Chapter 7: Pipelining and Performance

Topic

1. Modern microprocessors (CISC and RISC)
2. Pipelines: definition and usage
3. The effects of pipelining on the computer performance

   – Chapter 5-5 by Null
   – Chapter 13 by Berger(Available online)

Microprocessor Overview

- Microprocessors span a wide range of speed, power, functionality and cost
  - $0.25- for a 4-bit controller
  - $10,000+ for a space-qualified custom processor

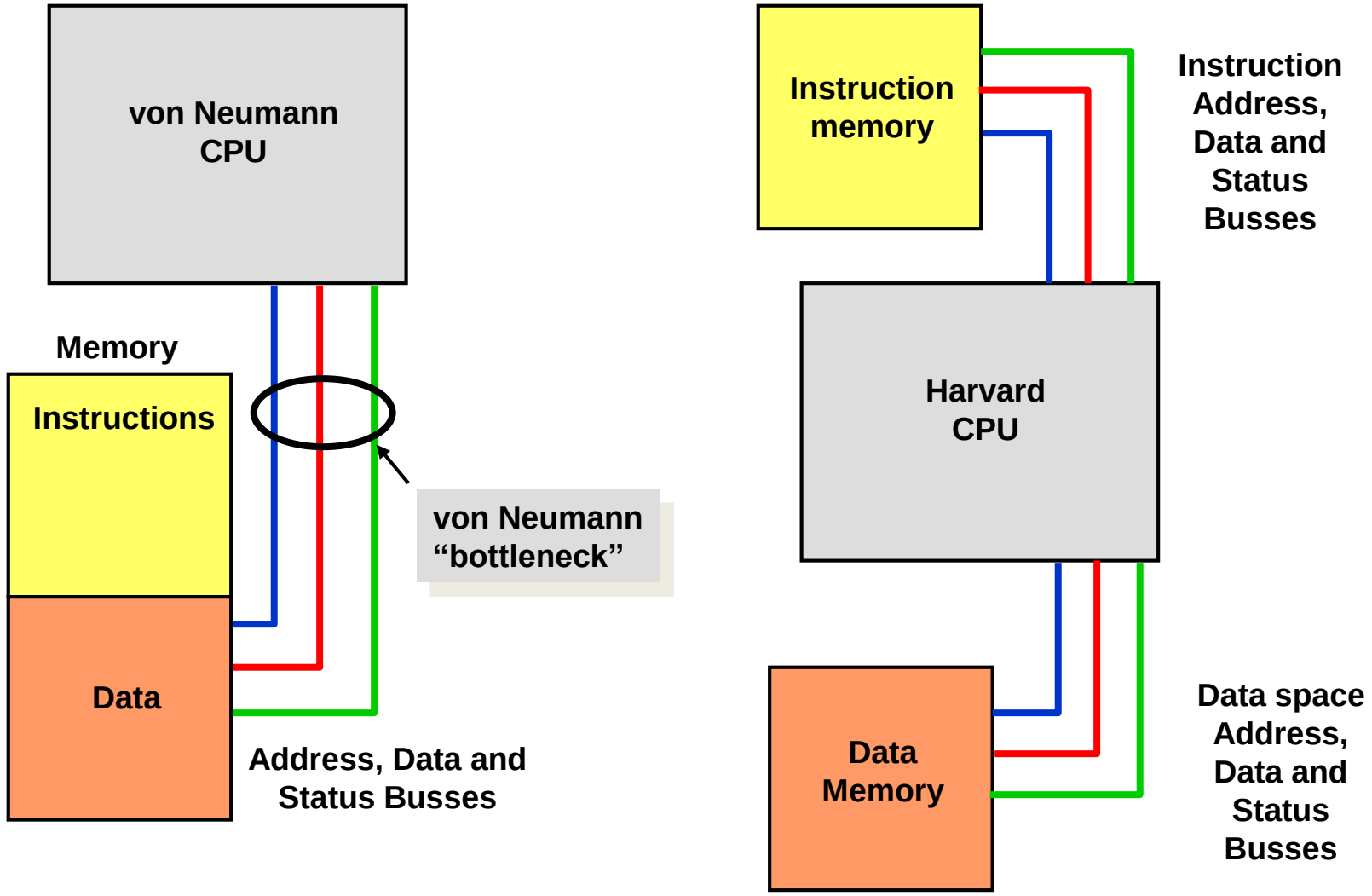
- Differentiating among microprocessors
  - **Processor architecture**: CISC, RISC, DSP
  - **Data path width**: 4, 8,16, 32, 64, 128 bits
  - **Addressable address space**: 1KB - multi-gigabyte
  - **Instruction set architecture (ISA)**: ARM, IA, 68K, PPC, MIPS
  - **Processor speed**: 1- MHz → 4+ GHz**

