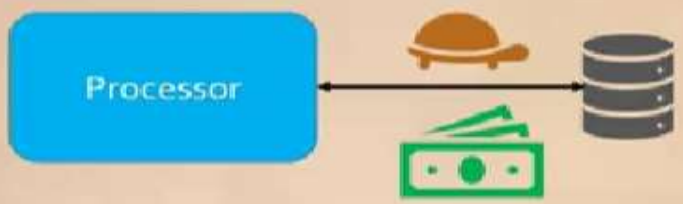
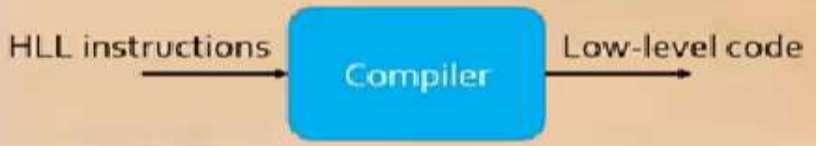
Chapter 8
RISC

Brief History





- Memory was slow and expensive
- Designing a compiler was difficult due to advanced High Level Languages
- Compact instructions simplified the task of designing a compiler
- Occupied less code space

Selection Criteria and Examples

CISC	RISC
Less Code Space	Low Power Consumption
Examples: Motorola 68K, x86, System/360, VAX	Examples: Sun's SPARC, Power PC, MIPS, AVR, ARM
Found in PCs & Servers	Preferred in mobile devices/IoT

Reduced Instruction Set Computing (RISC)

- An important aspect of computer architecture is the design of instruction set for the processor.
- The instruction set chosen for a particular computer determines the way, the machine language programs are constructed.
- Many computers have instruction set that include more than 100 and sometimes even more than 300 instructions.
- In 1980s, a number of computer designers recommended that computers use fewer instructions with simple construct so that they can be executed much faster within the CPU.
- This type of computer is classified as reduced instruction set computing(RISC).

=> The greater the number of instructions in an instruction set, the larger the propagation delay.

=> For eg; if the CPU had 32 instruction, a 5 x 32 decoder would have been needed. This decoder would require more time to generate output than the smaller 4 x 16 decoder, which would reduce the maximum clock rate of CPU.

Characteristics of RISC:

- Relatively few instructions.
- Fixed-length instructions:
  => instructions of same size; format can be different.
- Limited loading and storing instructions Access memory.
- Fewer Addressing modes.
- Instruction pipelining.
- Large number of registers so as to so as to store many operands internally. When the operands are needed, the CPU fetches them from registers, rather than from memory, thereby reducing the access time.
- Hardwired control unit rather than microprogram control so as to have a lower propagation delay.

RISC Instruction Set

- The instruction set of RISC processor is reduced whereas CISC(Complex Instruction Set Computing) processor might have over 300 instructions in its instruction set, RISC CPU typically have less than 100.
- These instructions perform a wide variety of function, each of which is being executed in a single clock cycle.
- When developing a RISC instruction set, it is important not to reduce the set too much.
- Consider, for example, the instruction set for a processor includes AND, OR, NOT and XOR instruction. Using De-Morgan's law, an OR can be implemented using only AND and NOT.
  A OR B = NOT((NOT A) AND (NOT B))

Contd…

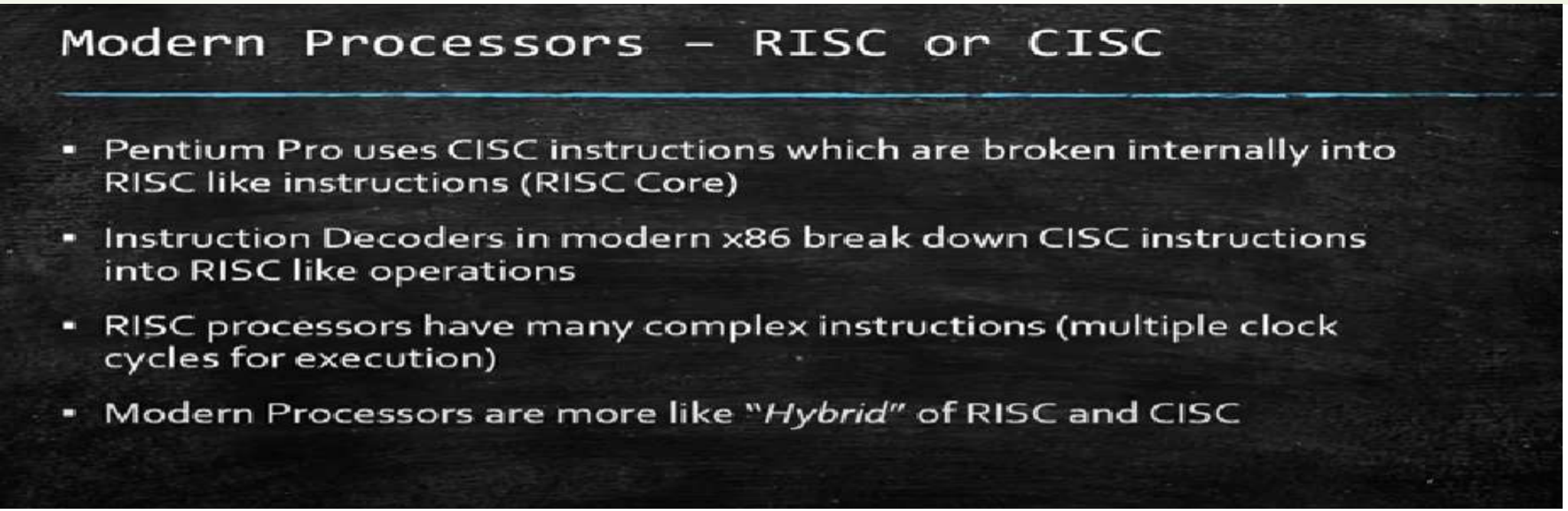
- Similarly, an XOR can be realized using the same operation
  A XOR B = NOT ((NOT (A AND (NOT B))) AND (NOT ((NOT A) AND B)))
- Therefore, we can exclude OR and XOR instruction from the instruction set and still allow the CPU to perform the same function.

