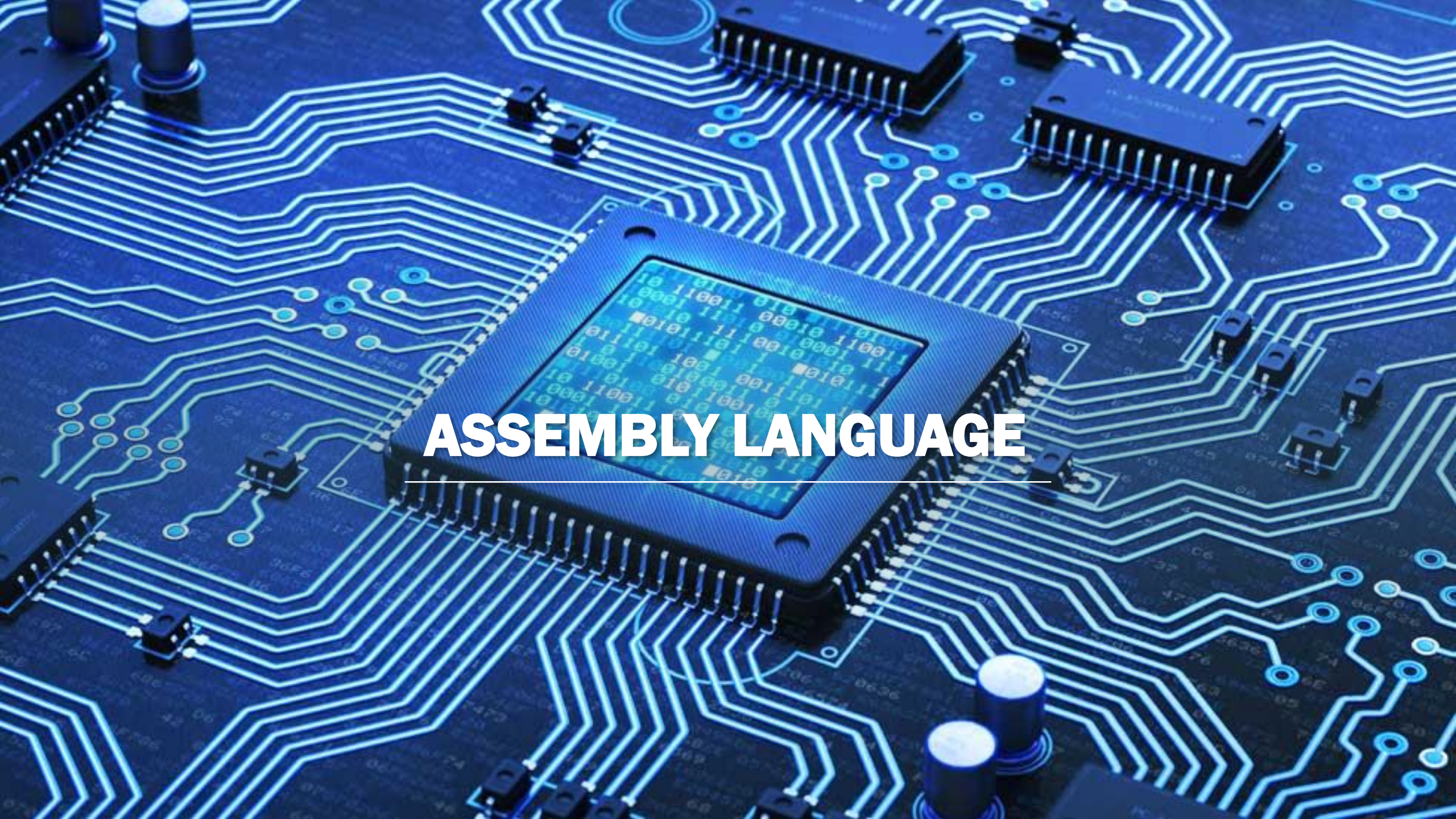

ASSEMBLY LANGUAGE

NUMBER REPRESENTATIONS

NUMBER REPRESENTATIONS

Consider an n-bit vector

$$B = b_{n-1} \ldots b_1 b_0$$

where $b_i = 0$ or 1 for $0 \leq i \leq n-1$. This vector can represent an unsigned integer value $V(B)$ in the range 0 to $2^n - 1$, where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

- We have three ways to represents numbers in computers –
  - Sign and magnitude representation
  - 1's complement representation
  - 2's complement representation
- In all systems, if left most bit is 0 for positive numbers and 1 for negative numbers.

NUMBER REPRESENTATIONS

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

SIGN-AND-MAGNITUDE REPRESENTATION

- In the ***sign and magnitude*** system, negative values are represented by changing the most significant bit from 0 to 1 in the bit vector of the corresponding positive value.
- For example, +5 is represented by 0101, and −5 is represented by 1101.
- What will be the representation of -19 in 8-bit sign-and-magnitude system?

1’S COMPLEMENT REPRESENTATION

- In the ***1’s complement*** system, negative values are represented are obtained by complementing each bit of the corresponding positive number.
- For example, the representation for −3 is obtained by complementing each bit in the vector of 3, 0011 to yield 1100.
- The same operation, bit complementing, is done to convert a negative number to the corresponding positive value. Converting either way is referred to as forming the 1’s-complement of a given number.
- For n-bit numbers, this operation is equivalent to subtracting the number from $2^n - 1$.
- What will be the representation of -19 in 8-bit 1’s complement system?

2’S COMPLEMENT REPRESENTATION

- In 2’s complement system, negative values are computing by first computing 1’s complement of a positive number then adding 1 into 1’s complement of number.
- The above operation is equivalent to subtracting number from 2^n.
- There are two representations of 0 in both sign-and-magnitude system and 1’s complement system but in 2’s complement system there is only one representation of 0.
- It’s the most common method used in modern computers.

NUMBER REPRESENTATIONS

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

ADDITION & SUBTRACTION IN 2'S COMPLEMENT SYSTEM

- To add two numbers, add their n-bit representations, ignoring the carry-out bit from the most significant bit (MSB) position. The sum will be the algebraically correct value in 2's-complement representation if the actual result is in the range $-2^n - 1$ through $+2^{n-1} - 1$.
- To subtract two numbers X and Y , that is, to perform X − Y , form the 2's-complement of Y , then add it to X using the add rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range $-2^n - 1$ through $+2^{n-1} - 1$.

ADDITION

(a)	0 0 1 0	(+2)	(b)	0 1 0 0	(+4)
	+ 0 0 1 1	(+3)		+ 1 0 1 0	(−6)
	0 1 0 1	(+5)		1 1 1 0	(−2)
(c)	1 0 1 1	(−5)	(d)	0 1 1 1	(+7)
	+ 1 1 1 0	(−2)		+ 1 1 0 1	(−3)
	1 0 0 1	(−7)		0 1 0 0	(+4)

SUBTRACTION

(e)	1101	(−3)	⇒	1101	
	− 1001	(−7)		+ 0111	
				0100	(+4)
(f)	0010	(+2)	⇒	0010	
	− 0100	(+4)		+ 1100	
				1110	(−2)
(g)	0110	(+6)	⇒	0110	
	− 0011	(+3)		+ 1101	
				0011	(+3)
(h)	1001	(−7)	⇒	1001	
	− 1011	(−5)		+ 0101	
				1110	(−2)
(i)	1001	(−7)	⇒	1001	
	− 0001	(+1)		+ 1111	
				1000	(−8)
(j)	0010	(+2)	⇒	0010	
	− 1101	(−3)		+ 0011	
				0101	(+5)

SIGN EXTENSION & OVERFLOW

- We often represent a small number using a large number of bits. There to represent a positive number, we simply add 0s to left side of bit vector.
- To represent negative numbers, we add 1s to the left side of bit vector. This is called the sign extension since we are expanding the sign bit.
- Using 2's-complement representation, n bits can represent values in the range -2^{n-1} to $+2^{n-1}-1$.
- For example, the range of numbers that can be represented by 4 bits is −8 through +7.
- When the actual result of an arithmetic operation is outside the representable range, an arithmetic overflow has occurred.