** Number of clock cycles per second

Processor Architecture

- **CISC** - **C**omplex **I**nstruction **S**et **C**omputer
  - Many instructions and addressing modes
    - CISC code is compact
    - Can be many clock cycles per instruction
    - Large silicon area → Higher cost per die
    - *von Neumann Architecture: s*ame memory space services instruction and data
    - **Causes *von Neumann bottleneck***
- **RISC** - **R**educed **I**nstruction **S**et **C**omputer
  - Modern architecture
  - One instruction executed per clock cycle → Very fast
  - Originally, Harvard Architecture
    - Separate memory spaces for instructions and data
    - **Avoids** the ***von Neumann bottleneck***
  - May use von Neumann with specified Cache

von Neumann and Harvard Architectures



Characteristics of RISC

- RISC processors are very **fast and efficient**
  - However, RISC code images tends to be larger
- **RISC** computers tend to prevail in **data processing applications**
  - **CISC** computers tend to prevail in **control** applications
- RISC CPU cores tend to be small, since microcode ROM is not necessary
- RISC core are very prevalent in ASIC (Application-Specific Integrated Circuit) devices
    - ARM Ltd. sells RISC CPU designs as ***Intellectual Property*** to IC builders
    - iPhone uses an ARM-based RISC microcontroller

Instruction Set Architectures (ISA)

- The PC world is dominated by the Intel Architecture – IA or X86
  - 8080 > 8086 > 80186 > 80286 > 80386 > 80486 > Pentium > …
- The embedded processor world **was** *dominated by the Motorola 68K instruction set architecture*
  - 68000 > 68020 > 68030 >68040 > 68060 > ColdFire
  - ColdFire unites a modern CISC/RISC processor architecture with backwards compatibility to the original 68K
    - Why? Because there is so much 68K code still around
- ARM (Advanced RISC Machines) is now the dominant embedded microprocessor architecture
  - Owns smart phones, and maybe tablets
- Atmel (Arduino) and TI microprocessors