RISC vs. CISC

- Let us take an example of multiplying two numbers; A = A * B

CISC approach	RISC approach
> The entire task of multiplying two numbers can be completed with one instruction: MULT A,B; which:	> The "MULT" command is divided into several command to perform multiplication.
1. Loads two values into separate registers.	1. LOAD R1, A LOAD R2 , B
2. Multiplies the operands.	2. PROD A , B
3. Stores the product in the appropriate register.	3. STORE R3 , A
	> All executed in one clock cycle.

CISC	RISC
1. Emphasis on hardware	1. Emphasis on software
2. Includes multi-clock	2. Single clock
3. Complex instructions	3. Reduced instruction only
4. "LOAD" and "STORE" incorporated in instructions	4. "LOAD" and "STORE" are independent instructions
5. Small code size	5. Large code size



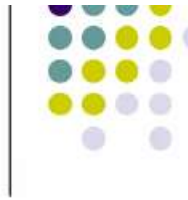
CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions

Examples of processors with the RISC architecture include MIPS, PowerPC, Atmel's AVR, the Microchip PIC processors, Arm processors, RISC-V, and all modern microprocessors have at least some elements of RISC. Some examples of CISC microprocessor instruction set architectures (ISAs) include the Motorola 68000 (68K), the DEC VAX, PDP-11, several generations of the Intel x86, and 8051.

The RISC ISA emphasizes software over hardware. The RISC instruction set requires one to write more efficient software (e.g., compilers or code) with fewer instructions. CISC ISAs use more transistors in the hardware to implement more instructions and more complex

There is really no “better” architecture, each has its own advantages and disadvantages that make it useful in different applications. CISC is most often used in automation devices whereas RISC is used in video and image processing applications.
When microprocessors and microcontroller were first being introduced, they were mostly CISC. This was largely because of the lack of software support present for RISC development.
Later a few companies started delving into the RISC architecture, most notable, Apple, but most companies were unwilling to risk it (pun intended) with an emerging technology.
Fast forward a few decades and the CISC architecture was becoming extremely unwieldy and difficult to improve and develop. Intel however had a lot of resources at their hand and were able to plow through most of the major roadblocks. They were doing this mainly to make all their hardware and software back compatible with their initial 8086 processors.
Nowadays however, the boundary between RISC and CISC architectures are very blurred and in most cases it is not important. ARM devices, PICs and almost all smartphone manufacturers use RISC devices as they are faster and less resource and power hungry.
The only completely CISC device still in existence is probably the Intel x86 series.
Intel/HP EPIC/IA-64 Architecture(EPIC Architecture)

pipeline



- It is technique of decomposing a sequential process into suboperation, with each suboperation completed in dedicated segment that operates concurrently with all other segments.
- Pipeline is commonly known as an assembly line operation.

