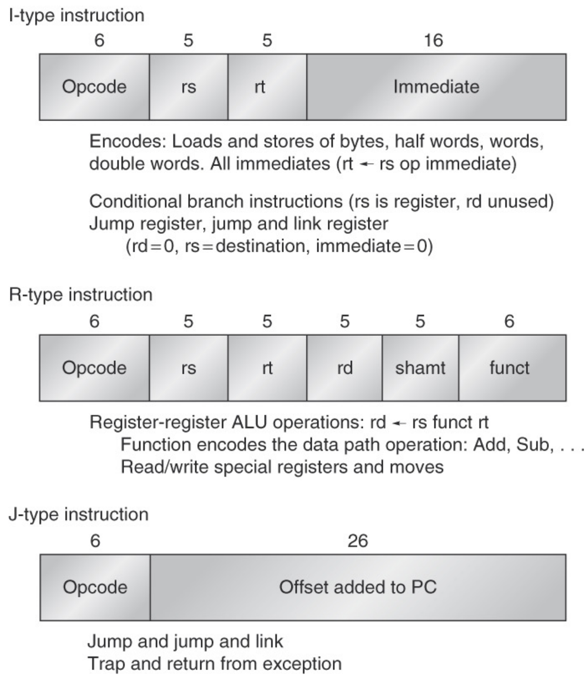
Five-Stage Pipelined Processor

1. Instruction Fetch (IF)
   - Send PC to memory & fetch current instruction from memory
   - Update PC to next sequential PC (e.g., 4 for 32-bit architecture and 8 for 64-bit architecture)
2. Instruction Decode (ID) / Register Fetch
   - Decode instruction *and* read source registers in parallel
     - Why is this possible? "Fixed-field decoding"
   - Do *EQUALITY* test on registers during read for possible branch
     - Sign-extend the offset field of instruction, if needed
     - Compute possible branch target address (by adding offset to PC)


I-type instruction
6
5
5
16
Opcode
rs
rt
Immediate
Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd=0, rs=destination, immediate=0)
R-type instruction
6
5
5
5
5
6
Opcode
rs
rt
rd
shamt
funct
Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves
J-type instruction
6
26
Opcode
Offset added to PC
Jump and jump and link
Trap and return from exception

Five-Stage Pipelined Processor

3. Execute (EXE) / Effective Address
   - EXE on operands from previous cycle
     - Memory reference calculation
     - Register-Register ALU instruction
     - Register-Immediate ALU instruction
4. Memory Access (MEM)
   - LOAD: Memory read using effective address calculated
   - STORE: Memory write data from register read to effective address
5. Write Back (WB)
   - Register-Register ALU instruction or LOAD instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

Jump and jump and link
Trap and return from exception

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to *minimize* CPI (or *maximize* IPC)
  - Pipeline CPI =

    Ideal pipeline CPI +

    Structural stalls +

    Data hazard stalls +

    Control stalls

Nomenclature

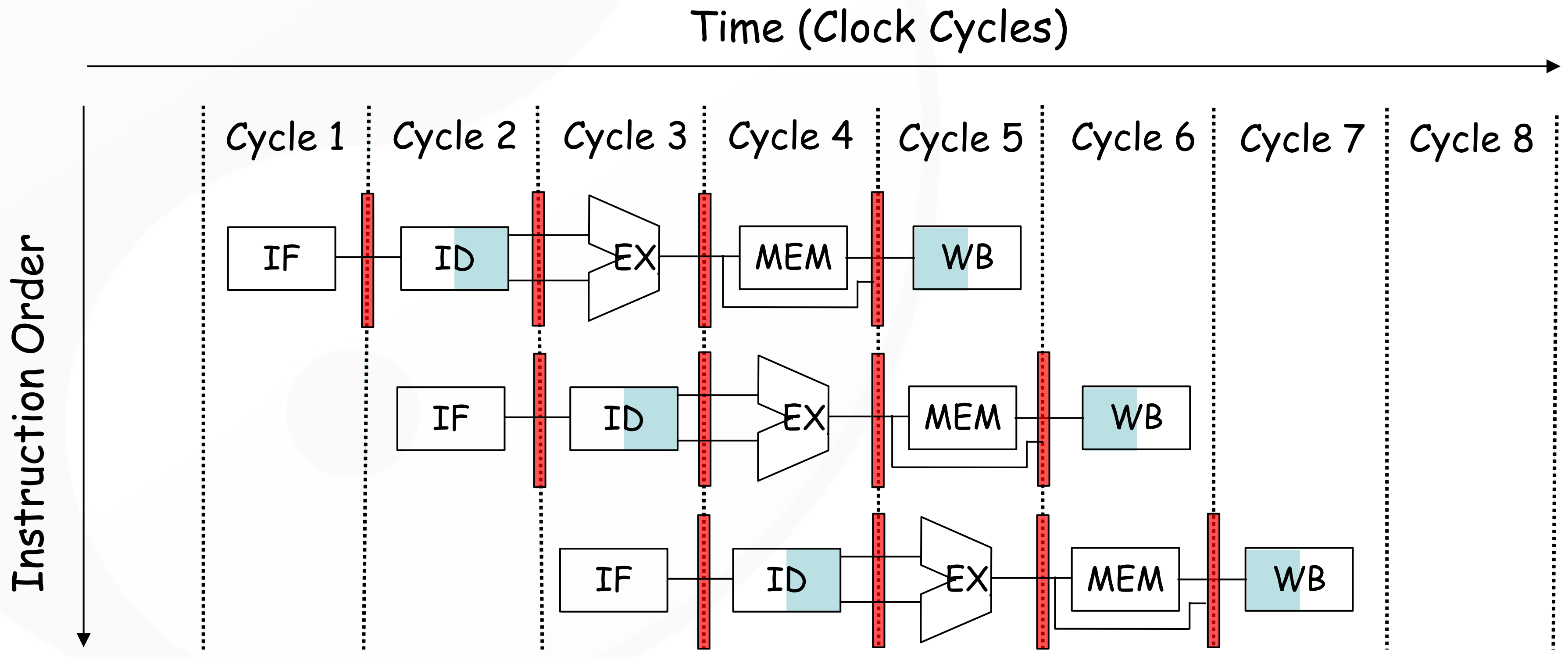
- Pipeline stalls are *synonmous* with "bubbles" in the pipeline.
- Pipeline stalls are *caused* by hazards.

