Computer Architecture
CS-213

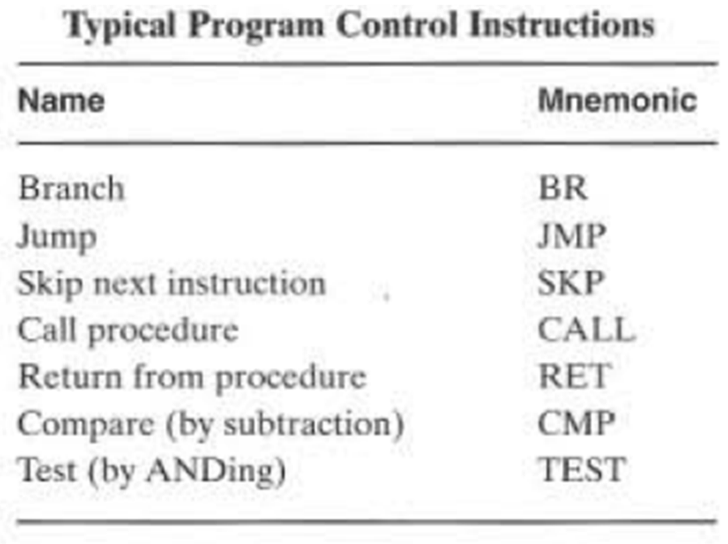
Lecture - 8

Outline

- ➢ Program Control Instructions
- Branch Instructions
- Procedure **Call** and **Return** Instructions
- ➢ Program Interrupts
- Difference between Procedure call and Interrupt
- Types of Interrupts
- Interrupt Servicing Mechanism

Program Control Instructions

- A program control instruction changes address value in the PC and hence the normal flow of execution.
- Change in PC causes a break in the execution of instructions.
- It is an important feature of the computers since it provides the control over the flow of the program and provides the capability to branch to different program segments.

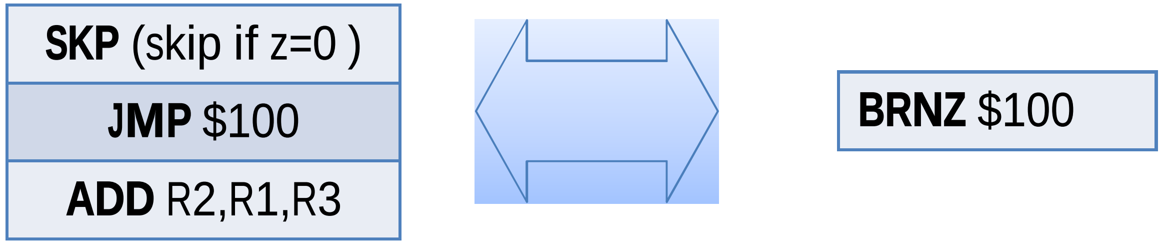


Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip next instruction	SKP
Call procedure	CALL
Return from procedure	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

Typical Program Control Instructions

- Branch (BR) and Jump (JMP) instructions are used sometimes interchangeably but, they are different.
- Branch and Jump instructions usually differ in addressing modes.
- Usually Jump is used to refer to unconditional version of branch.
- Skip (SKP) instructions is used to skip one(next) instruction. It can be conditional or unconditional. It does not need an address field.
- In case of conditional skip instruction, the combination of conditional skip and a unconditional branch can be used an alternative of conditional branch. But, storing two instructions will take extra space.
- In skip instruction we increment the PC in execution stage, effectively incrementing it by 2.



Typical Program Control Instructions

- Compare (CMP) instruction performs a comparison via a subtraction, with difference not retained.
- The comparison causes one of the three following operations

a. A conditional Branch (3ree addresses [2 registers and 1 memory])

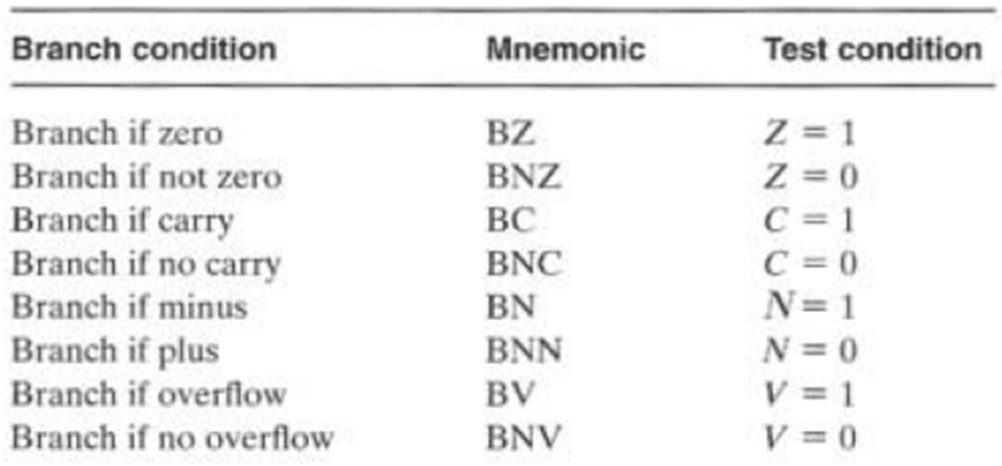
b. Change in the contents of a register (3ree addresses)

c. Sets or resets stored status bits (2 addresses), this type of instruction is usually followed by a branch instruction to conditionally check the status bit and perform a branch.

- Similarly test (TEST) instructions performs the AND of two operands without retaining the result.
- It also causes one the above three functions.

Conditional Branch Instructions

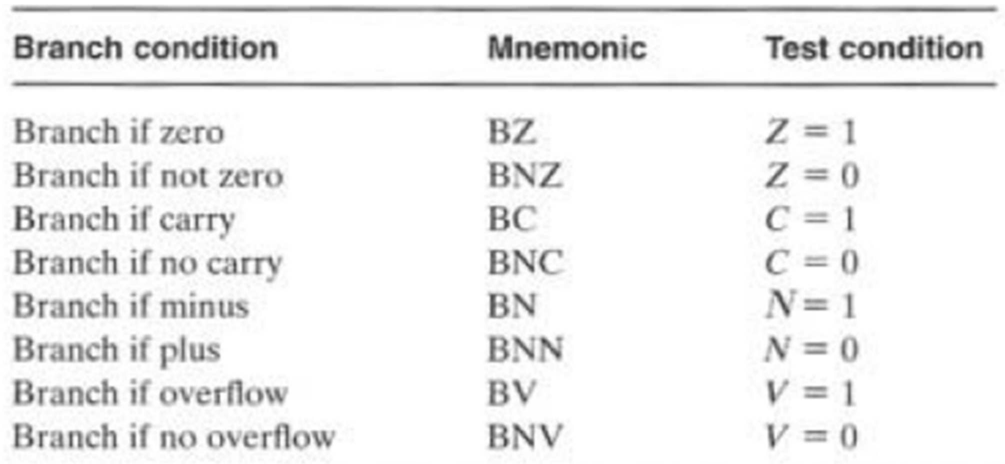
- A conditional branch instruction is a branch instruction that may or may not cause a transfer of control depending on the value of stored bits in the PSR (processor status register).
- Each conditional branch instruction tests a different combination of Status bits for a condition.
- If the condition is true, control is transferred to the effective address (PC←Add). If the condition is false, the program continues with the next instruction (PC←PC+1).
- Below is a list of Conditional Branch instructions, letter 'N' stands for NOT.