Example

$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

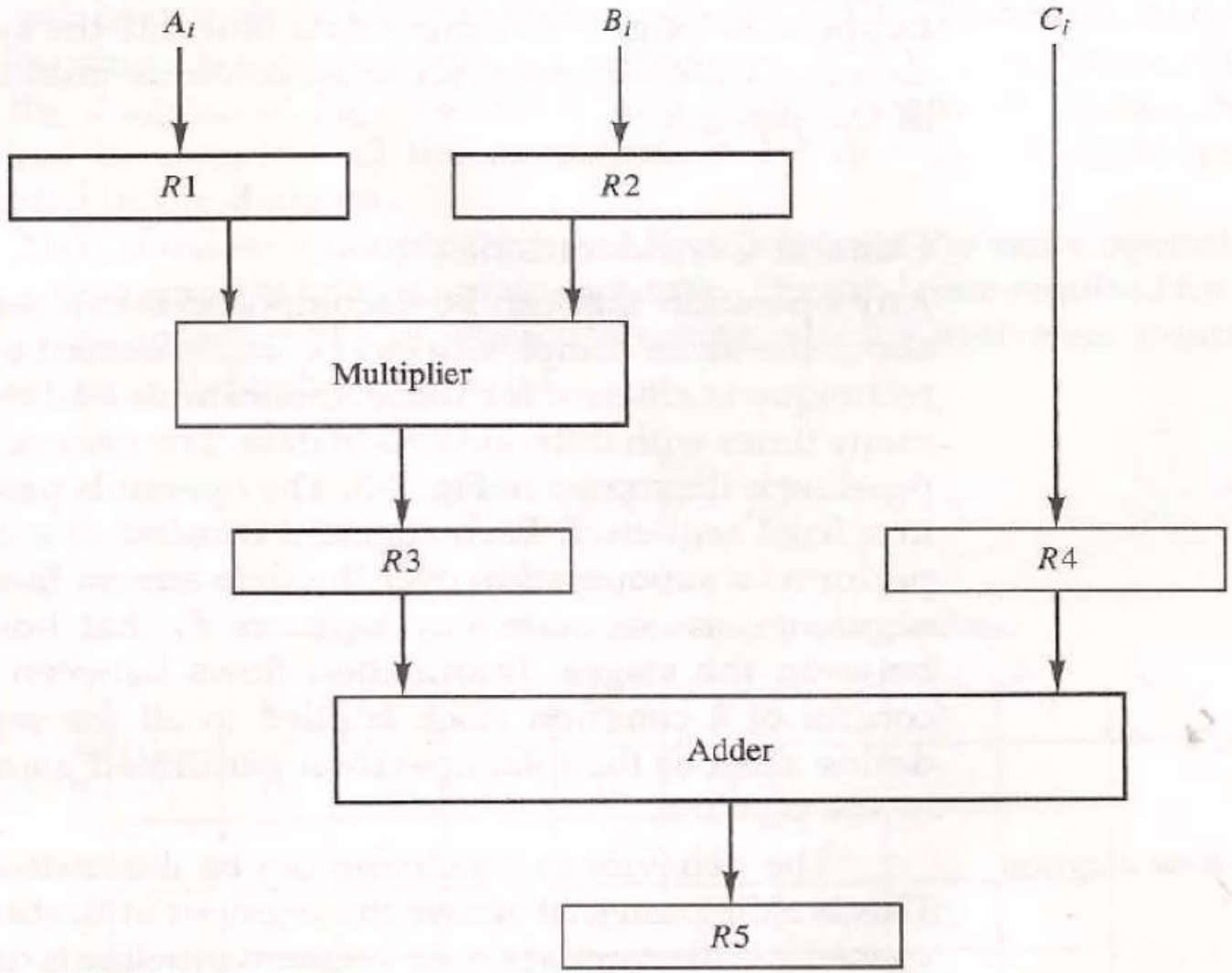
Each sub operation is to be performed in a segment within a pipeline. Each segment has one or two registers and a combinational circuit.

- The sub operations in each segment of the

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

pipeline are as follows:

Figure Example of pipeline processing.



- Operations In Each Pipeline Stage

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	---	---	-------
2	A2	B2	A1 * B1	C1	-------
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

advantages

- 1- Pipelining is widely used in modern processors .
- 2- Quicker time of execution large number of instruction.
- 3- More efficient use of processor.
- 4- Arrange the hardware so that more than one operation can be performed at the same time.
- 5- This technique is efficient for applications that need to repeat the same task in many time with different set of data.

Disadvantages

- 1- Pipelined organization requires complex compilation techniques.
- 2- pipelining involves adding hardware, then cost of the system increases.

Idea of pipelining in computer

- The processor execute the program by fetching and executing instructions. One after the other.