Review: Pipeline Hazards

- Hazards
  - Situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle.
  - Consequence
    - Insertion of “stalls” or “bubbles” into the pipeline
  - Types
    - Structural Hazards
    - Data Hazards
    - Control Hazards

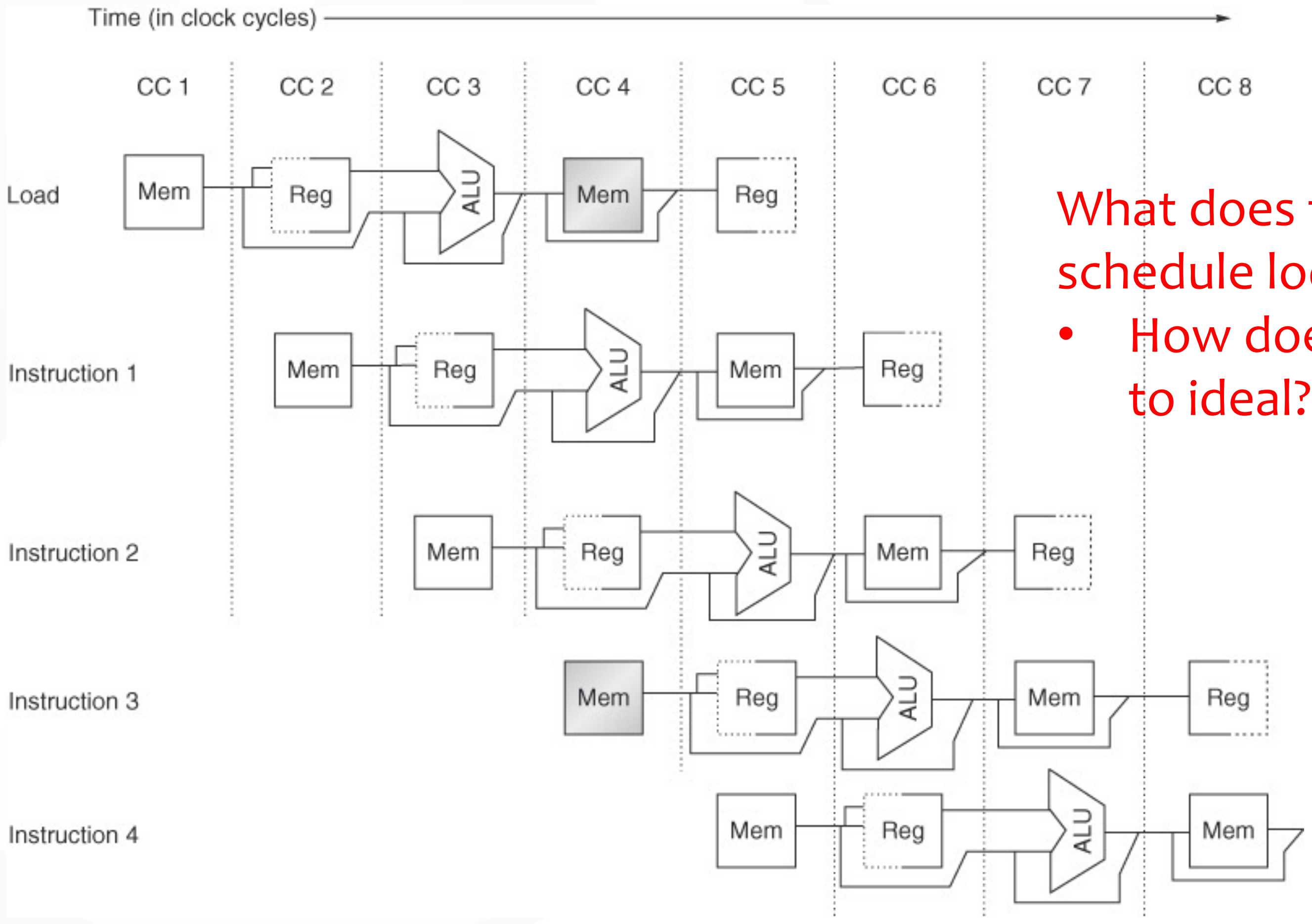
Review: Pipeline Hazards

- Structural Hazards
  - What are they? How do we eliminate them?
    - Design pipeline to eliminate structural hazards



Structural Hazard

- Assume a CPU with only one memory port

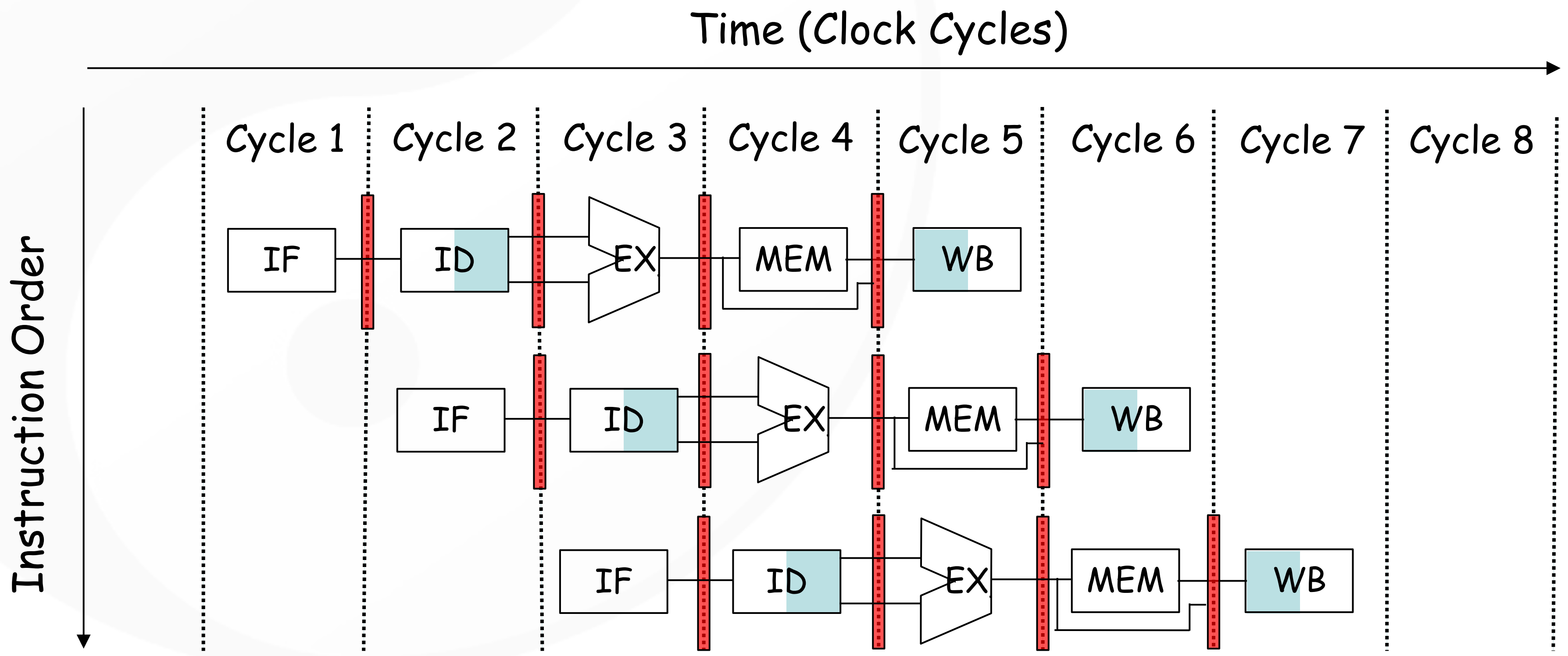


What does the pipeline schedule look like?