Branch condition	Mnemonic	Test condition
Branch if zero	BZ	$Z = 1$
Branch if not zero	BNZ	$Z = 0$
Branch if carry	BC	$C = 1$
Branch if no carry	BNC	$C = 0$
Branch if minus	BN	$N = 1$
Branch if plus	BNN	$N = 0$
Branch if overflow	BV	$V = 1$
Branch if no overflow	BNV	$V = 0$

Conditional Branch Instructions

- ‘C’ represents the carry, or borrow after arithmetic addition or subtraction.
- ‘N’ represents the leftmost bit of the result of the operation i.e. sign bit.
- ‘V’ is for overflow i.e. if the sign of the result is changed (inverted).
- ‘Z’ is for zero i.e., to check whether the result of an operation is zero (Z=1) or not zero (Z=0).



Branch condition	Mnemonic	Test condition
Branch if zero	BZ	$Z = 1$
Branch if not zero	BNZ	$Z = 0$
Branch if carry	BC	$C = 1$
Branch if no carry	BNC	$C = 0$
Branch if minus	BN	$N = 1$
Branch if plus	BNN	$N = 0$
Branch if overflow	BV	$V = 1$
Branch if no overflow	BNV	$V = 0$

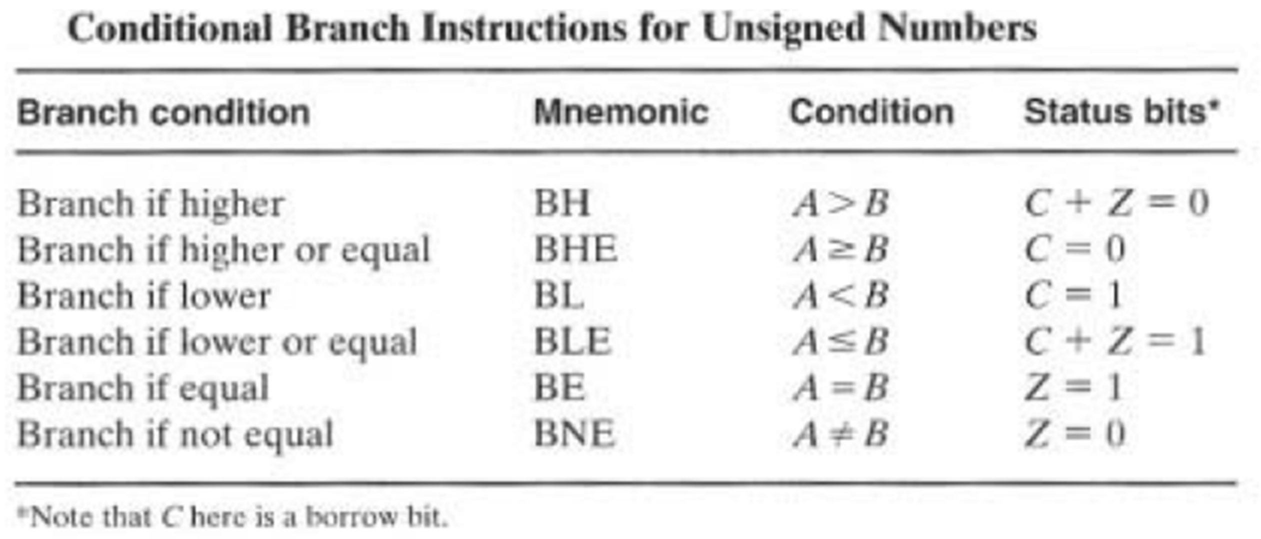
Comparison Branch Instructions

- Some branch instructions are a combination of compare and conditional branch instructions. They are run after the compare instruction has performed the comparison and status bits are updated.
- Different status bits are checked for signed and unsigned numbers.
- Keep in mind that
- **A≥B** is complement of **A<B** and

A≤B is complement of **A>B**. That means if we know the condition of status bits for one, the condition for the other complementary relation is obtained by complement.

Comparison Branch Instructions

- For unsigned numbers.

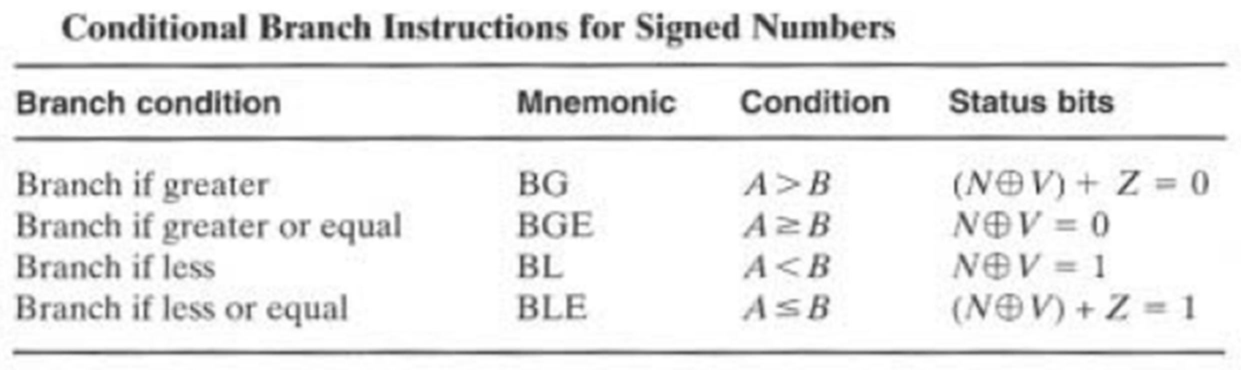


Conditional Branch Instructions for Unsigned Numbers

Branch condition	Mnemonic	Condition	Status bits*
Branch if higher	BH	$A > B$	$C + Z = 0$
Branch if higher or equal	BHE	$A \geq B$	$C = 0$
Branch if lower	BL	$A < B$	$C = 1$
Branch if lower or equal	BLE	$A \leq B$	$C + Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

*Note that *C* here is a borrow bit.

- For signed numbers.