- Let F_i and E_i refer to the fetch and execute steps for instruction I_i

Use the Idea of Pipelining in a Computer

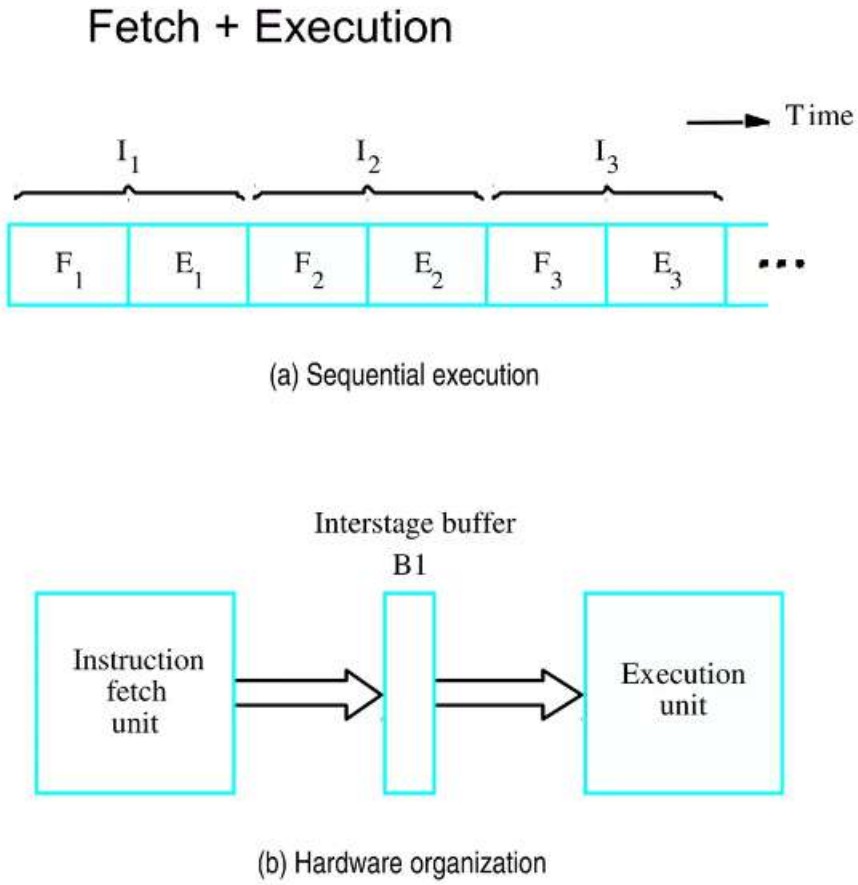


Figure 8.1. Basic idea of instruction pipelining.

Use the Idea of Pipelining in a Computer

- Computer that has two separate hardware units, one for fetching and another for executing them.

- This buffer needed to enable the execution unit while fetch unit fetching the next instruction.
- The computer is controlled by a clock.

Role of Cache Memory

- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.[ten times greater than the time needed to perform pipeline stage]
- Fortunately, we have cache.

Types of pipeline

- 1) Software Pipelining
    1) Can Handle Complex Instructions.
    2) Allows programs to be reused.
- 2)Hardware Pipelining
    1) Help designer manage complexity – a complex task can be divided into smaller, more manageable pieces.
    2) Hardware pipelining offers higher performance.

Types of pipeline

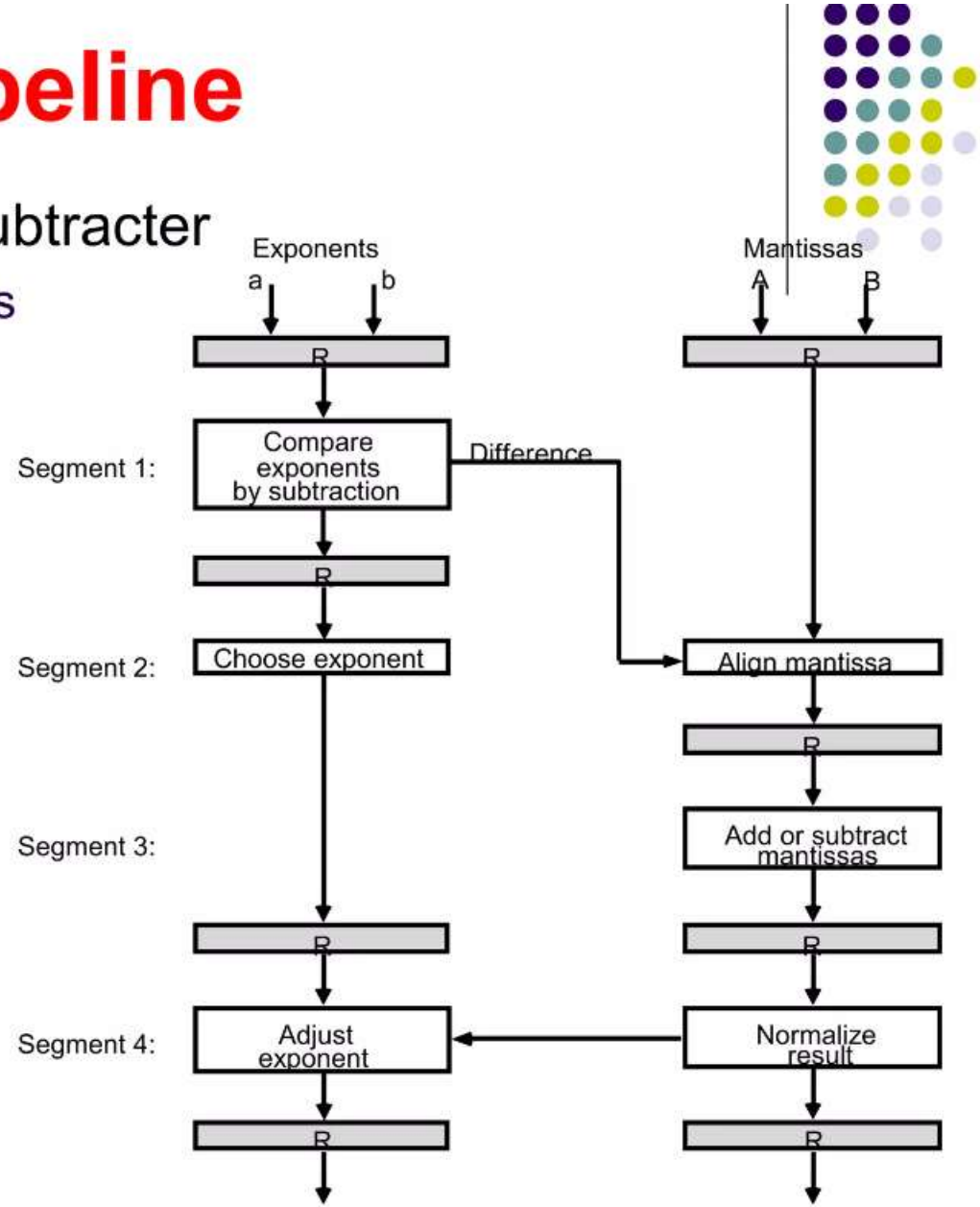
- Arithmetic Pipeline : Pipeline arithmetic units are usually found in very high speed computers.
- Floating–point operations, multiplication of fixed-point numbers, and similar computations in scientific problem.
- Instruction Pipeline: Pipeline processing can occur also in the instruction stream. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.