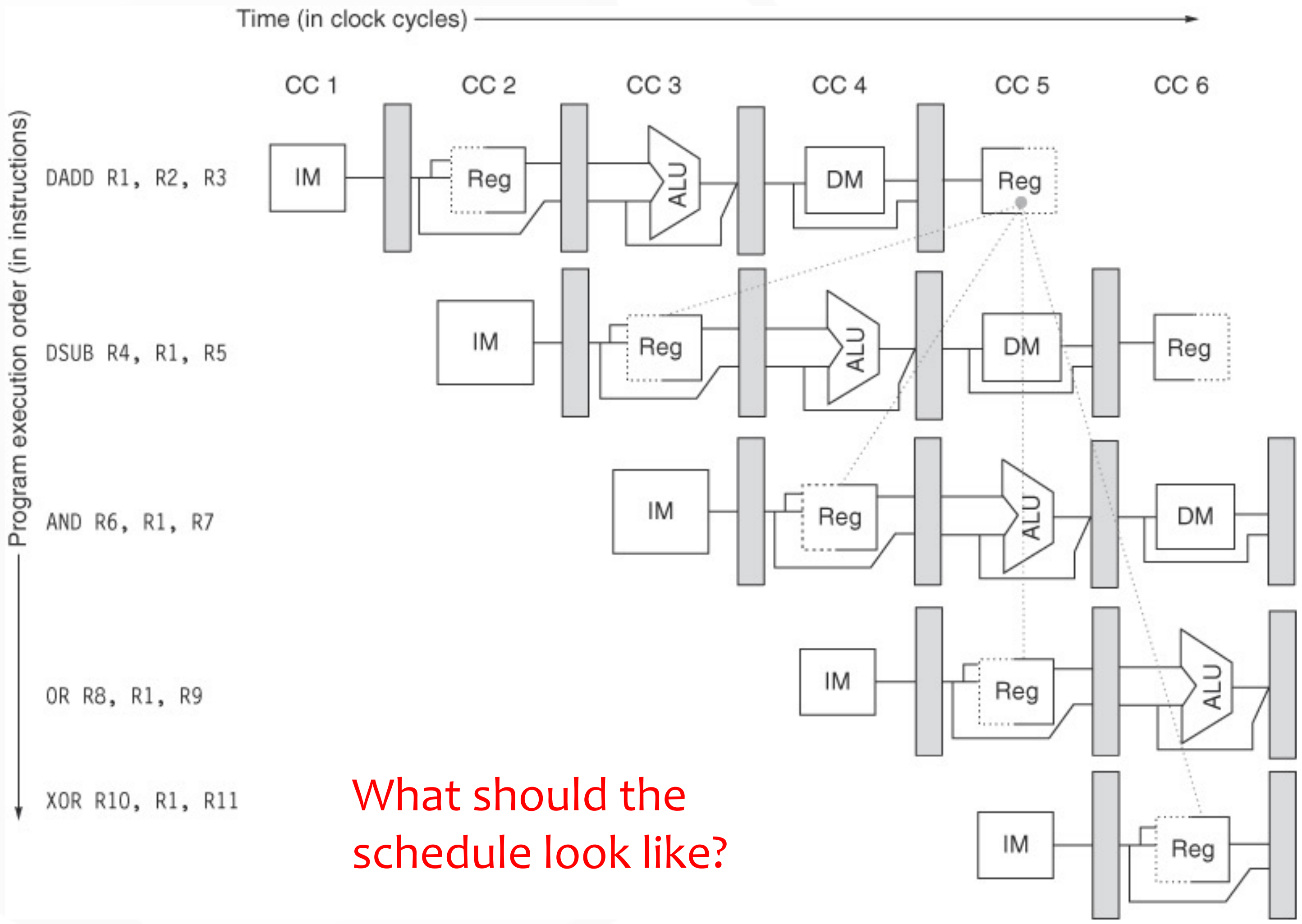
- How does it compare to ideal?

Review: Pipeline Hazards

- Data Hazards – Read After Write (RAW)
  - What are they? How do we eliminate them?
    - Use data forwarding inside the pipeline
    - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream

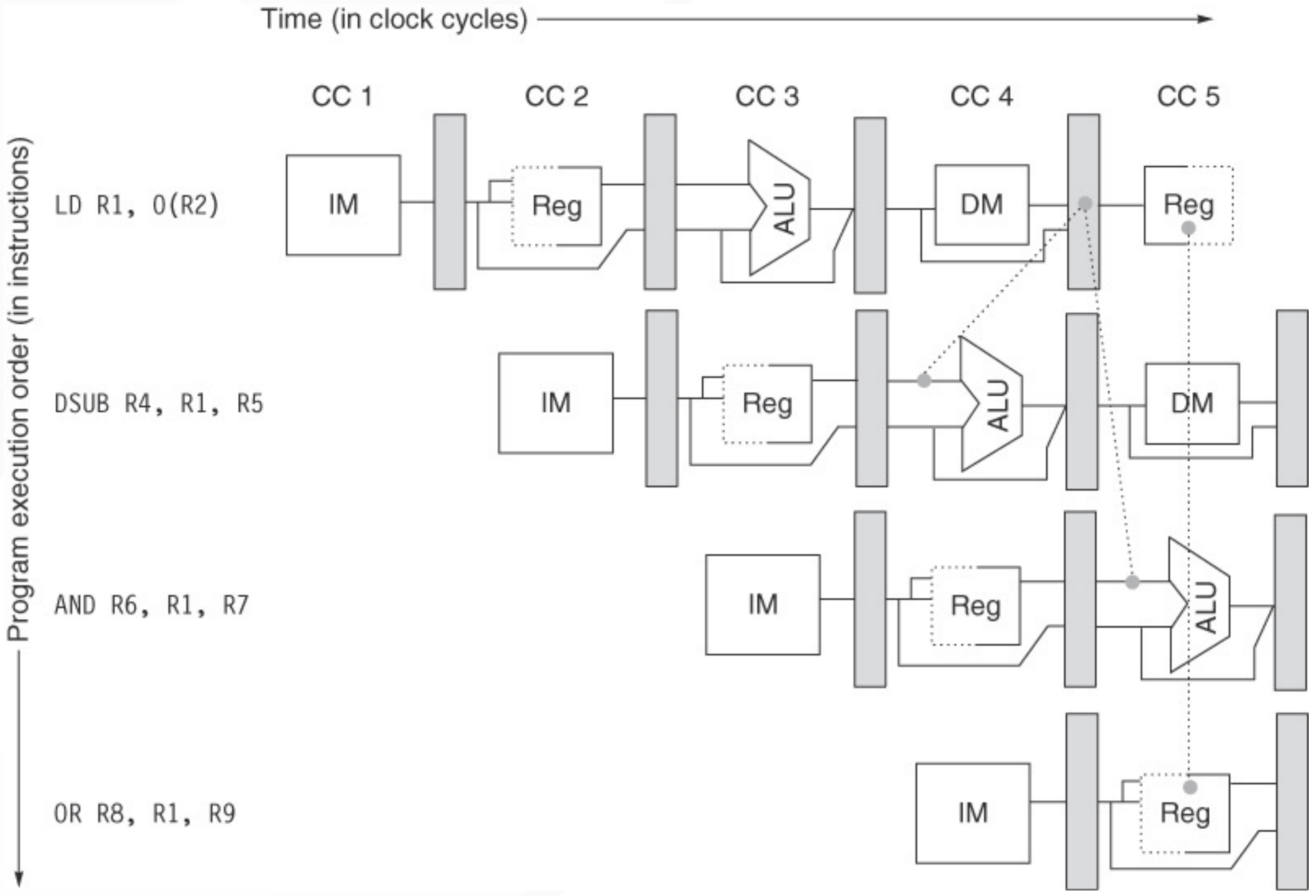


Data Hazard



What should the schedule look like?

Unavoidable Data Hazards



Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect

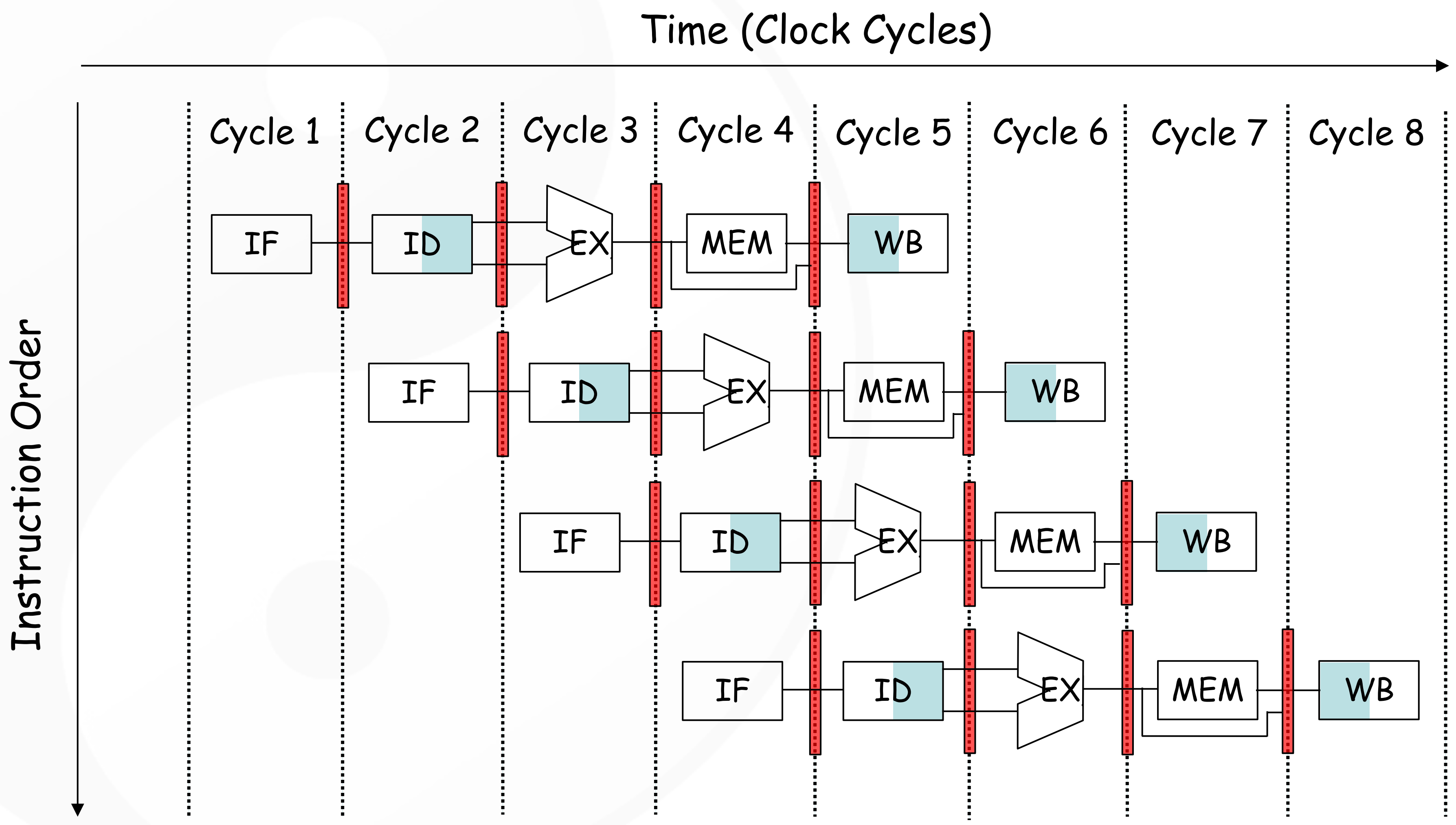
Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence but is a problem when reordering instructions
  - ***Antidependence:*** instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - ***Output dependence:*** instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use renaming techniques