Conditional Branch Instructions for Signed Numbers

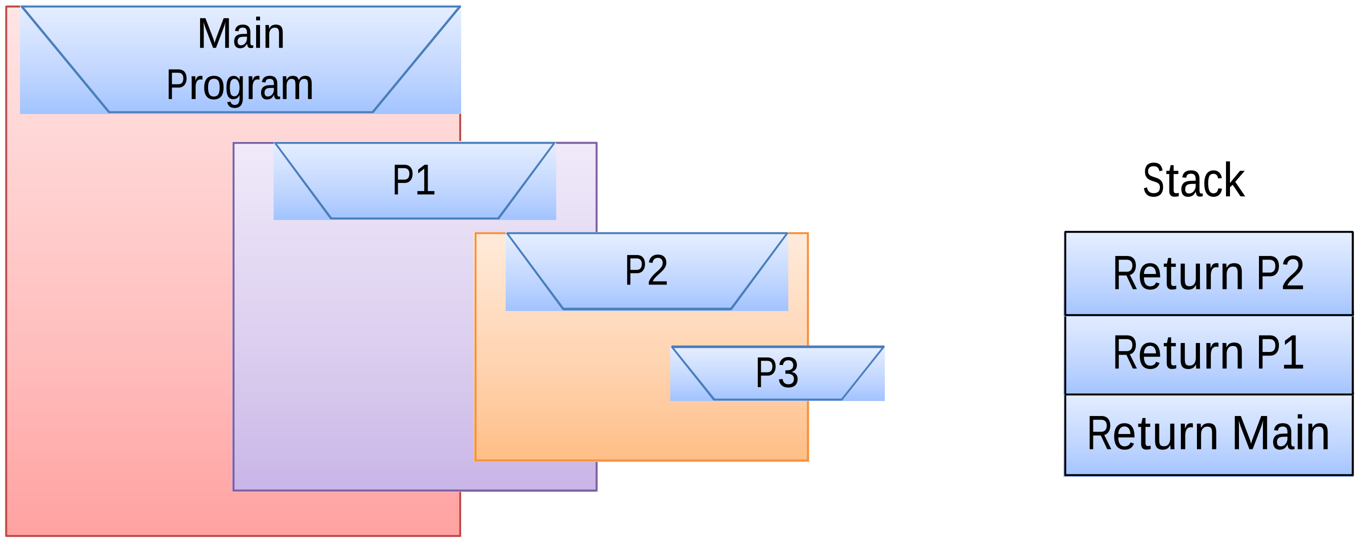
Branch condition	Mnemonic	Condition	Status bits
Branch if greater	BG	$A > B$	$(N \oplus V) + Z = 0$
Branch if greater or equal	BGE	$A \geq B$	$N \oplus V = 0$
Branch if less	BL	$A < B$	$N \oplus V = 1$
Branch if less or equal	BLE	$A \leq B$	$(N \oplus V) + Z = 1$

Procedure Call and Return Instructions

- A procedure is a self contained sequence of instructions that performs a given task.
- It is also called a subroutine.
- When a procedure is called, the starting address of the procedure is stored in the PC and the instruction following the current instruction is temporarily stored elsewhere. When the procedure (block of code) is executed, the return is made to the main program by loading the PC with its old value.
- Instruction following the procedure call is called continuation point and the corresponding address is called the return address.
- It is actually a low level form of functions in C++. (e.g., square(22);)
- Procedure can also be called within another procedure.
- The final instruction of every procedure must be return to the calling program.

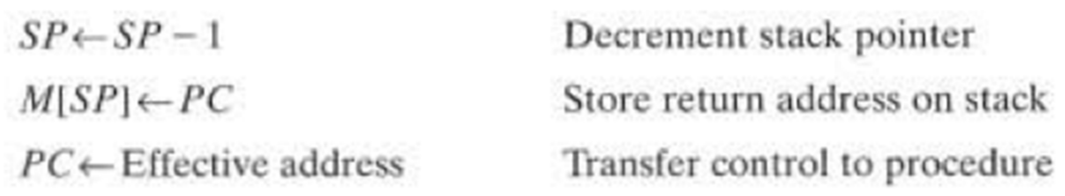
Procedure Call and Return Instructions

- The return address can be stored in memory, register or stack.
- Stack is preferred because of its ease of access when we need to call a procedure inside another procedure. In that case that the return address at the TOS (top of stack) is always to the program which called the current procedure.



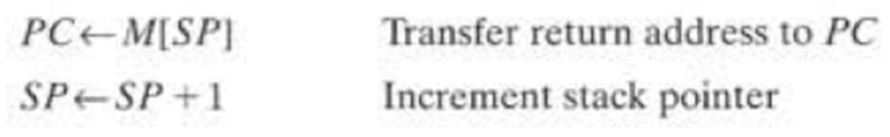
Procedure Call and Return Instructions

- For calling Procedure



$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$PC \leftarrow$ Effective address	Transfer control to procedure

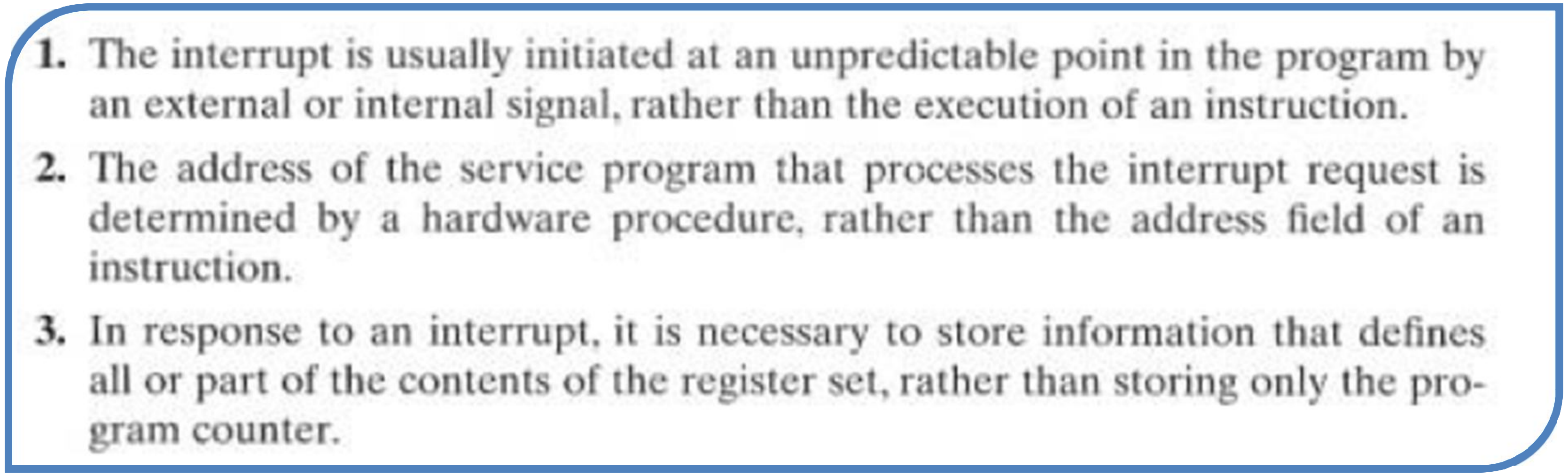
- For Return



$PC \leftarrow M[SP]$	Transfer return address to PC
$SP \leftarrow SP + 1$	Increment stack pointer