Arithmetic Pipeline

- Floating-point adder/subtracter
- [1] Compare the exponents
- [2] Align the mantissa
- [3] Add/sub the mantissa
- [4] Normalize the result

$X = A \times 10^a = 0.9504 \times 10^3$
$Y = B \times 10^b = 0.8200 \times 10^2$

1) Compare exponents :
   $3 - 2 = 1$
2) Align mantissas
   $X = 0.9504 \times 10^3$
   $Y = 0.08200 \times 10^3$
3) Add mantissas
   $Z = 1.0324 \times 10^3$
4) Normalize result
   $Z = 0.10324 \times 10^4$



The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$
$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain $3 - 2 = 1$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 \times 10^3$$
$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent non-pipeline floatingpoint adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

Instruction Pipeline

- **Instruction execution process lends itself naturally to pipelining**
  - **– overlap the subtasks of instruction fetch, decode and execute**

- **Instruction pipeline has six operations,**
  - Fetch instruction (FI)
  - Decode instruction (DI)
  - Calculate operands (CO)
  - Fetch operands (FO)
  - Execute instructions (EI)
  - Write result (WR)

  Overlap these operations

Instructions Fetch

- The IF stage is responsible for obtaining the requested instruction from memory. The instruction and the program counter are stored in the register as temporary storage.

Decode Instruction

- The DI stage is responsible for decoding the instruction and sending out the various control lines to the other parts of the processor.

Calculate Operands

- The CO stage is where any calculations are performed. The main component in this stage is the ALU. The ALU is made up of arithmetic, logic and capabilities.