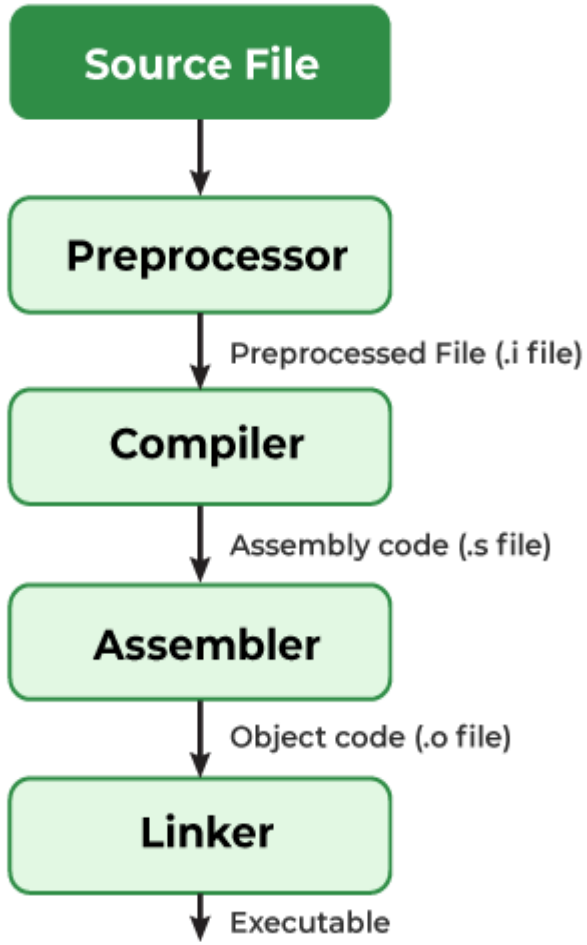
EXERCISES

- What is representation of -19 in 8-bit 2’s complement system?
- Compute the sum of 20 and -19 in 8-bit 2’s complement system.
- Compute the sum of 90 and 80 in 8-bit 2’s complement system?

INSTRUCTION SET ARCHITECTURE

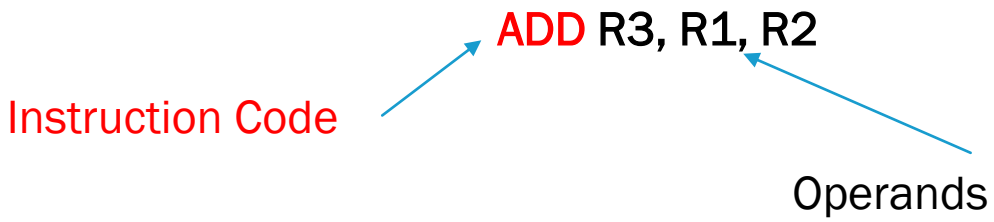
COMPILATION PROCESS

- Compilation is a process of converting high level language program into machine code.
- Overall compilation process of a program can be divided in multiple intermediate steps.
- You can use command **$gcc –Wall –save-temps filename.c –o filename** to see all the intermediate files generated by gcc compiler.



ASSEMBLY LANGUAGE

- A low level programming language uses simple statements that correspond to typically just one machine instruction. These languages are specific to the ISA.
- They have a generic structure that consists of a sequence of assembly statements.
- Typically, each assembly statement has two parts –
  - an instruction code that is a mnemonic for a basic machine instruction
  - a list of operands.



ASSEMBLY TO MACHINE CODE

SUB R1, R2, 3

Field	Encoding
sub	00001
I	1
R1	0001
R2	0010
3	11

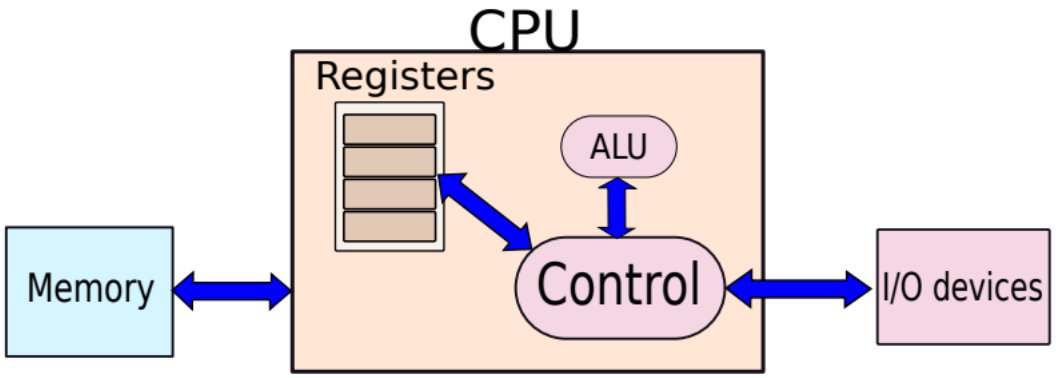
So now if represent above instruction in sequence of 1s and 0s, it will look like this –

00001 1 0001 0010 00 0000 0000 0000 0011

WHY DO WE NEED TO STUDY ASSEMBLY LANGUAGE?

- A compiler might not be using all the instructions provided ISA, so writing your code in assembly language directly might enable you to use all features of a ISA.
- Sometimes a compiler might not able to optimize your code as much as it can be because compiling algorithm might not find all hidden patterns in a code. So writing your code directly in assembly language might make your code more efficient.
- It's essential for hardware designers to study assembly language so that they can design a processor which can efficiently execute all programs of a ISA. Assembly language helps them in understanding instruction format instead of directly figuring out from sequence of bits.