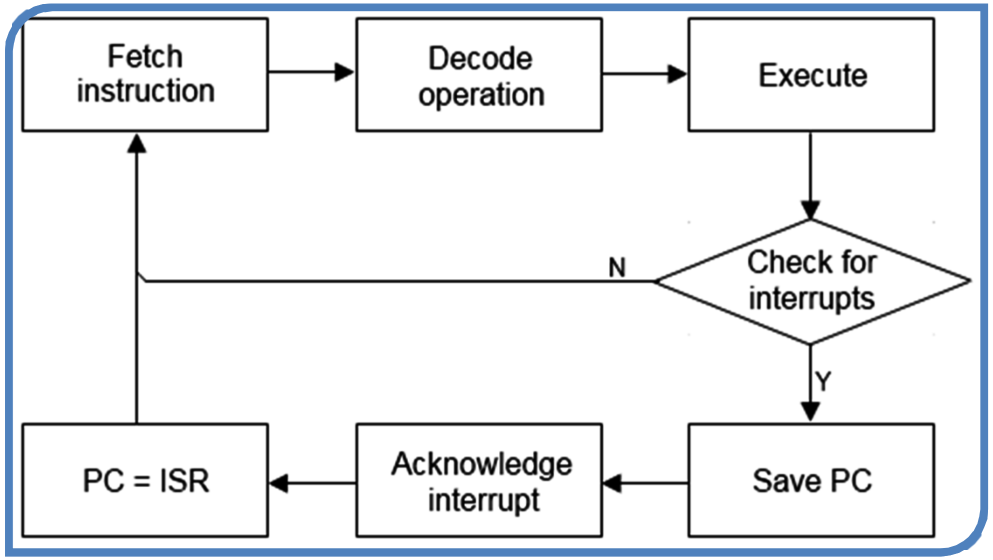
Program Interrupts

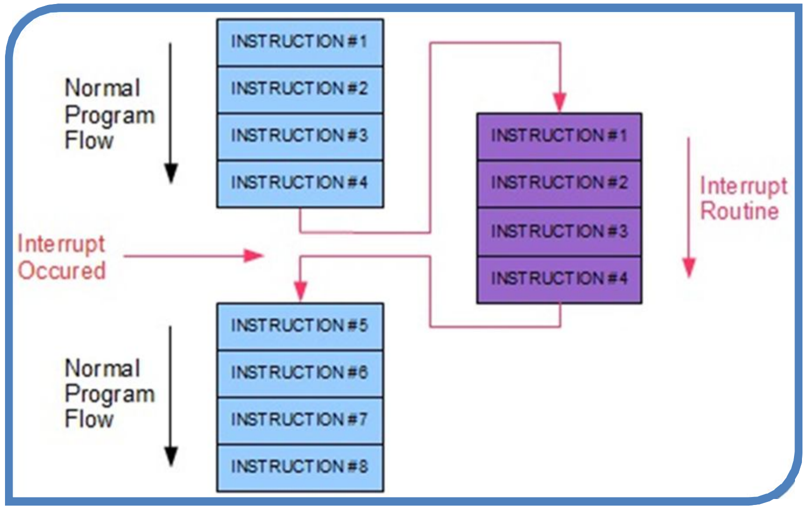
- An interrupt transfers control from a program that is currently running to another program as a result of externally or internally generated request.
- The procedure for servicing the interrupt in this case is called the interrupt service routine (ISR).



1. The interrupt is usually initiated at an unpredictable point in the program by an external or internal signal, rather than the execution of an instruction.
2. The address of the service program that processes the interrupt request is determined by a hardware procedure, rather than the address field of an instruction.
3. In response to an interrupt, it is necessary to store information that defines all or part of the contents of the register set, rather than storing only the program counter.

Program Interrupts





Program Interrupts

- Usually a computer runs in two modes i.e. system mode and user mode.
- They differ in the privilege level i.e., whether they are allowed to execute certain instructions or not.
- System mode is basically operating system, it has more privilege than the user mode as it is allowed to perform certain tasks which cannot be performed by operating in the user mode.
- Application softwares are run in user mode.
- Usually mode of the processor is indicated by PSR.

Types of Interrupts

- An interrupt has three types

1. External Interrupts
2. Internal Interrupts
3. Software Interrupts

- External and Internal interrupts are both called hardware interrupts.
- External interrupts come from input or output devices, from timing devices, from a circuit monitoring the power supply. Or from any other external source.
- Conditions that cause external interrupts are an input or output device requesting a transfer of data, the external device completing a transfer of data, the time-out of an event, or an impending power failure.

Types of Interrupts

- Internal interrupts arise from the invalid or erroneous use of an instruction or data. Internal interrupts are also called traps.
- Examples of interrupts caused by internal conditions are an arithmetic overflow, an attempt to divide by zero, an invalid opcode, a memory stack overflow and a protection violation.
- A protection violation is an attempt to address an area of memory that is not supposed to be accessed by the currently executing program.
- The service programs that process internal interrupts, determine the corrective measure to be taken in each case.

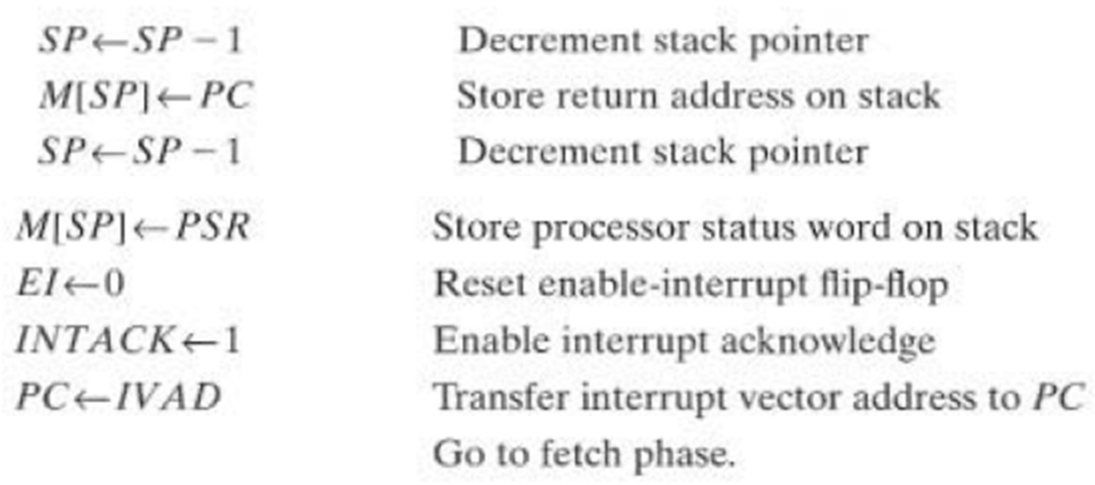
Types of Interrupts

- Interrupt generated by executing an instruction is called software interrupt. Software interrupts are generally used to make system calls i.e. to request operating system to perform an I/O operation or to run a new program. Therefore these type of interrupts request the transfer of mode from user to system.

- In C++, A cout or cin statement would generate a software interrupt because it would make a system call to print something.