Performance

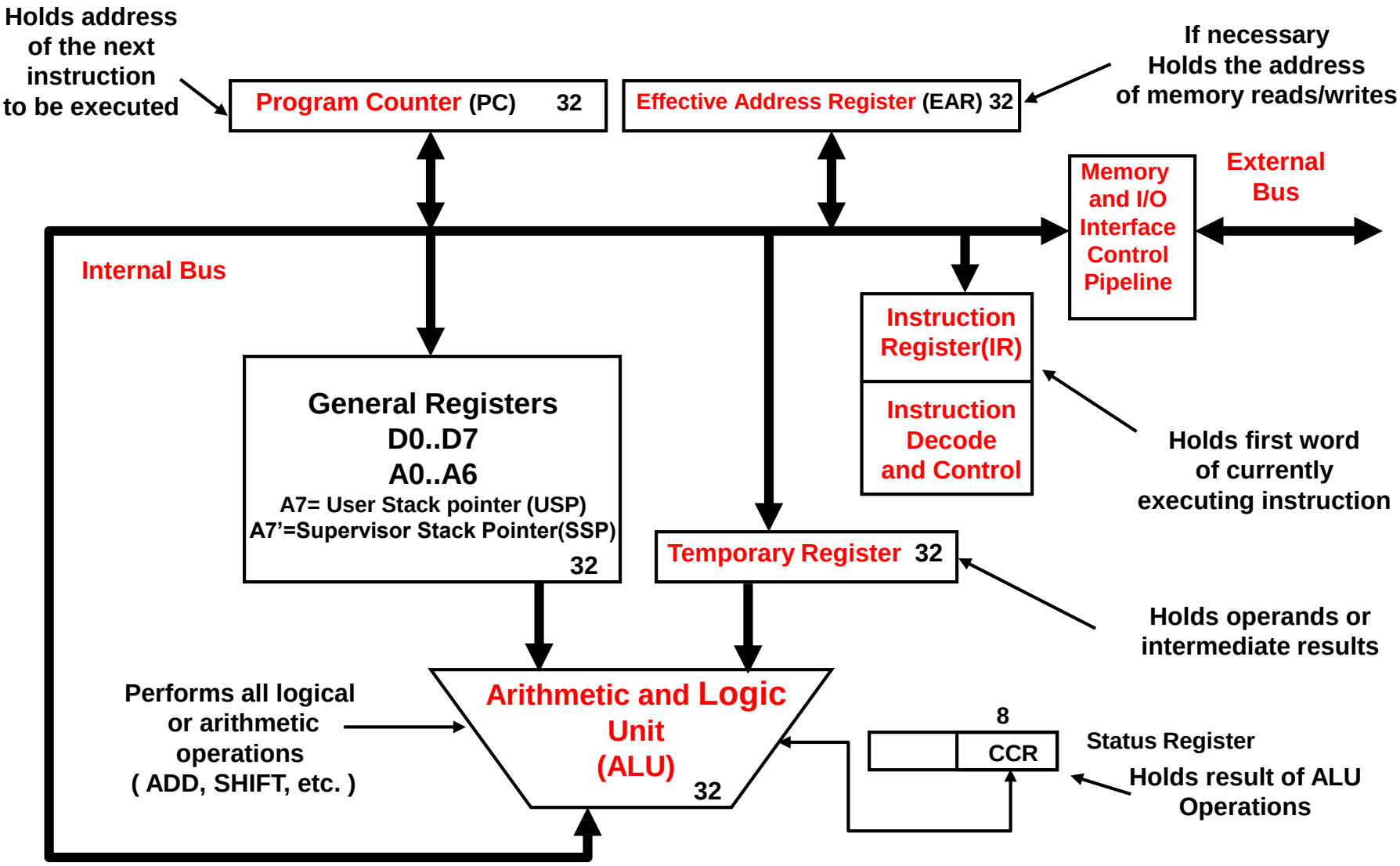
- Why we have different architectures?
- How to design for speedup with these architectures?
  - Reduce propagation delay: IC fabrication process
  - Reduce the number of gates
  - Hire a better hardware designer?
  - **Pipelining → the favorable approach**

Pipelining

- Processor is an expensive resource
  - Want to **always keep it busy** since an idle processor is wasting space, energy, time, etc.
  - Need a way to increase performance

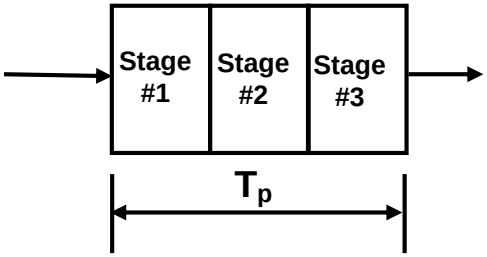
- Idea
  - Break execution task into **smaller tasks** (**stages**)
  - In **Each stage, a different hardware** component is used (**data path**)
  - Overall time to execute any one instruction is the same
  - **Next instruction is only one stage behind**
  - **Increase the *throughput*** of the processor
    - Multiple instructions within the pipeline at the same time