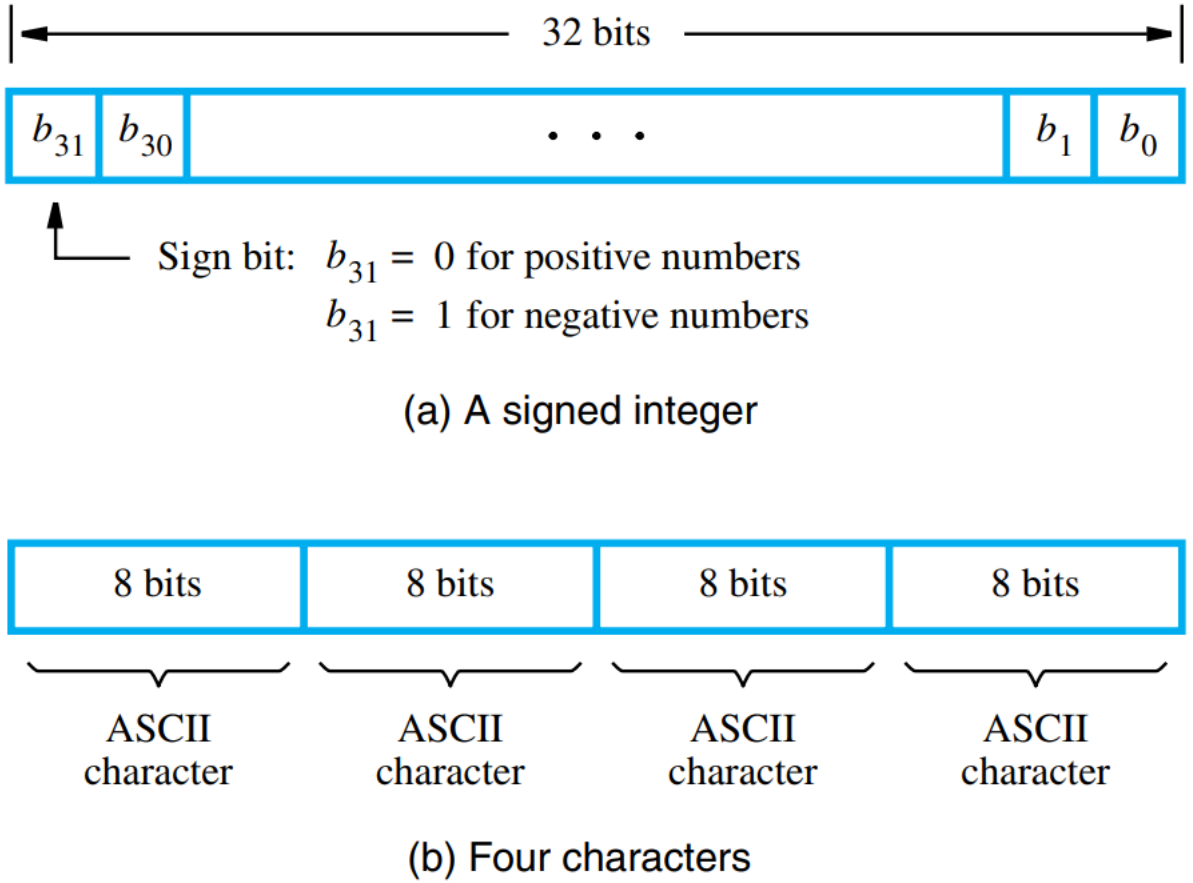
MACHINE MODEL



- The program is stored in a part of the main memory.
- The central processing unit (CPU) reads out the program instruction by instruction, and executes the instructions appropriately.
- The program counter keeps track of the memory address of the instruction that a CPU is executing.

REGISTERS

- Every machine has a set of registers that are visible to the assembly programmer. ARM has 16 registers, x86 (32-bit) has 8 registers, and x86 64 (64 bits) has 16 registers.
- ARM names them r0 . . . r15, and x86 names them eax, ebx, ecx, edx, esi, edi, ebp, and esp. A register can be accessed using its name.

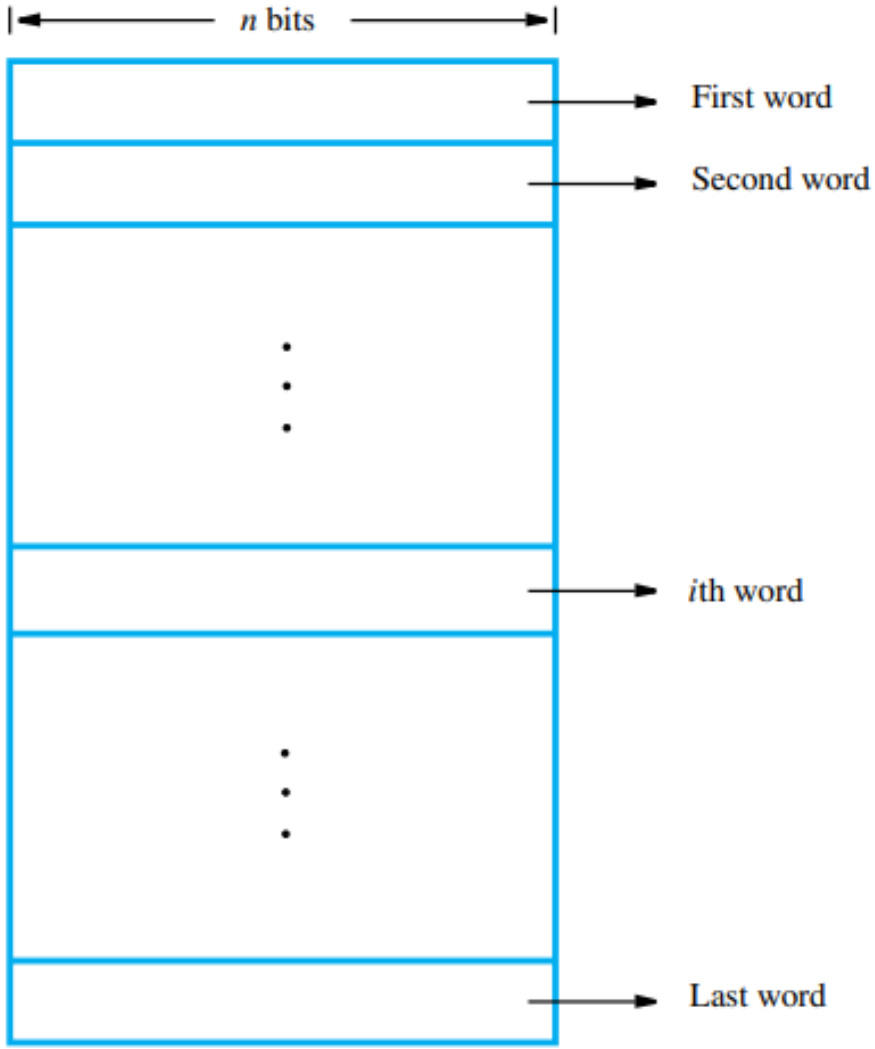


MEMORY

- Computer memory consists of millions of small storage cells. Each cell can store one bit of information having values 0 or 1.
- Memory can be think of as a large array of bytes where each byte has a unique address. First byte has address 0, second 1 and so on.
- The address is a 32-bit unsigned integer in 32-bit machines and it is a 64-bit unsigned integer in 64-bit machines.
- A 32-bit processor can have at max 4GB memory. Since it can only have 2^{32} $unique\ byte\ addresses$ starting from 0, 1, …, till $2^{32} - 1$.

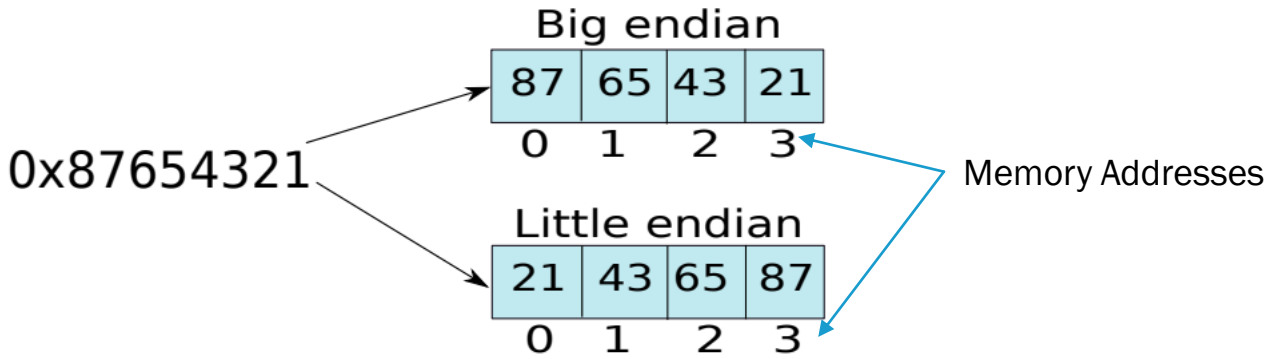
MEMORY ACCESS

- Memory is organized in such form that processor can access a group of **n** bits together. This group is called ***word*** and **n** is called ***word length***.
- Modern computers have word lengths that typically range from 16 to 64 bits.
- To access the memory, each location should have unique address. In modern computers, a unique address is assigned to each byte in memory. This is called **byte-addressable memory**.
- In memory, Byte locations have addresses 0, 1, 2,... so on. So if a machine uses 32-bit word size then word location addresses would be 0, 4, 8, ... so on.



BIG-ENDIAN AND LITTLE-ENDIAN REPRESENTATIONS

- Big-Endian and Little-Endian are two ways to store a **word** in memory.
- In Big-Endian representation, as the name suggest most/big significant byte is stored first i.e. at lower memory addresses and least significant bytes are stored at higher memory addresses.
- In Little-Endian representation, least significant byte is stored first at lower memory addresses and higher significant bytes are stored at higher addresses.
- For example, suppose we want to store a integer **0x87654321** in memory. We know that an integer is represented by 4 bytes so we can store it in two ways -



ARRAY REPRESENTATION

- An array is a linearly ordered set of objects, where an object can be a simple data type such as an integer or character, or can be a complex data type also.
- Suppose we want to store an integer array **int arr[4],** Since one integer takes 4 bytes, total bytes required to store **arr** would be 16 bytes.
- So if start storing array **arr** from memory location **loc** then **arr[0]** would be store on locations **loc+0, loc+1, loc+2, loc+3** and arr[1] will start from **loc+4** and so on.
- There are two representations for 2-D arrays –
  - **Row Major** – Store array row wise i.e. store first row, then second row and so on.
  - **Column Major** – Store array column wise i.e. store first column, then second column and so on.