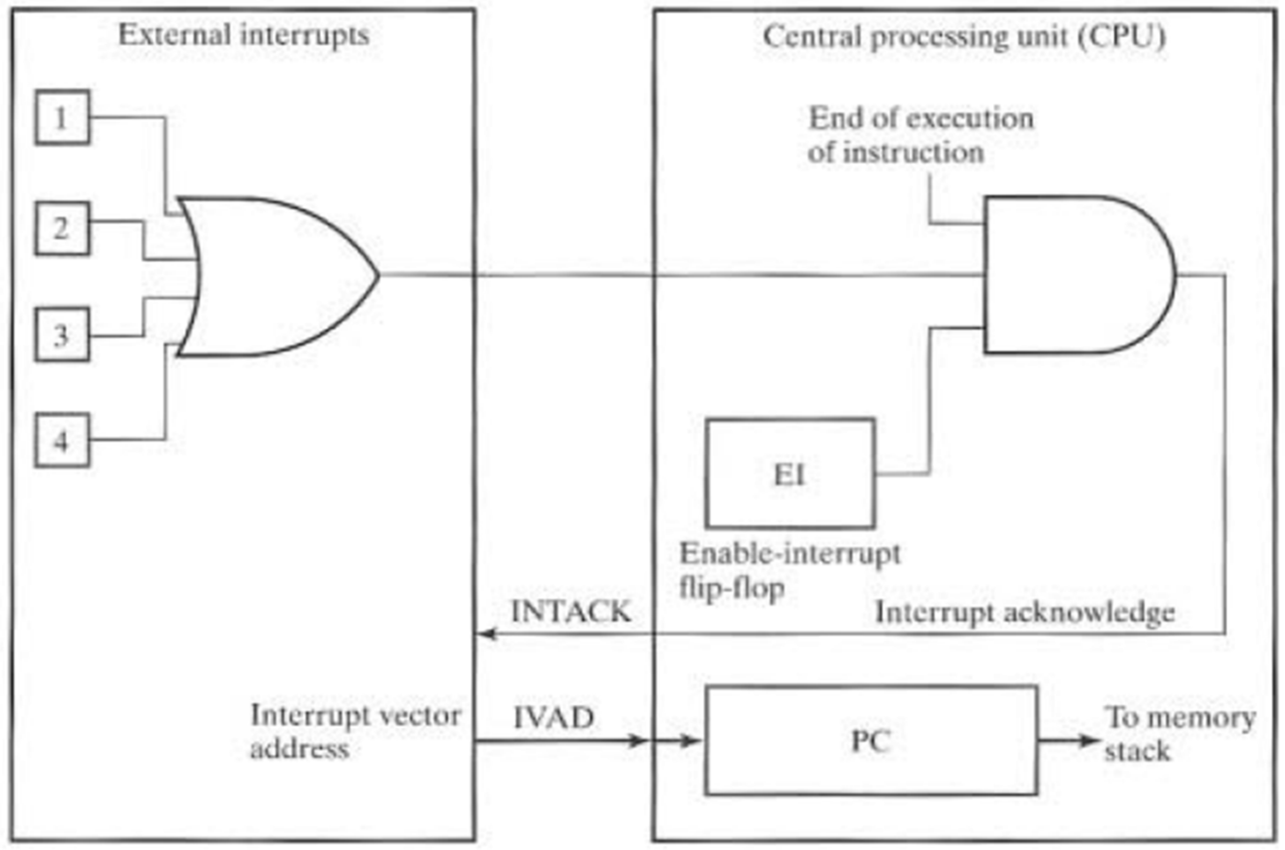
Processing External Interrupts

- IVAD is interrupt vector address.
- ENI stands for enable interrupt
- DSI stands for disable interrupt. These instructions are used to enable or disable a set of interrupts.
- INTACK is signal that is used by the processor to tell the device that interrupt is now being processed and send IVAD.
- Following is a sequence of instructions that is executed while servicing an interrupt request.



$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on stack
$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PSR$	Store processor status word on stack
$EI \leftarrow 0$	Reset enable-interrupt flip-flop
$INTACK \leftarrow 1$	Enable interrupt acknowledge
$PC \leftarrow IVAD$	Transfer interrupt vector address to PC
	Go to fetch phase.

Processing External Interrupts



Polling Vs Interrupt

- In polling, the processor waits for the request to come and sits idly until a request arrives, while in interrupt, when there is no interrupt being processed, the processor is busy in doing other tasks.
- Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring.

Lecture-8

Part-2

Outline

- Performance
- Defining Performance
- Measuring Performance
- CPU Performance and Its factors
- Instruction Performance
- The classic CPU Performance Equation

Defining Performance

- For some program running on machine X,

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

- "X is n times faster than Y“

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = n$$

Example:

- Computer A runs a program in 10 seconds while
- Computer B runs the same program in 15 seconds
- How much faster is A compared to B

Using above equation We have

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{15}{10} = 1.5$$

A is 1.5 times faster than B.

Measuring Performance

- Time is a measure of performance
- Program execution time is measured in seconds per program
- Elapsed Time/Response Time: It is the total time to complete a task including
- disk accesses
- memory accesses
- input/output activity
- operating system overhead

- A throughput measure is an amount of something per unit time. For processors, the number of instructions executed per unit time is an important component of performance.

➢ Embedded and desktop computers are more focused on **response time.**

➢ While Servers are more focused on **throughput.**

- Replacing a microprocessor with a faster one would increase **both**.
- Adding another processor to a multiprocessor system only increases **throughput.**

Measuring Performance

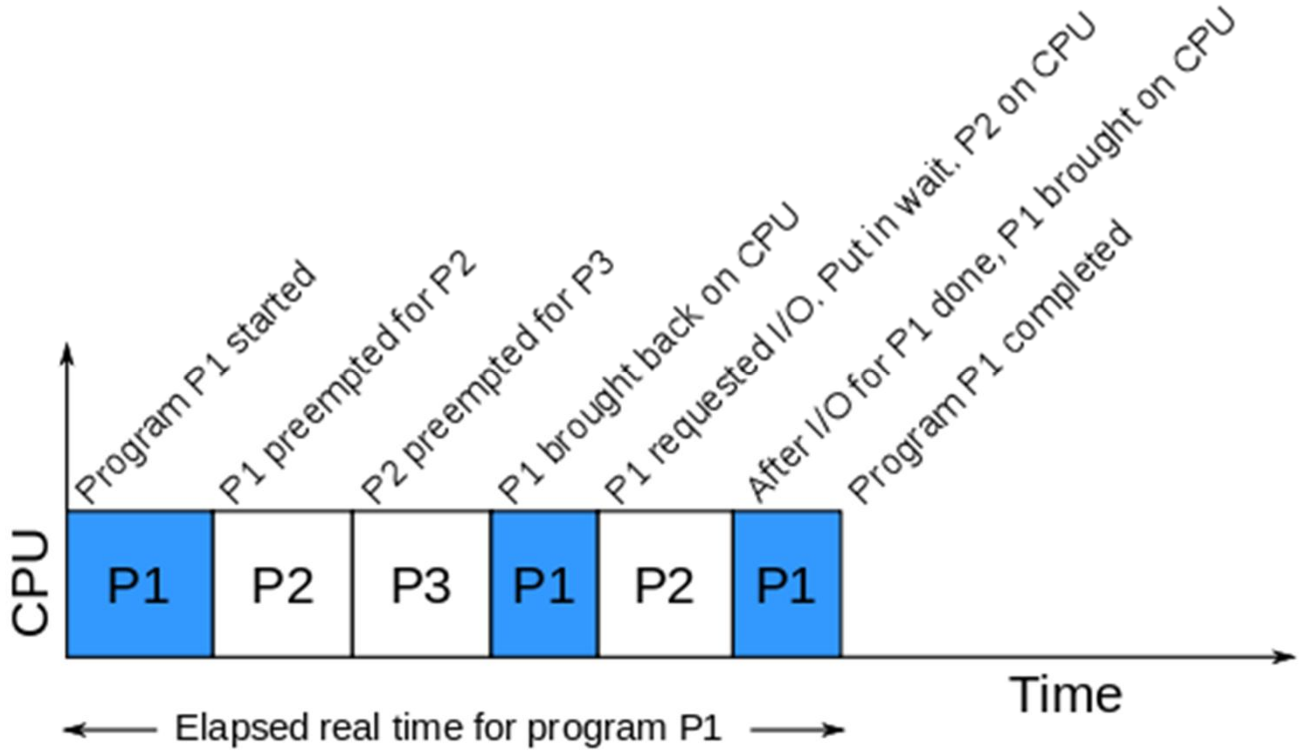
- A processor may work on several programs simultaneously in order to increase throughput there we want to know the time it spends on our program.
- CPU time:
- CPU time is the amount of time for which a central processing unit (CPU) was used for processing instructions of a computer program, as opposed to, for example, waiting for input/output (I/O) operations.