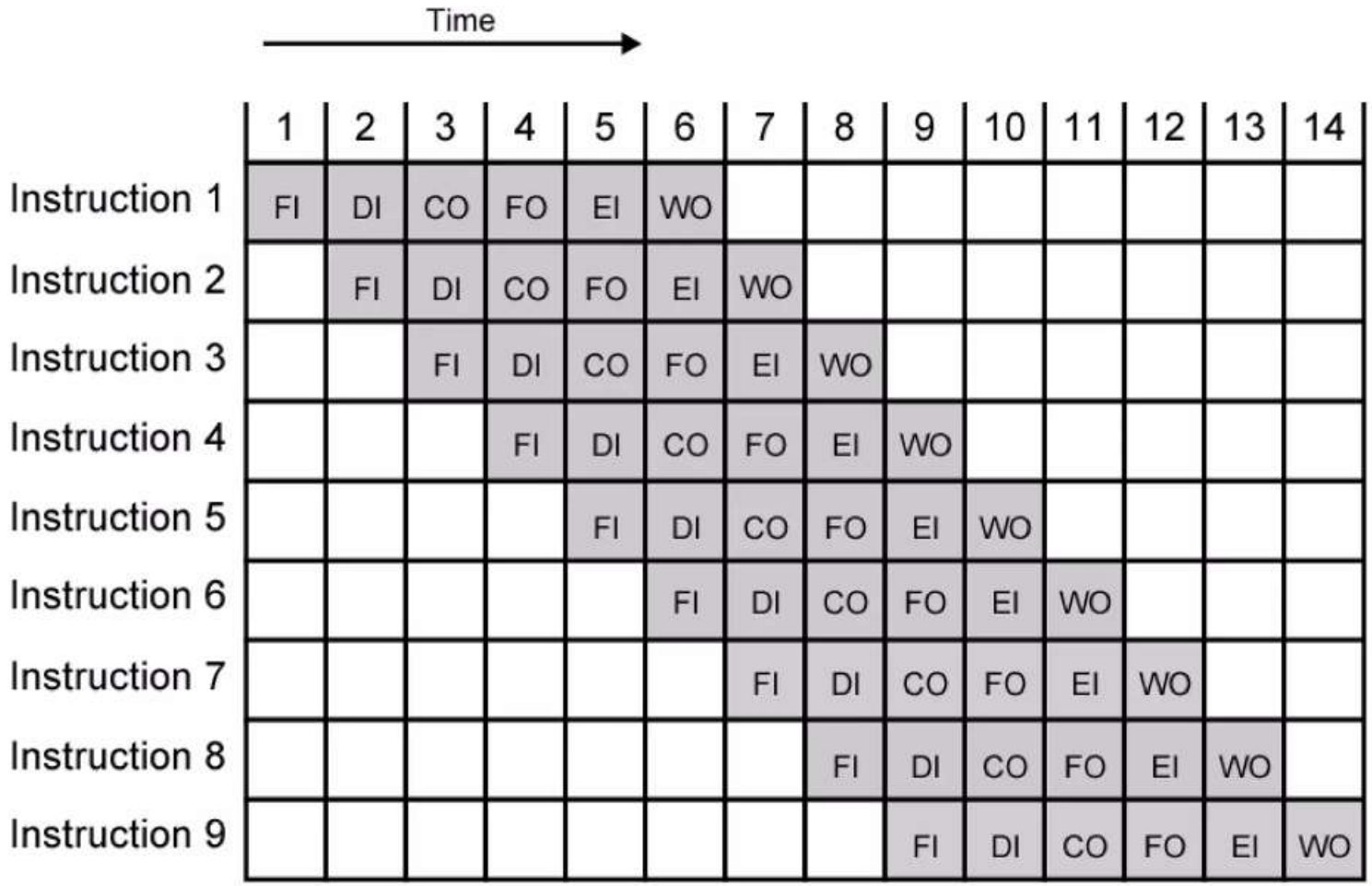
Fetch Operands and Execute Instruction

- The FO and EI stages are responsible for storing and loading values to and from memory. They also responsible for input and output from the processor respectively.

Write Operands

- The WO stage is responsible for writing the result of a calculation, memory access or input into the register file.

Timing Diagram for Instruction Pipeline Operation



Four stage pipeline

Example

1. FI: is the segment that fetches an instruction.
2. DA: is the segment that decodes the instruction and calculates the effective address.
3. FO: is the segment that fetches the operand.
4. EX: is the segment that executes the instruction.

Steps	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction 1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
Branch 3			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5					-	-	-	FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Figure: timing of instruction pipeline

Pipeline Performance: Speedup & Efficiency

k-stage **pipeline processes *n* tasks in k + (n-1) clock cycles:**

k **cycles for the first task and *n-1* cycles for the remaining *n-1* tasks**

Total time to process *n* tasks

$$T_k = [\, k + (n-1)] \, \tau$$

For the non-pipelined processor

$$T_1 = n \, k \, \tau$$

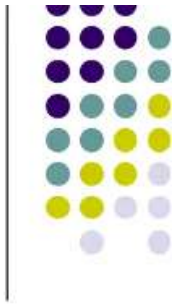
Speedup factor

$$S_k = \frac{T_1}{T_k} = \frac{n \, k \, \tau}{[\, k + (n-1)] \, \tau} = \frac{n \, k}{k + (n-1)}$$

RISC pipeline

- RISC (Reduced Instruction Set Computer)
- **1-** To use an efficient instruction pipeline
- **a)** to implement an instruction pipeline using a small number of
- suboperations, with each begin executed in one cycle.
- **b)** because the fixed length instruction format , the decoding of the
- operation can occur at the same time as register selection.
- **2-** Data transfer instruction in RISC are limited to load and store
- instruction.by using **cache memory.**
- **3-**One of major advantage of **RISC** is ability to execute instruction
- at the rate of one per clock cycle that can achieve pipeline
- segments requiring just one clock cycle.
- **4-** The compiler supported that translates the high-level language
- program into machine language program.
- .

RISC pipeline



- Instruction Cycle of Three-Stage Instruction Pipeline.
- I: Instruction Fetch from program memory
- A: Decode, Read Registers, ALU Operation
- E: Transfer the output of ALU to a register, Transfer EA to a data memory for loading or storing , Transfer branch address to the program counter.
- **Types of instructions**