Hardware Organization of the MC68000

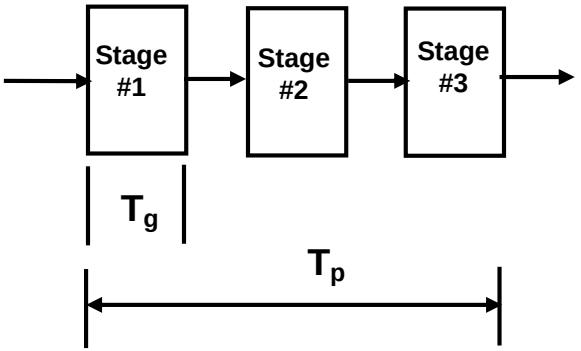


Decomposition of a Process

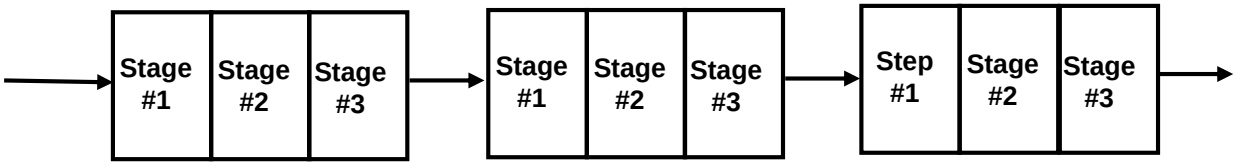
- A computational process with three **logical stages**:



- The same process is divided into three **physical stages**:
  - Each stage is handled by dedicated hardware
  - Here $T_P = N \times T_g$
  - N = Number of stages
  - T_g = Process time per stage



- A computational process with series of logical stages (three processes):
  - Processing time of one logical state: $T_p = 3 \times T_g$
  - Total processing time for 3 instructions: $3T_p = 9T_g$



Decomposition of a Process (2)

A pipelined execution of the same sequence of three processes

- We can create a pipeline because the **hardware in Stage #1** becomes idle after executing its task
- Start another task in Stage #1 before the entire process is completed
- With n instructions:

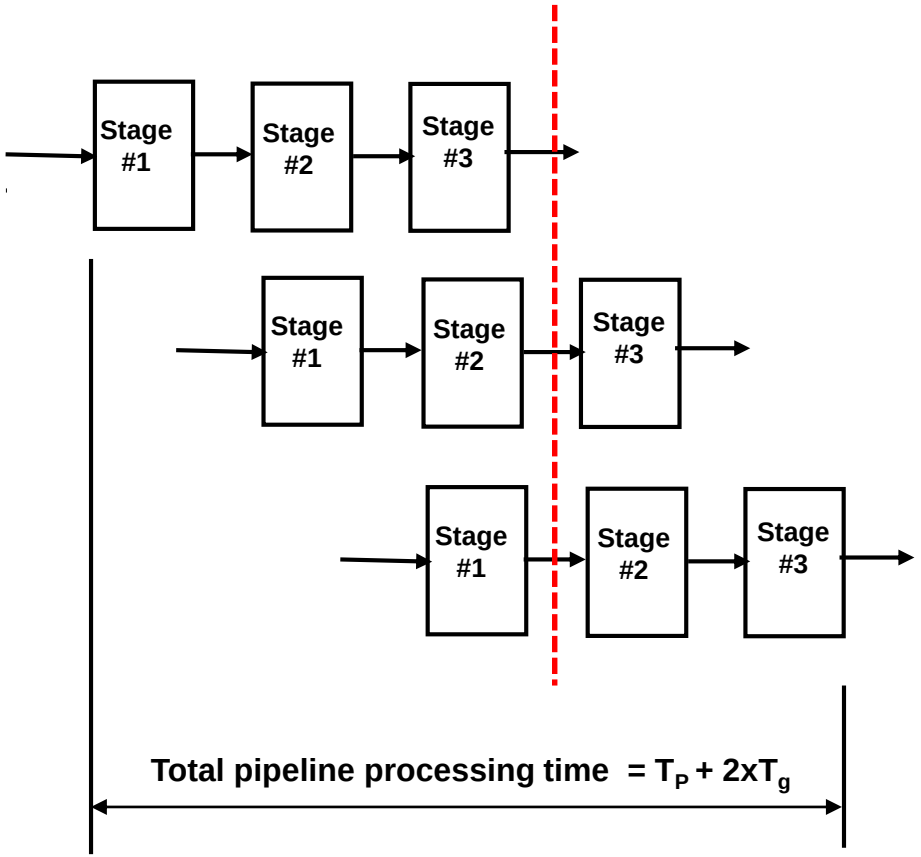
  First instruction: need all stages

  The rests: only one stage more

  Therefore, for n instruction,

  Total: 3 stage + **(n-1)** stages

First instruction | **The rest of instructions**



Class Exercise

We want to run **10 instructions** on a processor with the following features:

- **4-stage pipeline** and **100 MHz CPU** Frequency
- **Each stage** takes **2 clock cycles**

Q1) Without pipeline, how much time will elapse from the time to execute all the 10 instructions?