  It can be further divided into two categories

  – System CPU Time; time spent in OS performing tasks on behalf of the program

  – User CPU Time; time spent in the program

Measuring Performance



Measuring Performance

- Clock
- Instead of using seconds to measure execution time, often we use clock cycles, clock ticks, clock periods, clocks, or cycles.
- Clock rate (frequency) = cycles per second.
- Measured in Hertz (1 Hz = 1 cycle/s).
- Clock period is the time between ticks of the clock and is measured in seconds per cycle.
- Period = 1/frequency
- Example: A 200 MHz (MegaHertz) clock has a clock period of 5 nanoseconds

CPU Performance and Its factors

- CPU time can be represented by this simple Equation

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

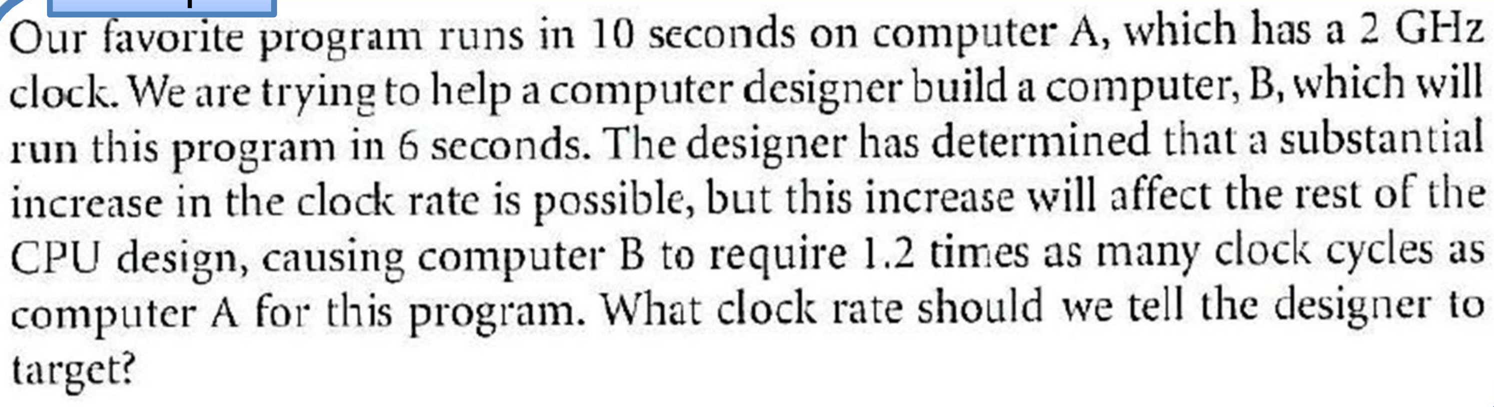
- If we are using clock rate then

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

- So, the goal of the designer is to minimize the number of clock cycles or the length of the clock cycle.
- There is a trade-off between the two. (Remember single cycle processor?)