EXERCISE

- How would you store -320 in 32-bit 2's complement system using both little-endian and big-endian representations?
- Represent array int arr[3] = [50, -60, 30] in 32-bit 2's complement system using both little-endian and big-endian representations?
- Compute the sum of 90 and 80 in 8-bit 2's complement system?

INSTRUCTION FORMAT

- An assembly language statement is made of instruction and list of operands.
- The instruction is textual identifier of a actual machine instruction.
- The list of operands contains either value or location of each operand.
- Value of a operand is a numeric constant which also known as a ***immediate.***
- The operand locations can be either register locations or memory locations.

INSTRUCTION FORMAT

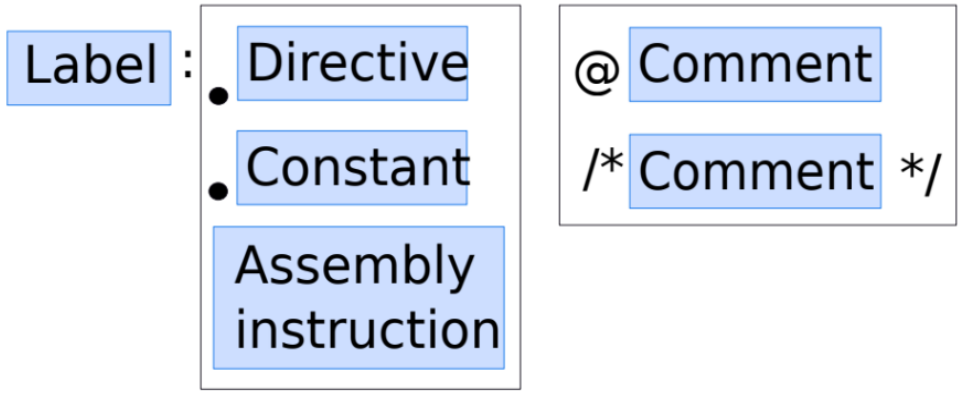
add r3, r1, r2

- The format of add instruction is as follows - $< instruction >< destination\ register >< operand\ register\ 1 >< operand\ register\ 2 >$
- The step by step execution of add instruction is as follows –
  - First read the value of register r1, Let's say the value is v1
  - Read the value of register r2, Let's say the value is v2
  - Compute sum of both values v3 = v1 + v2
  - Store the sum into register r3

EXERCISE

- Write execution steps for following instructions –
  - sub r3, r1, r2
  - mul r3, r1, 3

GENERAL STRUCTURE OF ASSEMBLY STATEMENT



- A label in an assembly file uniquely identifies a given point or data item in the assembly program. Which is helpful while defining a loop or if-else conditions.
- Directives are instructions used by the assembler to help automate the assembly process and to improve program readability.
- Directive - tells the assembler to do something like declare a function
- Constant - declares a constant

MEMORY OPERATIONS

- Both program instructions and data is stored in the memory.
- To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor.
- The Read operation transfers a copy of the contents of a specific memory location to the processor.
- To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.
- The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location.
- To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

REGISTER TRANSFER NOTATION

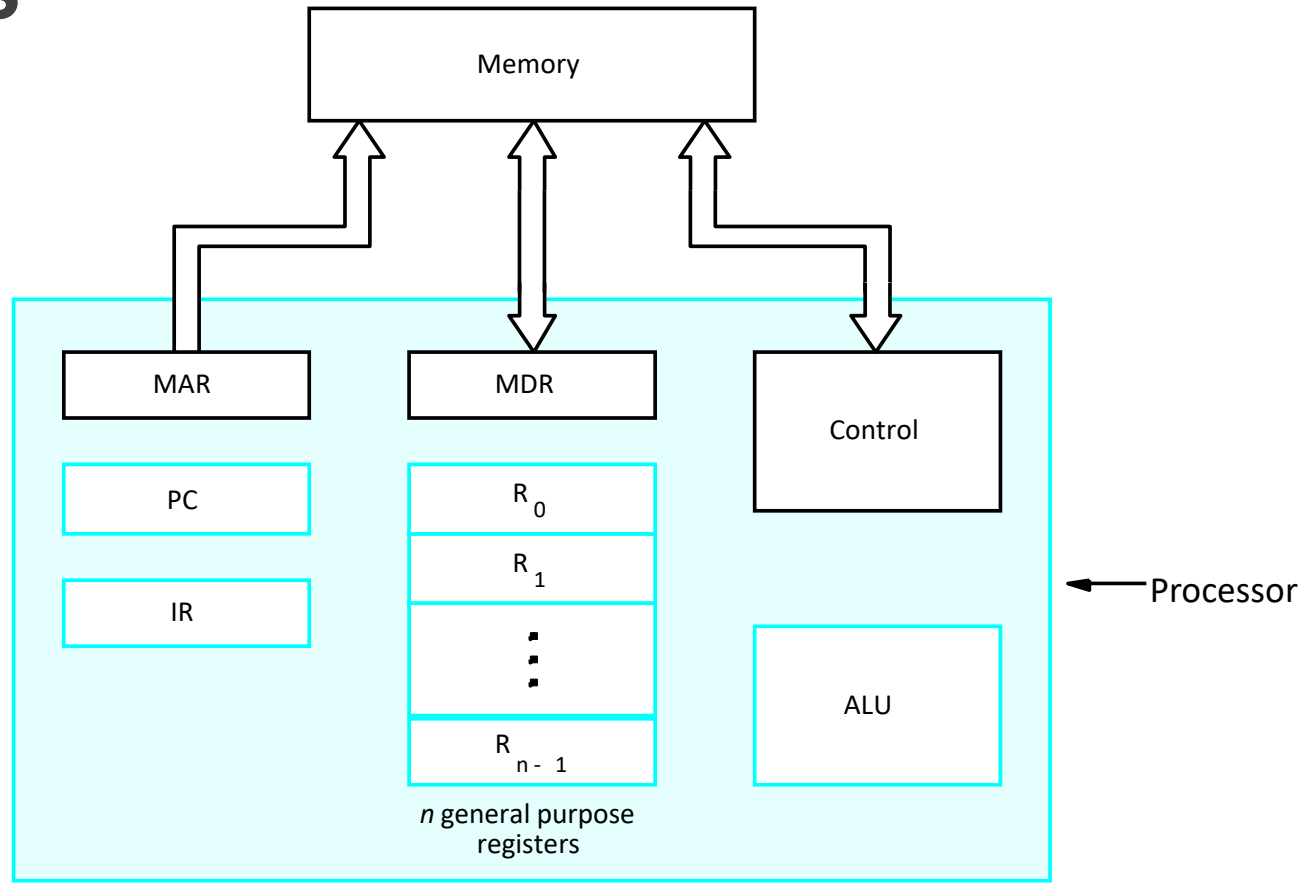
- We need some way to describe how information or data is transferring from one place to another.
- There are two notations –
  - Register Transfer Notation
  - Assembly Language Notation
- **r1 ← r2** represents that contents of register r2 is copied in register r1.
- r1 ← r2 + r3 represents that sum of contents of r2 and r3 is stored in r1.
- r1 ← [LOC] represents that contents of memory location **LOC** has been copied into r1.
- What does r1 ← [r2] + [r3] represents ?