Q2) With pipeline, what is the total elapsed execution time for all ten instructions, assuming that there is no pipeline stalls.

Q3) What is the speedup for the 10 instructions with pipeline vs. without pipeline?

Q4) What is the speedup for the millions of millions of (infinite) instructions?

Class Exercise

We want to run 10 instructions on a processor with the following features:

- 4-stage pipeline and 100 MHz CPU Frequency
- Each stage takes 2 clock cycles

Q1) Without pipeline, how much time will elapse from the time to execute all the 10 instructions?

100MHz → 1cc = 10ns. Total time is 4*2*10*10ns = 800 ns

Q2) With pipeline, what is the total elapsed execution time for all ten instructions, assuming that there is no pipeline stalls.

(1*4*2 (first) + 9 * 2 (rest of them))*10ns = 260ns

Q3) What is the speedup for the 10 instructions with pipeline vs. without pipeline?

Speed is inverse proportional to time:

Pipeline speed/non-pipeline speed = non-pipeline time/pipe time = 800/260 = 3.0769

Q4) What is the speedup for the millions of millions of (infinite) instructions?

Assume with n instructions: 4n/(n+3)→4 as n goes infinite

RISC Architecture: Optimized for Pipelining

- In the early 1980's computer scientists studied the frequency of use of computer instructions by programmers and high-level languages (**CISC**)
  - At that time, instruction sets were characterized by many instructions and many addressing modes
    - Wide range of clock cycles per instruction
    - Variable length instructions
  - **Difficult to optimize the pipelining of instructions because of the variability**

- Began developing architectures with **fewer instructions**
  - Called ***RISC,*** for **R**educed **I**nstruction **S**et **A**rchitecture
- **Pipelining**
  - All instructions are the same length (RISC)
  - Just a few instruction formats (RISC)
  - Memory operands appear only in loads and stores (RISC)

Pipeline Issues

- Number of stages to consider
  - How many? What is the optimized number of stages?
  - What if too many?
    - Faster speed but **more sensitive to hazards**
  - What if too few?
    - Slower speed but **more tolerant of hazards**

- Example

  A certain RISC processor (MIPS Architecture) has a 4-stage pipeline
  - **Instruction Fetch** and PC Increment
  - **Instruction Decode**
  - **Operand Fetch (Operand Address)**
  - **Execution/Write Back**

Pipeline Hazards

- **Structural hazards** or **Resource conflicts**: only one memory or only one CPU

- **Data hazards** or **Data dependency**: an instruction depends on a previous instruction

- **Control hazards**: branch instructions

Structural Hazards

- General Solution
  - Reconfigure the architecture (to resolve some conflicts)
- Data and instruction can not be fetched at the same time
  - Solution
    - Build separate paths for data and instruction
- Multiple instructions are waiting for executing
  - Solution
    - Have multiple ALU units
    - Have multiple ports into main memory