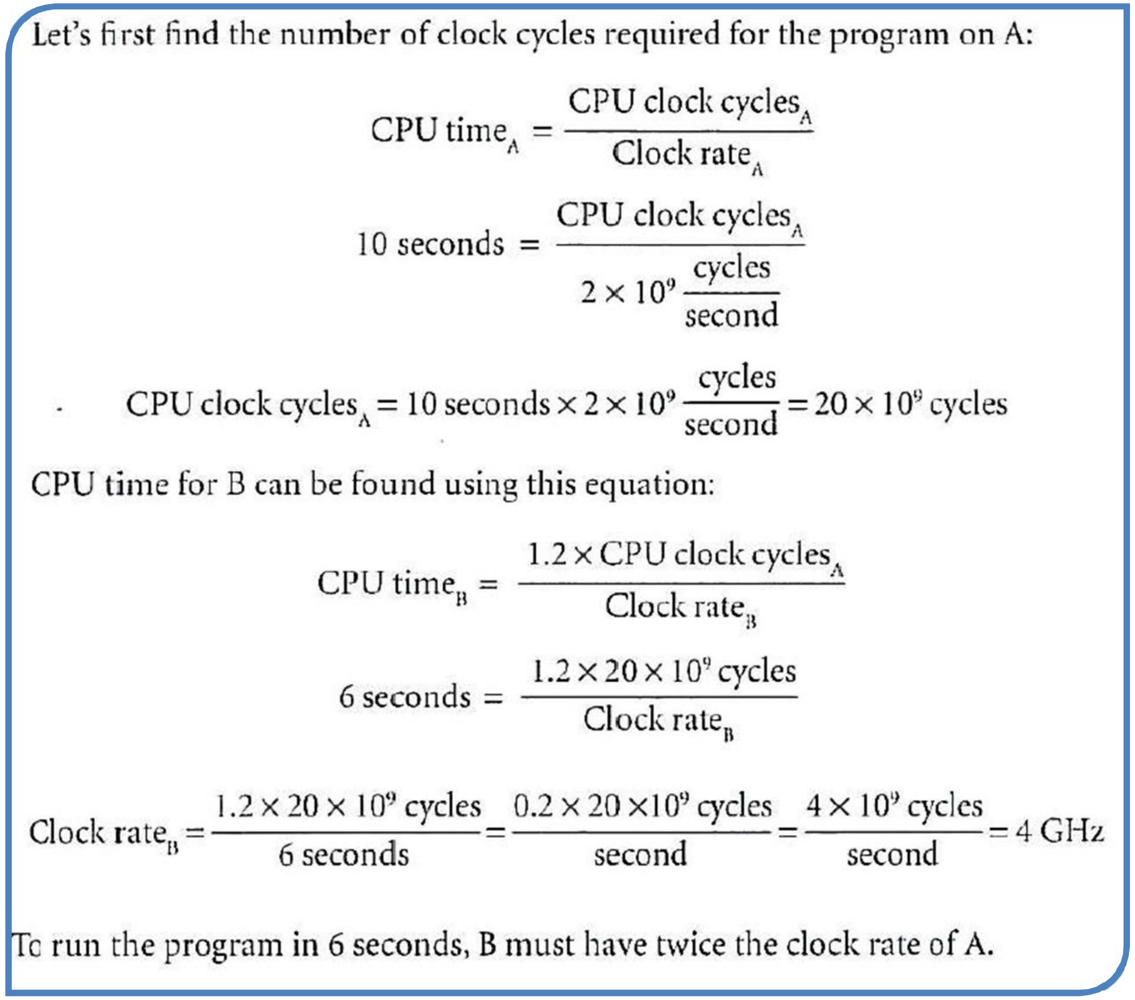
CPU Performance and Its factors

Example


Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

CPU Performance and Its factors

Example



Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

Instruction Performance

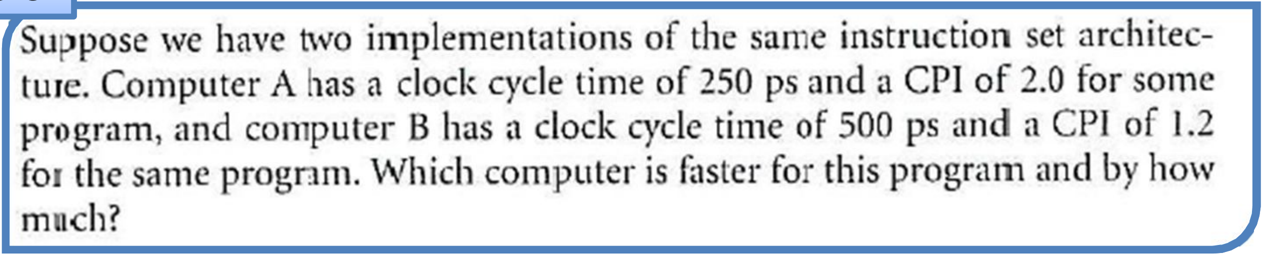
- Execution time and CPU time must depend on the number of instructions as well.

$$\boxed{\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{array}{c}\text{Average clock cycles}\\ \text{per instruction}\end{array}}$$

- Clock Cycles per instruction CPI, represents the average number of clock cycles per instruction for a program or program fragment.

Instruction Performance

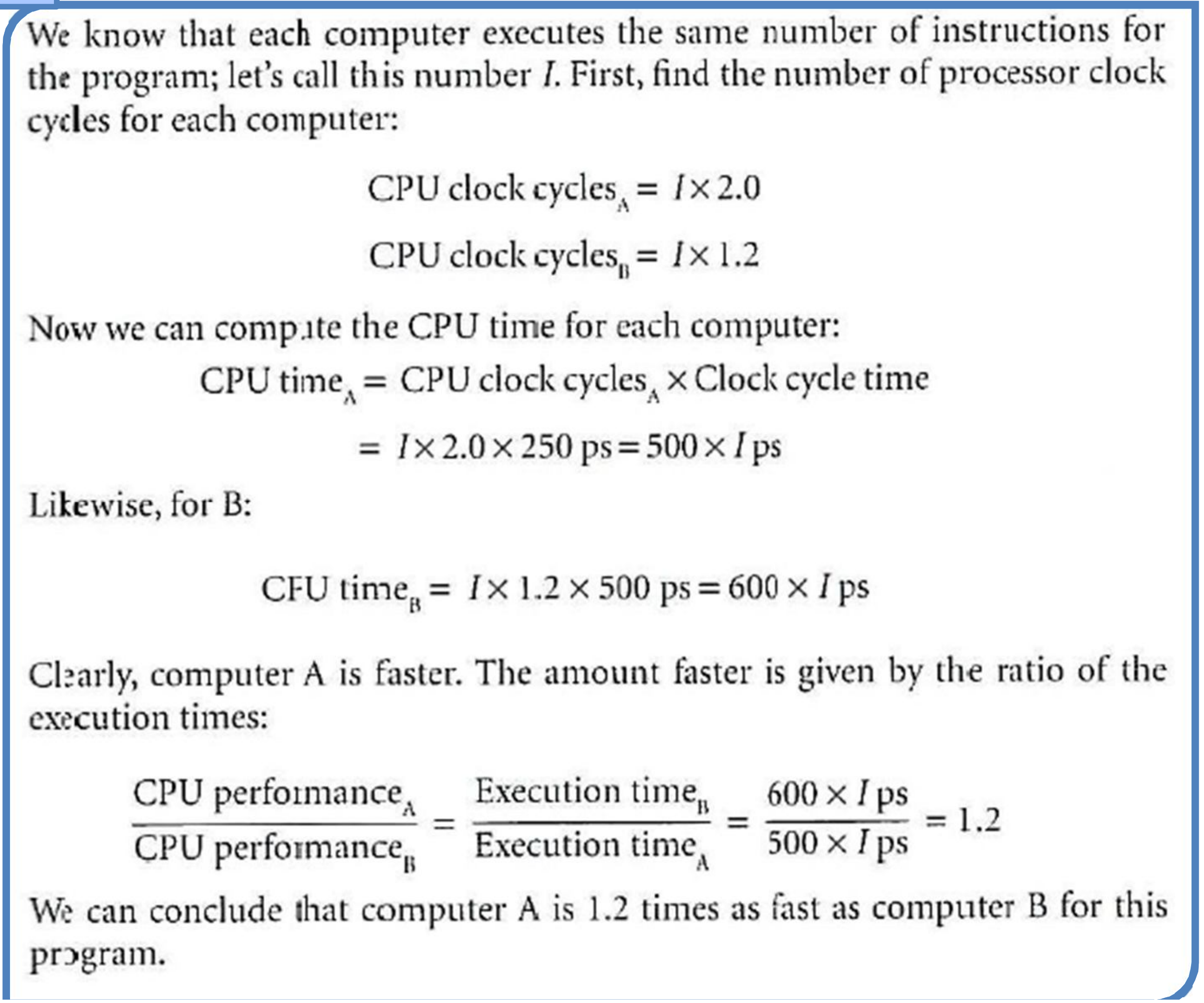
Example



Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

Instruction Performance

Example



We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}$$

$$= I \times 2.0 \times 250\text{ ps} = 500 \times I\text{ ps}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500\text{ ps} = 600 \times I\text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I\text{ ps}}{500 \times I\text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

The classic CPU Performance Equation

- The basic CPU equation can now be represented in terms of instruction count.