ASSEMBLY LANGUAGE NOTATION

- In Assembly Language Notation, an assembly statement consists of instruction and list of operands.
- **Load r3, LOC** represents that content of memory location **LOC** has been copied into register r3.
- **Store r2, LOC** represents that content of register r2 has been copied in the memory at location **LOC.**
- In assembly language notation, Instruction or operation to be performed is represented using ***mnemonics*** which are usually short forms for operation.
- These mnemonics are specifics to an assembly language. Which means ARM architecture might use different set of mnemonics compared to Intel architecture.

BASIC OPERATIONAL CONCEPTS

- To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.



REGISTERS

- Instruction register (IR)
- Program counter (PC)
- General-purpose register (R_0 – R_{n-1})
- Memory address register (MAR)
- Memory data register (MDR)

TYPICAL OPERATING STEPS

- Programs reside in the memory through input devices
- PC is set to point to the first instruction
- The contents of PC are transferred to MAR
- A Read signal is sent to the memory
- The first instruction is read out and loaded into MDR
- The contents of MDR are transferred to IR
- Decode and execute the instruction

TYPICAL OPERATING STEPS

- Get operands for ALU
  - General-purpose register
  - Memory (address to MAR – Read – MDR to ALU)
- Perform operation in ALU
- Store the result back
  - To general-purpose register
  - To memory (address to MAR, result to MDR – Write)
- During the execution, PC is incremented to the next instruction

INTERRUPTS

- Normal execution of programs may be preempted if some device requires urgent servicing.
- The normal execution of the current program must be interrupted – the device raises an *interrupt* signal.
- Whenever an interrupt is raised by a device, operating system executes *interrupt service routine.*
- Current system information backup and restore (PC, general-purpose registers, control information, specific information)

INTRODUCTION TO RISC ISA

key characteristics of RISC ISA –

- Each instruction must fit in a single word i.e. instruction should be represented using n-bits where n is size of word. For a 32-bit processor, it's usually 32-bit instruction.
- Memory is accessed using Load / Store instructions.
- All operands involved in arithmetic or logical instructions must be either registers or immediate values.

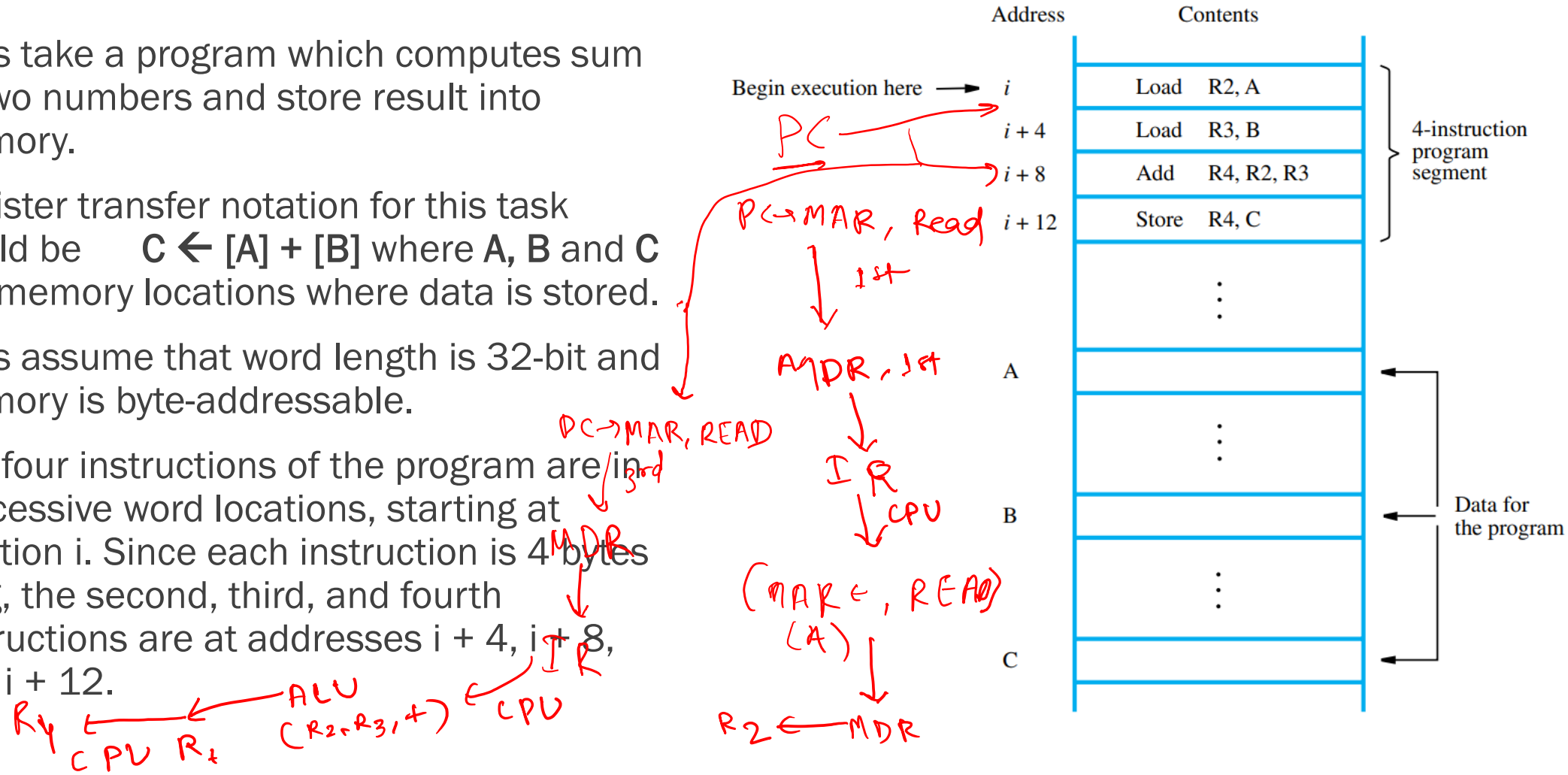
EXERCISE

- Write down a simple assembly program for computing sum of two numbers.

```
LOAD R1, LOCA @ Reads A from memory location LOCA and store it into register R1
LOAD R2, LOCB @ Reads B from memory location LOCB and store it into register R2

ADD R3, R1, R2 @ Compute the sum of contents of R1 and R2 and store it into register R3

STORE R3, LOCS @ Store the content of R3 into memory at location LOCS
```