Data Hazards

- **R**ead **A**fter **W**rite (**RAW**), or true dependency

Example:

```
MOVE.W D1,D0
ADD.W  D0,D3
```

- **W**rite **A**fter **W**rite (**WAW**), or output dependency

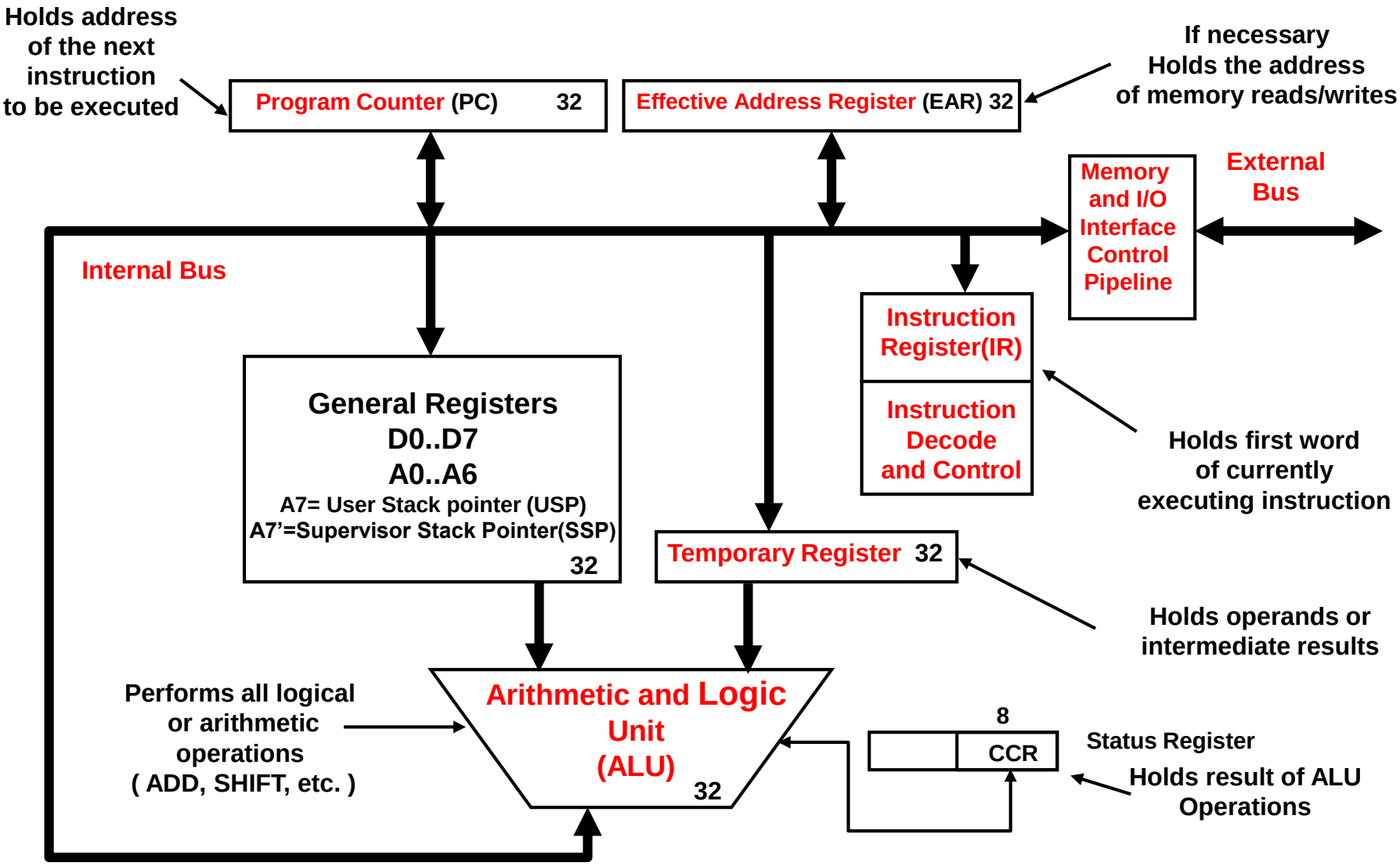
Example:

```
MOVE.W D1,D0
ADD.W  D3,D0
```

- **W**rite **A**fter **R**ead (**WAR**): hazard occurs ***only if write is faster than read (very rare to happen)***

- Solutions
  - Special hardware can be added to insert a brief delay, or insert a specified data path
  - **Reorder the instructions**