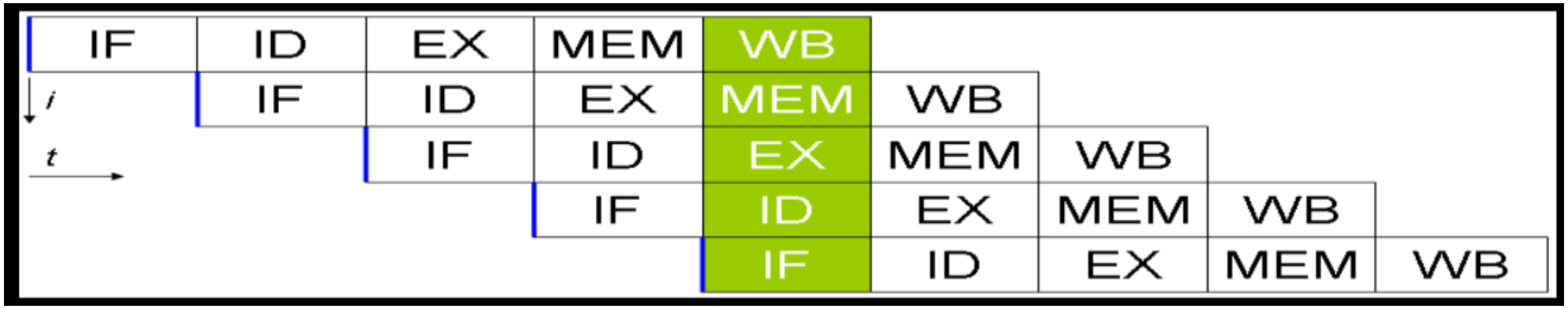
- 1- Data Manipulation Instructions
- 2- Load and Store Instructions
- 3- Program Control Instructions

The main benefit of RISC to implement instructions at the cost of one per clock cycle is continually not applicable because each instruction cannot be fetched from memory and implemented in one clock cycle correctly under all circumstances.

The method to obtain the implementation of an instruction per clock cycle is to initiate each instruction with each clock cycle and to pipeline the processor to manage the objective of single-cycle instruction execution.

RISC compiler gives support to translate the high-level language program into a machine language program. There are various issues in managing complexity about data conflicts and branch penalties are taken care of by the RISC processors, which depends on the adaptability of the compiler to identify and reduce the delays encountered with these issues.

The classic five stage RISC pipeline



Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). The vertical axis is successive instructions; the horizontal axis is time. So in the green column, the earliest instruction is in WB stage, and the latest instruction is undergoing instruction fetch.

Pipeline Performance

- **Any condition that causes a pipeline to stall is called a hazard.**

Data hazard – when an instruction depend on the result of a previous instruction, but this result is not yet available.

- **Instruction (control) hazard** – a delay in the availability of an instruction causes the pipeline to stall for example branch.

- **Structural hazard** – the situation when two instructions require the use of a given hardware resource at the same time.

Data Hazards

- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.
- Hazard occurs

  $A \leftarrow 3 + A$

  $B \leftarrow 4 \times A$
- No hazard

  $A \leftarrow 5 \times C$

  $B \leftarrow 20 + C$
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:

  Mul R2, R3, R4

  Add R5, R4, R6

Pipeline Hazards

- Pipeline, or some portion of pipeline, must stall
- Also called *pipeline bubble*
- Types of hazards
  - Resource
  - Data
  - Control

Resource Hazards

- Two (or more) instructions in pipeline need same resource
- Executed in serial rather than parallel for part of pipeline
- Also called *structural hazard*
- E.g. Assume simplified five-stage pipeline
  - Each stage takes one clock cycle
- Ideal case is new instruction enters pipeline each clock cycle
- Assume main memory has single port
- Assume instruction fetches and data reads and writes performed one at a time
- Ignore the cache
- Operand read or write cannot be performed in parallel with instruction fetch
- Fetch instruction stage must idle for one cycle fetching I3

- E.g. multiple instructions ready to enter execute instruction phase
- Single ALU

- One solution: increase available resources
  - Multiple main memory ports
  - Multiple ALUs