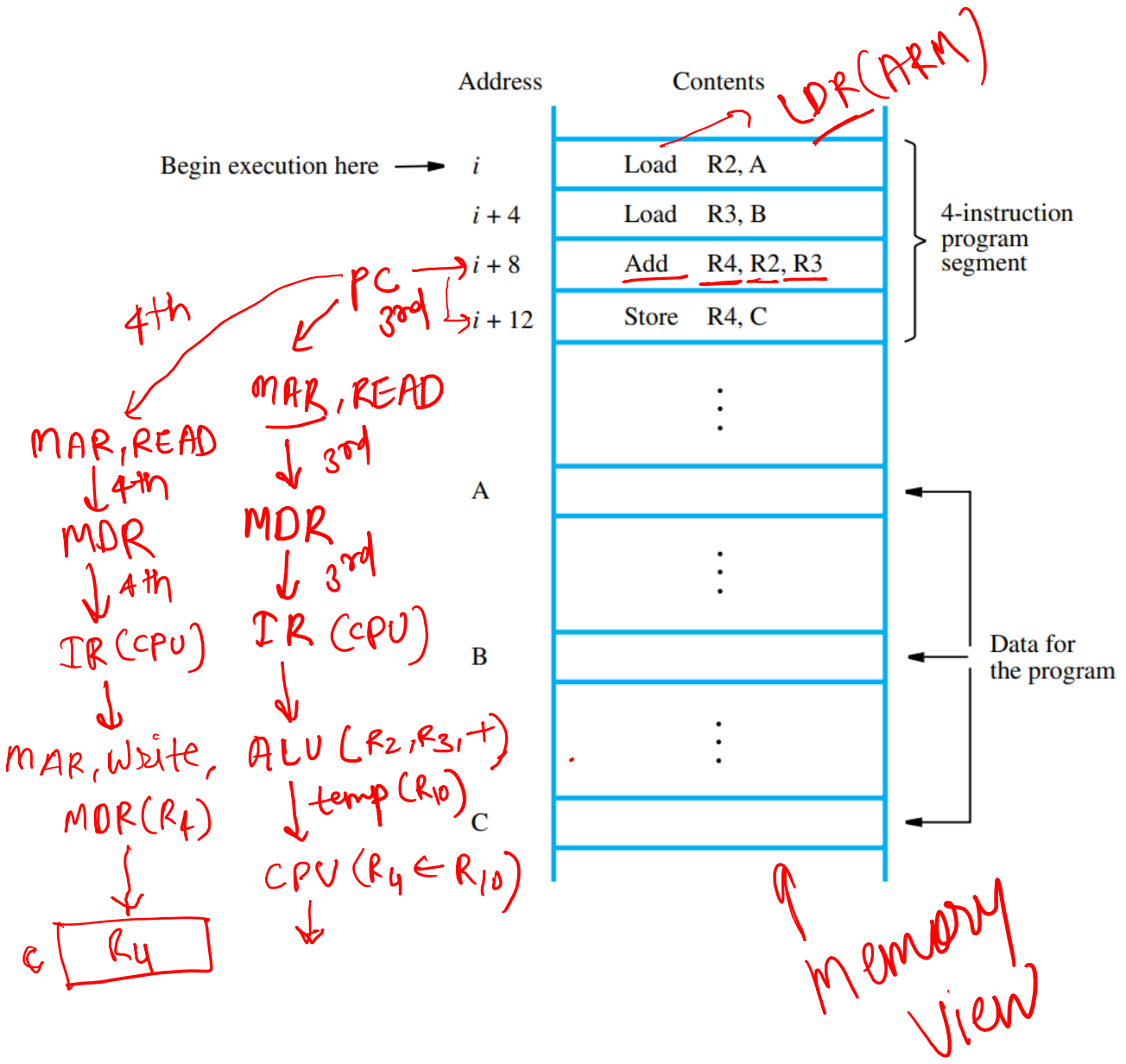
INSTRUCTION EXECUTION

- Let's take a program which computes sum of two numbers and store result into memory.
- Register transfer notation for this task would be **C ← [A] + [B]** where **A, B** and **C** are memory locations where data is stored.
- Let's assume that word length is 32-bit and memory is byte-addressable.
- The four instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses i + 4, i + 8, and i + 12.



INSTRUCTION EXECUTION

- First **PC** points to the memory location **i** where first instruction **LOAD R2, A** is stored, this instruction is then copied into **IR** and CPU starts executing this instruction. And **PC** is incremented to point next instruction.
- While executing first instruction, CPU sends read signal to memory to read data at location A. So address A will be stored in memory register **MAR** and data will be stored in the memory register **MDR** and then content of MDR will be copied into general purpose register **R2.**

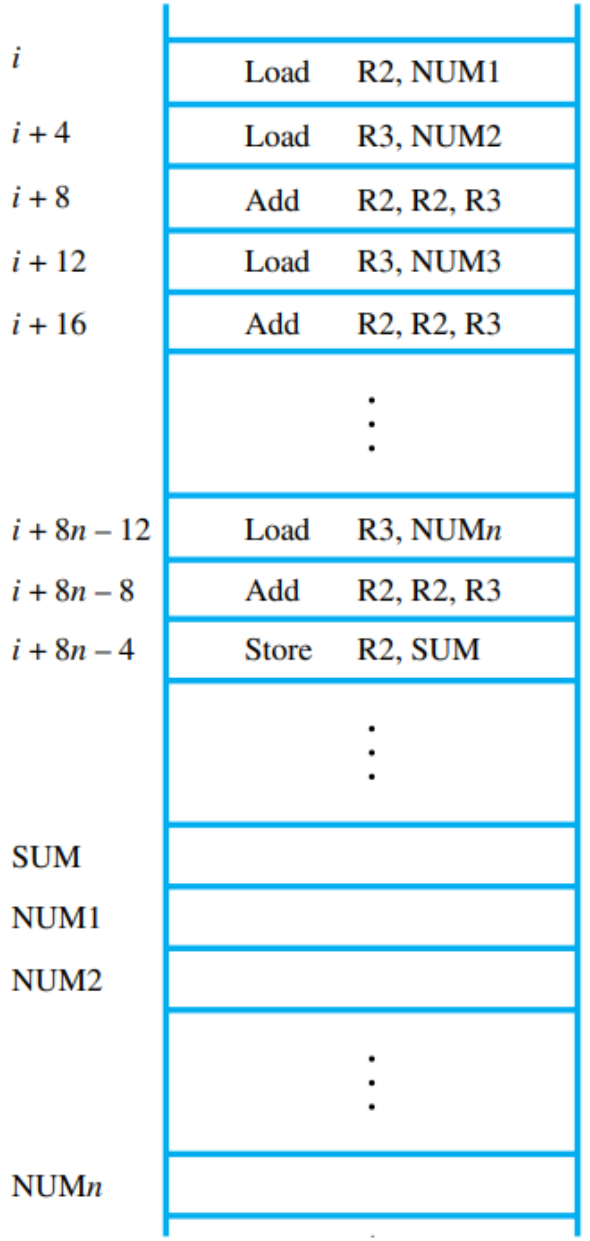


EXERCISE

- Write an Assembly Program to compute sum of 3 variables A, B, C and store sum in the variable **SUM.** Draw the memory view of program also.

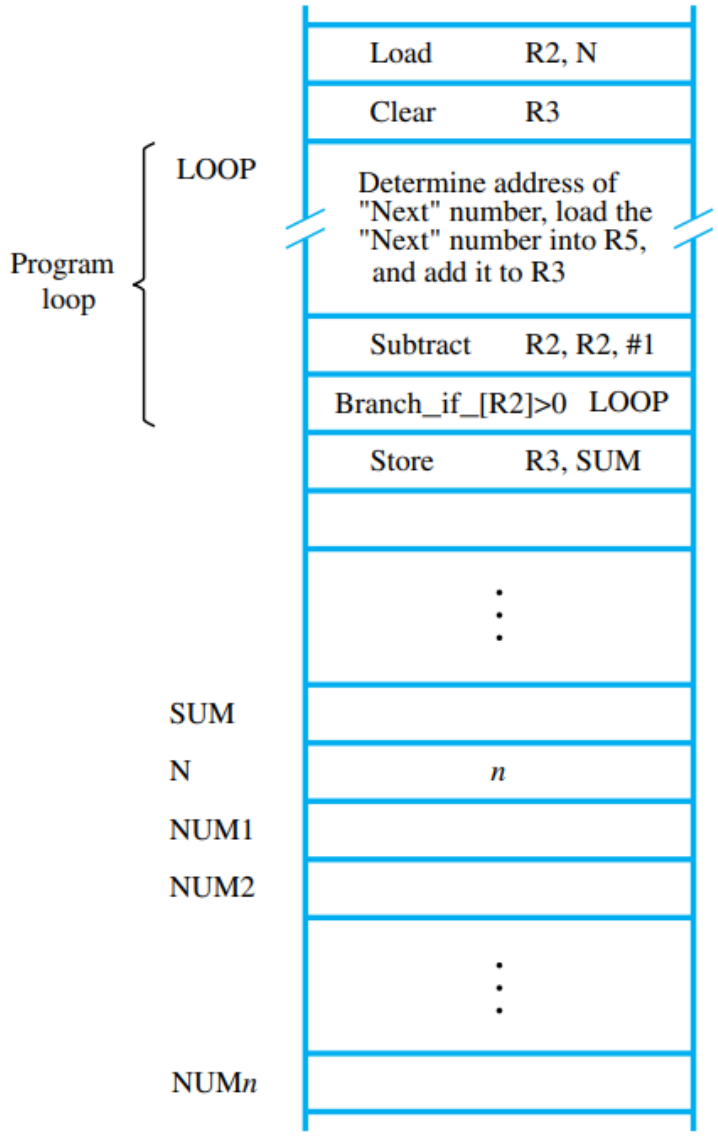
BRANCHING

- So far we have studied how sequence of instructions get executed one by one. But real-world program are rarely step by step. Sometimes a program needs to execute different section in different order then they are defined or execute same part again and again.
- Let's take an example of adding N numbers and storing their sum. One way to do this is repeat sequence of Load and Add instructions N times.
- This is very boring and tedious thing to do if N is very large.