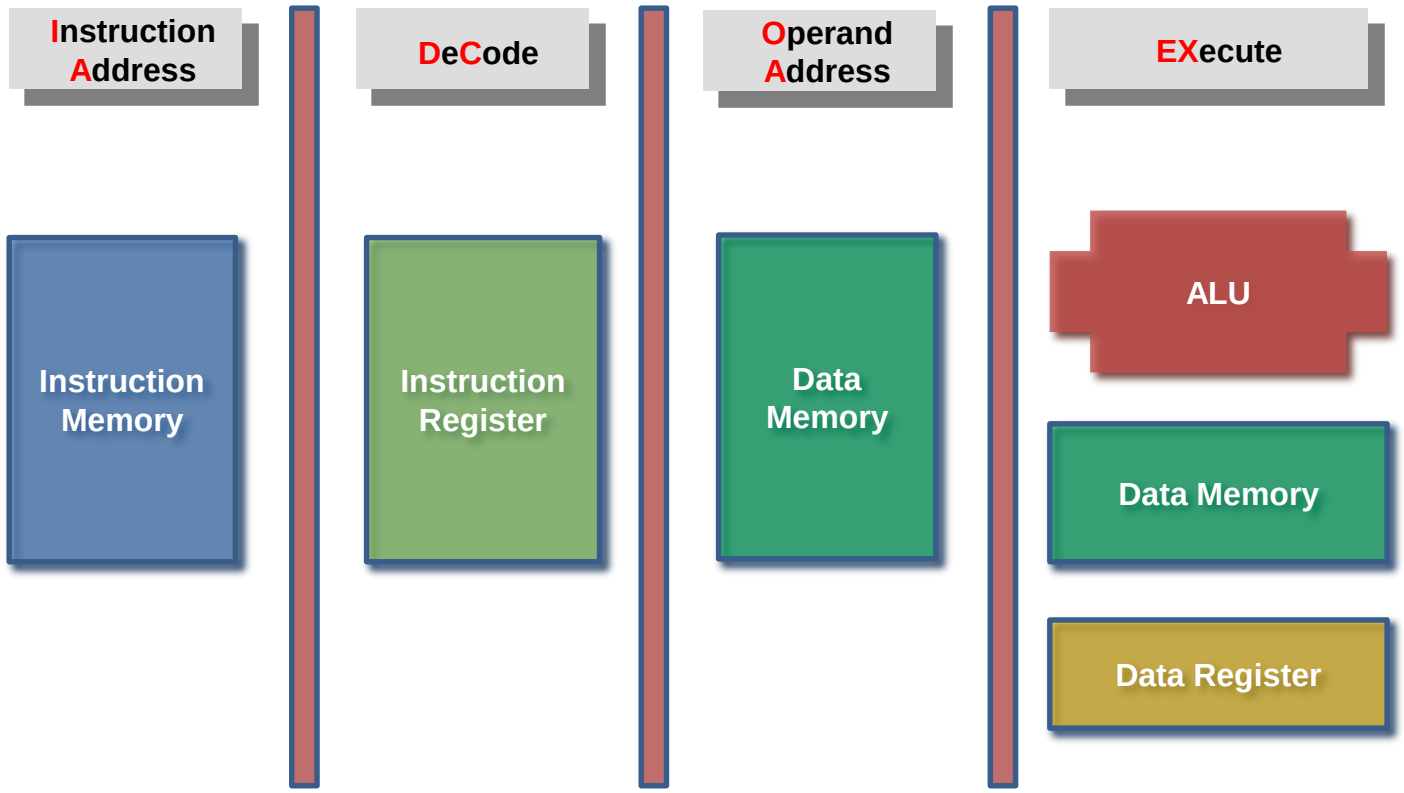
Hardware Organization of the MC68000



Class Exercise 2

Consider the following snippet of 68K assembly language instructions, and assume that it has **4-stages in a pipelined microprocessor**.

Suppose we have **separate instruction and data paths to memory**.



Class Exercise 2

Q1) Reorder the instructions so that they may execute more efficiently in a pipelined microprocessor.

```
MOVE.W      D2,D4
SUB.W       D4,D3
ADD.W       D3,D1
MOVE.W      #$3400,D2
MO VE.W     #$F6AA,D4
LEA         (A3),A6
ADDA.W      D5,A2
```

Q2) Suppose each stage takes 1 clock cycle, how many clock cycles can be saved after reordering the instructions?