Data Hazards

- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Pipelined Processor

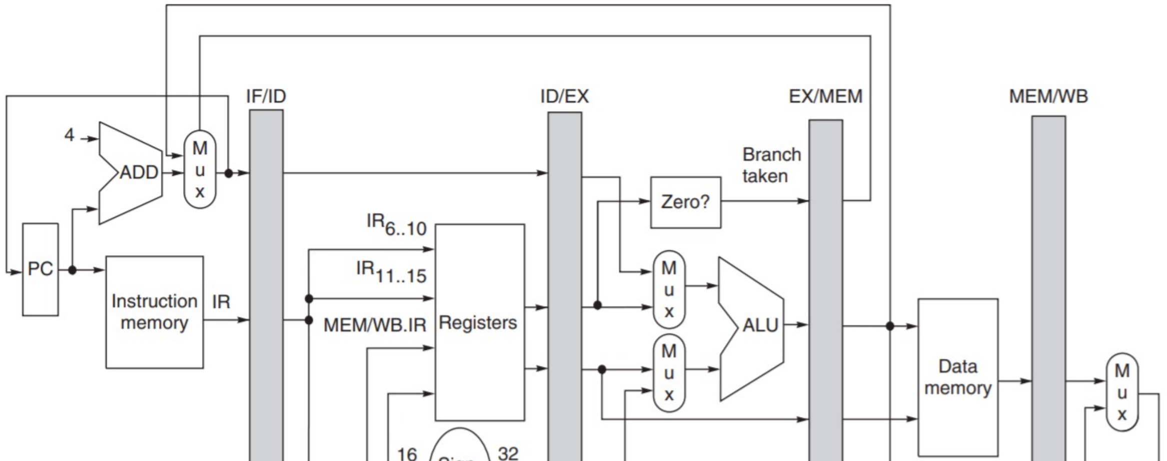


Review: Pipeline Hazards

- Control Hazards – `beq, bne, j, jr, jal`
  - What are they?
    - Determines the ordering of an instruction *i* with respect to a "change-in-flow" instruction, i.e., branch or procedure call.
  - How do we eliminate them?
    - Stall, but it hurts performance
    - Move decision point to as early in the pipeline as possible
      - Reduces # of stalls at the cost of additional HW
    - Delay decision (requires compiler support)
      - Not feasible for deeper pipes requiring more than one delay slot to be filled
    - Predict (with more HW)
      - Reduces the impact of control hazard stalls
        - » if branch prediction is correct and
        - » if branched-to instruction is cached

Recall:
Five-Stage Pipelined Processor

1. Instruction Fetch (IF)
   - Send PC to memory & fetch current instruction from memory
   - Update PC to next sequential PC (e.g., 4 for 32-bit architecture)
2. Instruction Decode (ID) / Register Fetch
   - Decode instruction *and* read source registers in parallel
     - Why is this possible? "Fixed-field decoding"
   - Do *EQUALITY* test on registers during read for possible branch
     - Sign-extend the offset field of instruction, if needed
     - Compute possible branch target address (by adding offset to PC)



Constraints of Control Dependences

1. An instruction that is control-dependent on a branch

   ... cannot be moved *BEFORE* the branch so that its execution *is no longer controlled* by the branch.