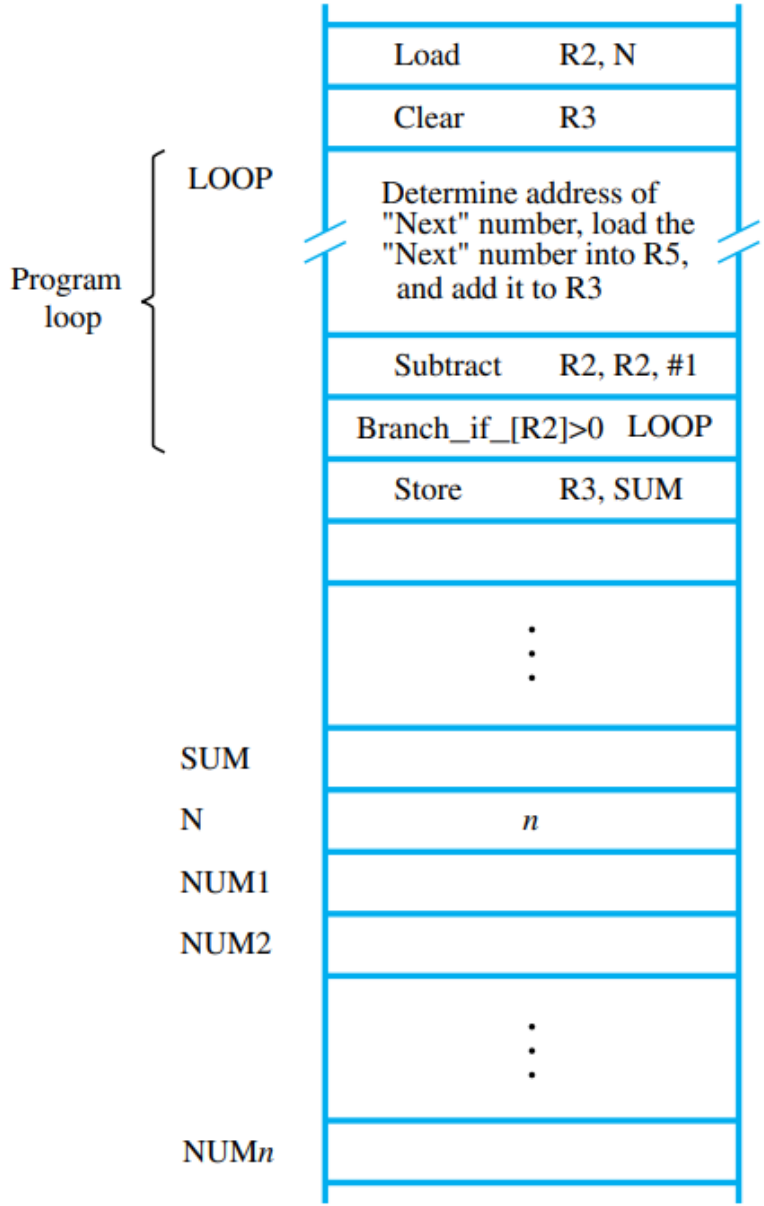
BRANCHING

- How can we simulate looping mechanism of high-level programming language in assembly language?
- Use a register to maintain a counter and use Branch Instruction to re-execute another instruction by changing PC value.



BRANCHING

- **Branch** instruction is used to change value of PC.
- **Branch_if <condition>** instruction only changes value of PC when the condition is satisfied.



EXERCISE

- Write an assembly program to compute sum of numbers from 1 to 10.

```
@ Program to compute sum of first 10 numbers.
LOAD R1, #1 @ Load 1 in register R1 to maintain a counter to keep track of loop
LOAD R2, #0 @ Load 0 in register R2, we will store sum in register R2

loop:
    ADD R2, R2, R1 @ Updating sum by adding counter value in it
    ADD R1, R1, #1 @ Updating counter by 1
    BRANCH_IF_[R1 <= 10] loop @ Check if counter is less than equal to 10. If yes then start the loop again
```

EXERCISE

- Convert following program into assembly code -

```
if(A > B){
    C = A - B;
}
else {
    C = B - A;
}
```

```
@ Assembly program to simulate if-else condition
LOAD R1, [A] @ Load variable A from memory into register R1
LOAD R2, [B] @ Load variable B from memory into register R2

BRANCH_IF_[R1 >= R2] if @ Branch to statements with label if, if condition is satisfied
BRANCH_IF_[R1 < R2] else @ Branch to statements with label else, if condition is satisfied

if:
    SUBTRACT R3, R1, R2 @ Subtract R2 from R1 and store the result in R3
    BRANCH exit @ Branch to statements with label exit

else:
    SUBTRACT R3, R2, R1 @ SUBTRACT R1 from R2 and store the result in R3
    BRANCH exit @ Branch to statements with label exit

exit:
    STORE R3, [Result]
```

CONDITIONAL FLAGS

- The processor keeps track of information about the results of various operations for the use by next conditional branching instruction.
- Processor has four conditional flags or codes **N, Z, V** and **C** to keep track of information. These flags are stored in a special register called **Current Program Status Register (CPSR).**
- Each instruction can set appropriate conditional flag based on the result –
  - N (**N**egative) – Set to 1 if the result is negative; otherwise, cleared to 0
  - Z (**Z**ero) – Set to 1 if the result is 0; otherwise, cleared to 0
  - V (o**V**erflow) – Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
  - C (**C**arry) – Set to 1 if a carry-out results from the operation; otherwise cleared to 0

CONDITIONAL CODES: N & Z FLAGS

- The N and Z flags record whether the result of an arithmetic or logic operation is negative or zero.
- In some processors, they may also be affected by the value of the operand of a Move instruction. This makes it possible for a later conditional branch instruction to cause a branch based on the sign and value of the operand that was moved.
- Some processors also provide a special Test instruction that examines a value in a register or in the memory without modifying it and sets or clears the N and Z flags accordingly.

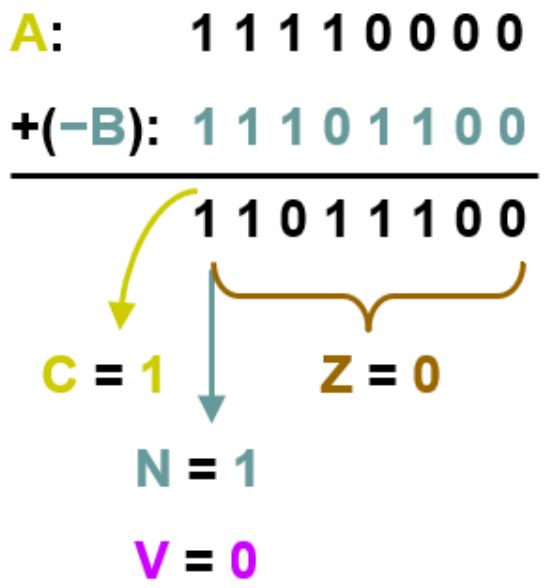
CONDITIONAL CODES: V FLAG

- The V flag indicates whether overflow has taken place.
- Overflow occurs when the result of an arithmetic operation is outside the range of values that can be represented by the number of bits available for the operands.
- The processor sets the V flag to allow the programmer to test whether overflow has occurred and branch to an appropriate routine that deals with the problem.
- Instructions such as **Branch_if_overflow** are usually provided for this purpose.