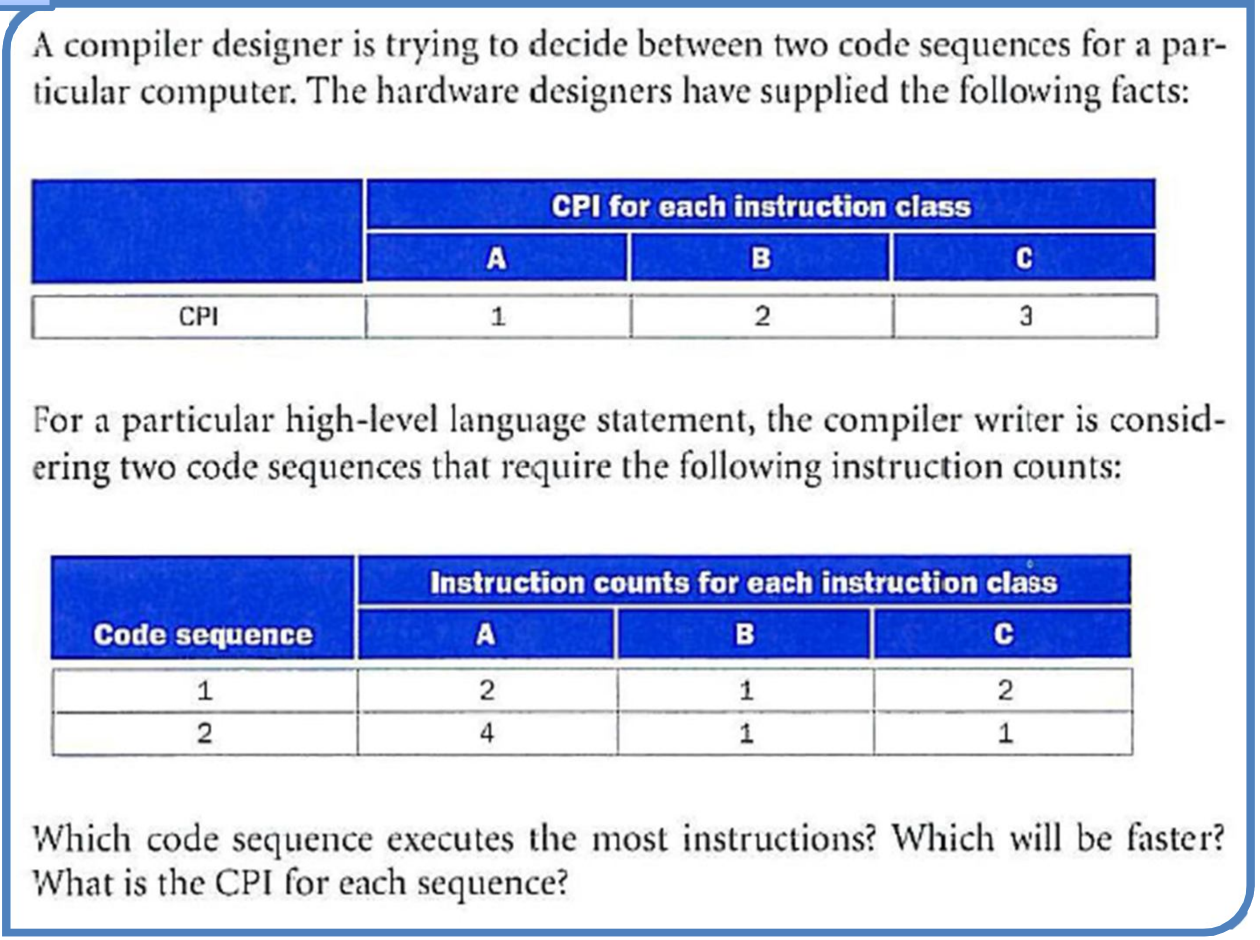
$$\boxed{\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}}$$

- In terms of Clock rate

$$\boxed{\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}}$$

Comparing Code Segments

Example



A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

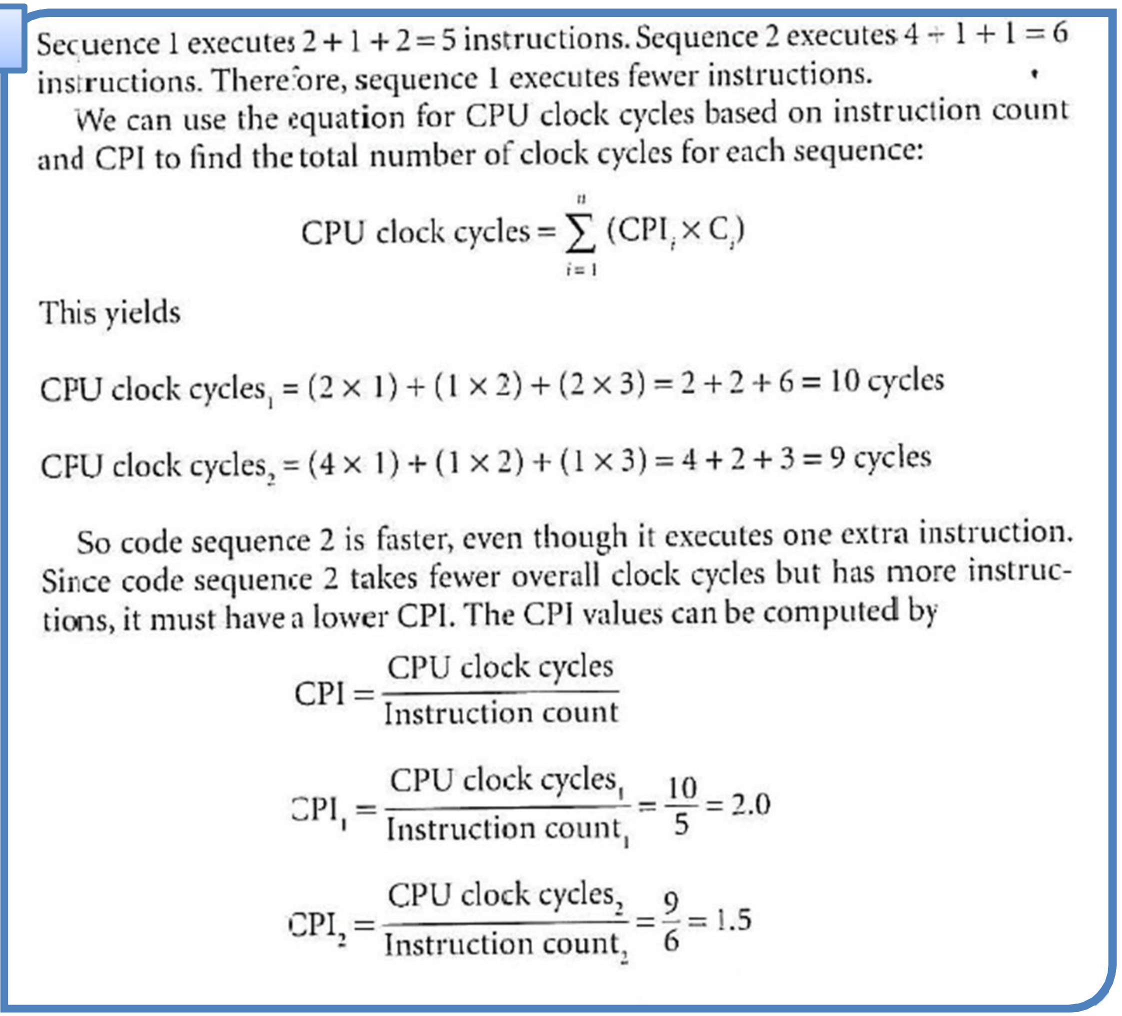
For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

	Instruction counts for each instruction class		
Code sequence	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

Comparing Code Segments

Example



Sequence 1 executes $2 + 1 + 2 = 5$ instructions. Sequence 2 executes $4 + 1 + 1 = 6$ instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

How to Determine these Values

- We can determine CPU execution time by running the program.
- Clock Cycle time is published as a part of the documentation of the microprocessor.
- We can find instruction count by using the software tools or by the simulator of the architectures.
- We can also use hardware counters which are found in microprocessors to conduct measurements.
- CPI varies even within the same architectures and same clock rates.