2. An instruction that it *not* control-dependent on a branch

   ... cannot be moved *AFTER* the branch so that its execution *is controlled* by the branch.

Caveat?

- May execute instructions that should not be executed, thus violating control dependences, but only *IF* we can do so without affecting the correctness of the program.

Examples

Example 1:

```
   DADDU R1,R2,R3
   BEQZ  R4,L
   DSUBU R1,R1,R6
L: …
   OR    R7,R1,R8
```

- OR instruction *dependent* on DADDU and DSUBU
  - R1 used by OR *depends* on branch

Due to constraint of control dependence, DSUBU cannot be moved above branch.

(Caveat: Speculation?)

Example 2:

```
   DADDU R1,R2,R3
   BEQZ  R12,skip
   DSUBU R4,R5,R6
   DADDU R5,R4,R9
skip:
   OR    R7,R8,R9
```

- Assume R4 is *not* used after skip
  - Possible to move DSUBU before the branch

Property of whether a value is used by an upcoming instruction: *liveness*.

Summary: Hazards

- Hazards are bad because they reduce the amount of achievable machine parallelism and keep us from achieving all the ILP in the instruction stream.

Recall: Instruction vs Machine Parallelism

- Instruction-Level Parallelism (ILP) of a Program
  - In Theory
    - A measure of the average # of instructions in a program that, in theory, a processor could execute at the same time
  - In Practice
    - A function of the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- Take Away
  - **ILP is traditionally** ***"extracting parallelism from a single instruction stream working on a single stream of data"***

(Recall Flynn's Classification …)

Recall: Instruction vs Machine Parallelism

- Machine Parallelism of a Processor
  - A measure of the ability of a processor to take advantage of the ILP of the program
  - In Theory (or "In the Limit")
    - A perfect machine with infinite machine parallelism can achieve the ILP of a program.
  - In Practice
    - A function of the number of instructions that can be fetched and executed at the same time

- ***To achieve high performance, need both ILP and machine parallelism.***

First more on instruction parallelism, then machine parallelism.

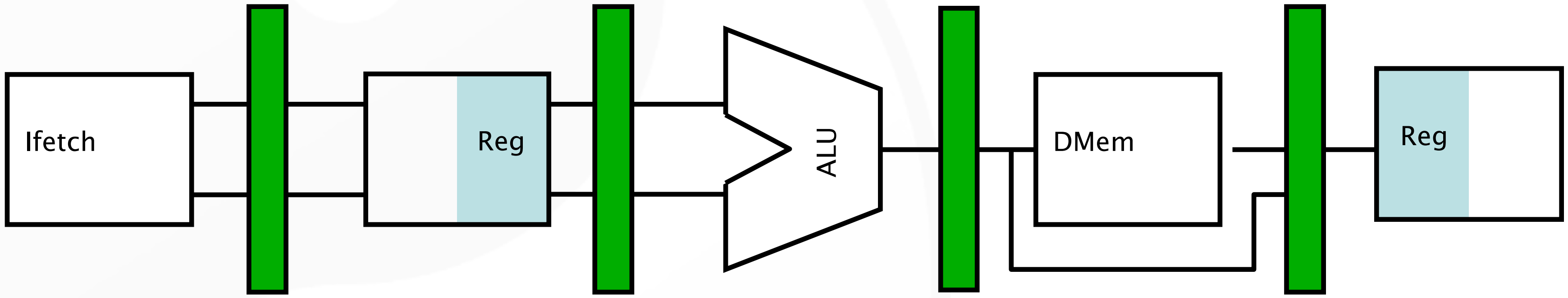
Machine Parallelism

- Two Approaches for Machine Parallelism
  Responsibility of resolving hazards is …
  - Hardware-based
    - Dynamic-issue superscalar
  - Software-based
    - Static "VLIW: Very Long Instruction Word"

Multiple-Issue Processor Styles

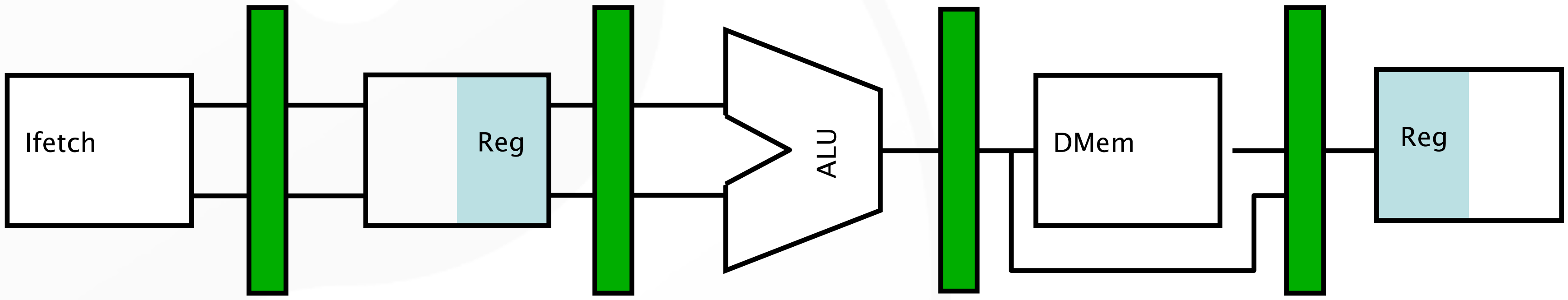
- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
    - Examples: IBM Power 2, Pentium Pro/2/3/4, Core, MIPS R10K, HP PA 8500
- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
    - Examples: Intel Itanium and Itanium 2 for the IA-64 ISA—EPIC (Explicit Parallel Instruction Computer)