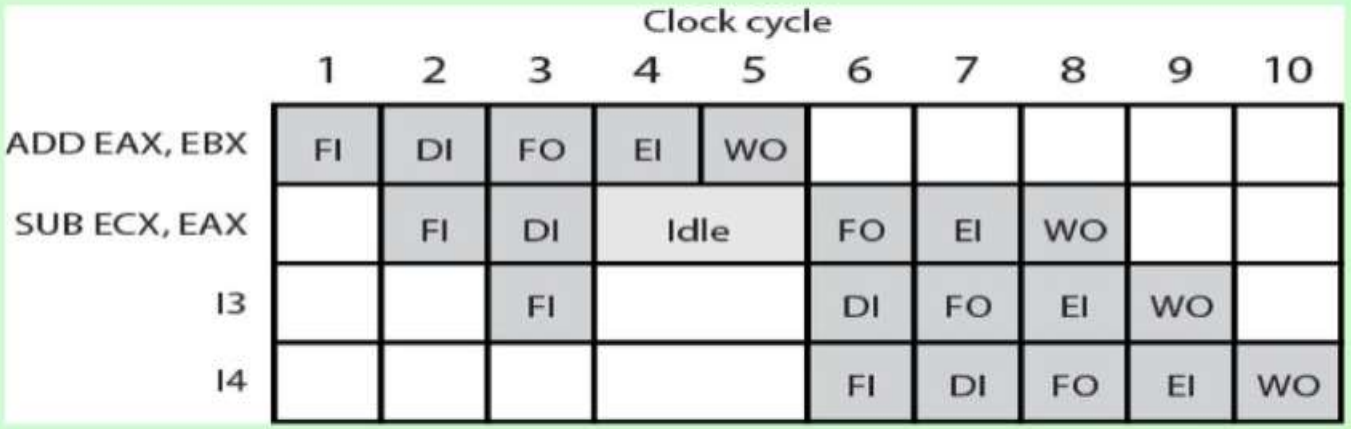
Data Hazards

- Conflict in access of an operand location
- Two instructions to be executed in sequence
- Both access a particular memory or register operand
- If in strict sequence, no problem occurs
- If in a pipeline, operand value could be updated so as to produce different result from strict sequential execution
- E.g. x86 machine instruction sequence:

- ADD EAX, EBX /* EAX = EAX + EBX
- SUB ECX, EAX /* ECX = ECX – EAX

- ADD instruction does not update EAX until end of stage 5, at clock cycle 5
- SUB instruction needs value at beginning of its stage 2, at clock cycle 4
- Pipeline must stall for two clocks cycles
- Without special hardware and specific avoidance algorithms, results in inefficient pipeline usage

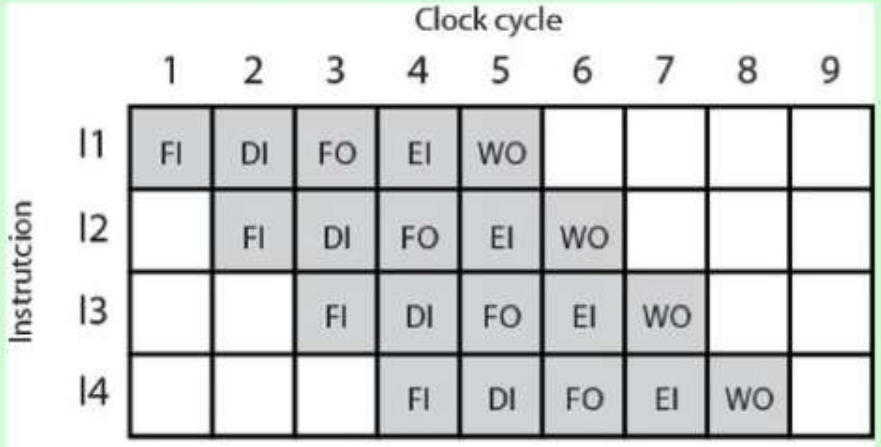
Data Hazard Diagram



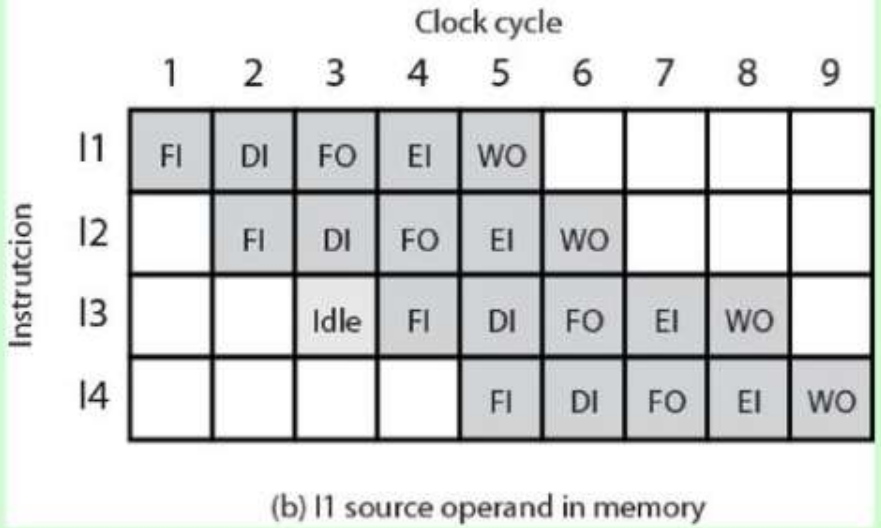
Types of Data Hazard

- Read after write (RAW), or true dependency
  - An instruction modifies a register or memory location
  - Succeeding instruction reads data in that location
  - Hazard if read takes place before write complete
- Write after read (RAW), or antidependency
  - An instruction reads a register or memory location
  - Succeeding instruction writes to location
  - Hazard if write completes before read takes place
- Write after write (WAW), or output dependency
  - Two instructions both write to same location
  - Hazard if writes take place in reverse of order intended sequence
- Previous example is RAW hazard

Resource Hazard Diagram



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Control Hazard

- Also known as *branch hazard*
- Pipeline makes wrong decision on branch prediction
- Brings instructions into pipeline that must subsequently be discarded
- Dealing with Branches
  - —Multiple Streams
  - —Prefetch Branch Target
  - —Loop buffer
  - —Branch prediction
  - —Delayed branching