Class Exercise 2

Q1) Suppose each stage takes 1 clock cycle, how many clock cycles can be saved after reordering the instructions?

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
MOVE.W D2,D4	IA	DC	OA	EX								
SUB.W D4,D3		IA	DC		OA	EX						
ADD.W D3,D1			IA	DC			OA	EX				
MOVE.W #$3400,D2				IA	DC			OA	EX			
MOVE.W #$F6AA,D4					IA	DC			OA	EX		
LEA (A3),A6						IA	DC			OA	EX	
ADDA.W D5,A2							IA	DC			OA	EX

Originally, it takes 12 clock cycles!

Class Exercise 2

Q2) Reorder the instructions so that they may execute more efficiently in a pipelined microprocessor.

MOVE.W	D2,D4
SUB.W	D4,D3
ADD.W	D3,D1
MOVE.W	#$3400,D2
MO VE.W	#$F6AA,D4
LEA	(A3),A6
ADDA.W	D5,A2

MOVE.W	D2,D4
MOVE.W	#$3400,D2
SUB.W	D4,D3
LEA	(A3),A6
ADD.W	D3,D1
MO VE.W	#$F6AA,D4
ADDA.W	D5,A2

Class Exercise 2

Q2) Suppose each stage takes 1 clock cycle, how many clock cycles can be saved after reordering the instructions?

Instruction	1	2	3	4	5	6	7	8	9	10
MOVE.W D2,D4	IA	DC	OA	EX						
MOVE.W #$3400,D2		IA	DC	OA	EX					
SUB.W D4,D3			IA	DC	OA	EX				
LEA (A3),A6				IA	DC	OA	EX			
ADD.W D3,D1					IA	DC	OA	EX		
MOVE.W #$F6AA,D4						IA	DC	OA	EX	
ADDA.W D5,A2							IA	DC	OA	EX

After reordering, it takes 10 clock cycles. So, saves 2 clock cycles!

Conditional Hazards

- Until the instruction is actually executed, ***it is impossible to determine whether the branch will be taken or not***.

- Solutions
  - **Branch prediction**: Make CPU do educated guesses. If it hits, then improvement, otherwise just flash the loaded instruction.
  - **Dynamic branch prediction**: Use statistical prediction, by incrementing the taken branches, and decrementing the untaken branches.
  - **Branch target caches**: Make special on-chip caches to store some previously taken branches.
  - **Delayed branch**: Reorder instructions.

Characteristics of a RISC Processor

- RISC Architecture has many of the following characteristics:
  - Many **general purpose** registers
    - Hold data or address
  - Instructions are **conceptually simple**
    - Uniform length
    - One (or very few) instruction formats
  - **Little or no overlapping functionality of instructions**
  - **All arithmetic operations are between registers**
  - Memory/Register transfers are exclusively ***LOADS*** and ***STORES***
    - LOAD = memory to register
    - STORE = register to memory
  - One (or very few) addressing mode

Optimizing RISC Compilers

- RISC architectures use ***optimizing compilers*** to take advantage of the speed-enhancing features of the processor
  - Performance is very sensitive to compiler
- Compilers will ***rearrange instructions***
  - Take advantage of the processor's parallelism
- Some techniques are (Wherever possible):
  - Fill load-delay and branch-delay slots with independent instructions
  - Aggressively use registers, avoid memory load/store penalties
  - Move LOAD instructions as early as possible in the instruction stream to avoid load-delay penalties
  - Move instructions that evaluate branch addresses as early as possible in the instruction stream to avoid branch-delay penalties

Optimizing RISC Compilers (2)

- Compiler techniques (continued)
  - Move condition code testing instructions as early as possible so that branch-test sequences can be optimized
  - Fix branch tests so that most common result of the test is not the branch
  - Unroll loops (inline loop code) to avoid penalty of non-sequential instruction execution
    - Try to maximize the size of the *Basic Block*

Wrap-up

- Pipelining improves performance by increasing instruction throughput, not decreasing instruction execution time
- Complications arise from hazards
- To make processors faster, try to start more than one instruction at a time
- Code rearranged (scheduled by compiler) to improve pipeline performance
- Hardware scheduling is used by modern microprocessors