Supporting Machine Parallelism?



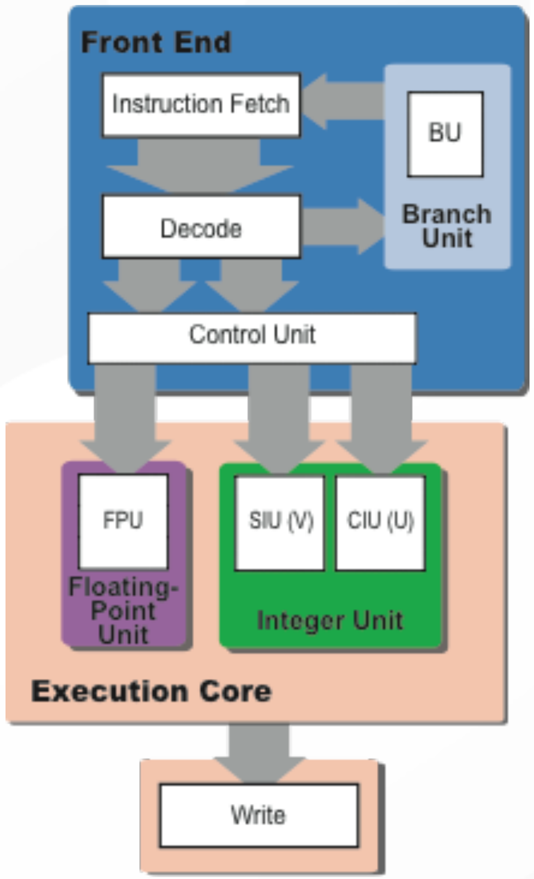
- Now, how do we support multiple integer instructions?

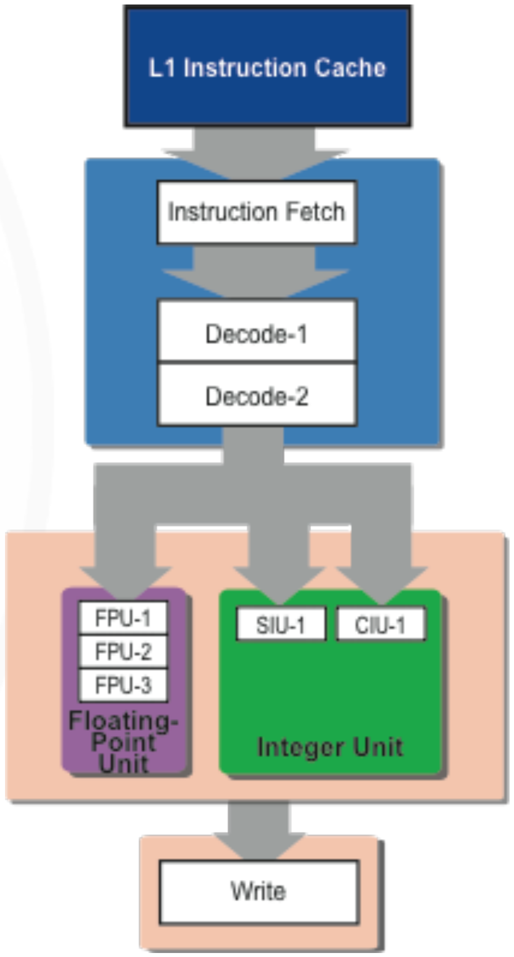
Supporting Machine Parallelism?



- First, let’s support parallel integer & FP instructions (MIPS)

Pentium Microarchitecture





Multiple-Issue Datapath Responsibilities

- Datapath handled with a combination of HW and SW.
- Fundamental limitations of hazards
  - *Storage (data) dependencies—aka data hazards*
    - Most instruction streams do not have huge ILP so …

      … this limits performance in a superscalar processor

Multiple-Issue Datapath Responsibilities

- Datapath handled with a combination of HW and SW.
- Fundamental limitations of hazards
  - *Procedural dependencies—aka control hazards*
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue

Multiple-Issue Datapath Responsibilities

- Datapath handled with a combination of HW and SW.
- Fundamental limitations of hazards
  - *Resource conflicts – aka structural hazards*
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by *duplicating the resource* or by *pipelining the resource*