CONDITIONAL CODES: C FLAG

- The C flag is set to 1 if a carry occurs from the most-significant bit position during an arithmetic operation.
- This flag makes it possible to perform arithmetic operations on operands that are longer than the word length of the processor.

A: 1 1 1 1 0 0 0 0
B: 0 0 0 1 0 1 0 0



TYPES OF INSTRUCTIONS

TYPES OF INSTRUCTIONS

Instructions can be classified into many types based on following two criteria –

- Classification by Functionality of an Instruction
- Classification by Number of operands in an Instruction

BASED ON FUNCTIONALITY OF INSTRUCTION

Based on functionality, Instructions can be divided into mostly five types –

- **Data Transfer Instructions** – Instructions required to transfer values from one location to another location. Examples – Load, Store, Move etc.
- **Data Processing Instructions** - Data processing instructions are typically arithmetic instructions such as add, subtract, and multiply, or logical instructions that compute bitwise or, and exclusive or. Comparison instructions also belong to this family.
- **Branch Instructions** - Branch instructions help the processor's control unit to jump to different parts of the program based on the values of operands. They are useful in implementing for loops and if-then-else statements.
- **I/O Transfer Instructions** – Instructions used for transferring data between processor and I/O devices.
- **Exception Generating Instructions** - These specialized instructions help transfer control from a user level program to the operating system.

BASED ON NUMBER OF OPERANDS

- We know all instructions follow same format $< instruction >< operand\ 1 >< operand\ 2 > \cdots < operand\ N >$.
- We can classify instructions based on the number of operands that they require. If an instruction requires n operands (including source and destination), then we say that it is a **n-address format** instruction.

0-ADDRESS INSTRUCTION FORMAT

- These instructions do not specify any operands or addresses.
- They operate on data stored in registers or memory locations implicitly defined by the instruction.
- For example, a zero-address instruction might simply add the contents of two registers together without specifying the register names.

1-ADDRESS INSTRUCTION FORMAT

- These instructions specify one operand or address, which typically refers to a memory location or register.
- The instruction operates on the contents of that operand, and the result may be stored in the same or a different location.
- For example, a one-address instruction might load the contents of a memory location into a register.

2-ADDRESS INSTRUCTION FORMAT

- These instructions specify two operands or addresses, which may be memory locations or registers.
- The instruction operates on the contents of both operands, and the result may be stored in the same or a different location.
- For example, a two-address instruction might add the contents of two registers together and store the result in one of the registers.

3-ADDRESS INSTRUCTION FORMAT

- These instructions specify three operands or addresses, which may be memory locations or registers.
- The instruction operates on the contents of all three operands, and the result may be stored in the same or a different location.
- For example, a three-address instruction might multiply the contents of two registers together and add the contents of a third register, storing the result in a fourth register.

CPU ORGANIZATION

Based on number of address fields, CPU can be organized in many types. For example –

- Single Accumulator Organization
- Stack Organization
- General Purpose Register Organization

SINGLE ACCUMULATOR ORGANIZATION

- In this CPU Organization, the first ALU operand is always stored into the Accumulator and the second operand is present either in Registers or in the Memory.
- Accumulator is the default address thus after data manipulation the results are stored into the accumulator.
- One address instruction is used in this type of organization.
- For Single Accumulator Based CPU design, Instruction format would be as following –

$$< instruction\ or\ opcode >< operand1 >$$

- This type of CPU organization is first used in **PDP-8 processors** and is used for process control and laboratory applications. It has been totally replaced by the introduction of the new general register-based CPU.

SINGLE ACCUMULATOR ORGANIZATION

- Mainly two types of operation are performed in a single accumulator CPU organization –
  - **Data Transfer Operation** – In this type of operation, the data is transferred from a source to a destination.
    - **LOAD X** – This operation will load data in accumulator from memory location X.
    - **STORE Y** – This operation will store accumulator data at memory location Y.
  - **ALU Operation** – In this type of operation, arithmetic operations are performed on the data.
    - **MULTPLY X** – This operation will multiply data at memory location [X] with data in accumulator and will store result back into accumulator. **AC ← AC * M[X]**
    - **ADD X** – Will be equivalent to **AC ← AC + M[X]**

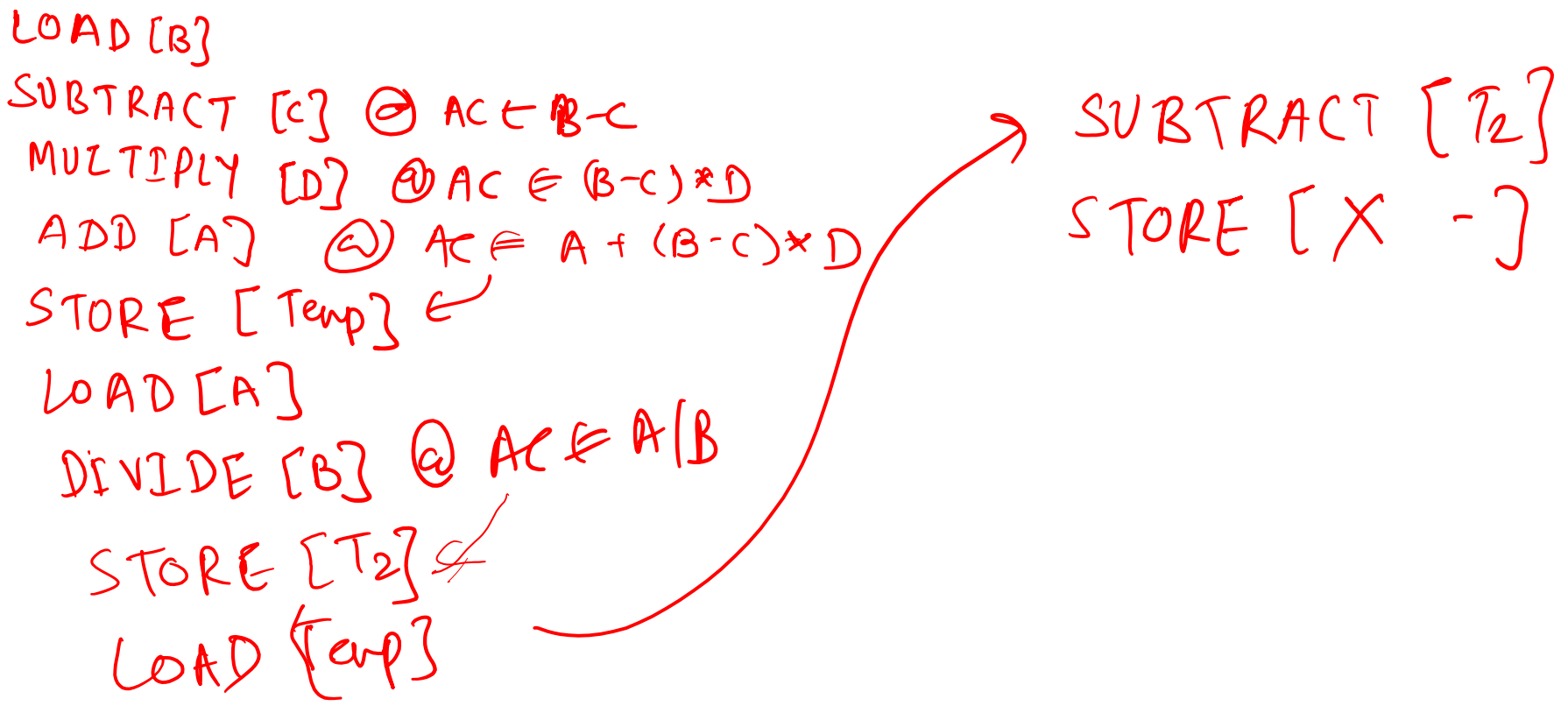
EXERCISE

- Evaluate the expression (A + B) * (C + D) using a single accumulator CPU organization.

```
@ Program to evaluate (A+B)*(C+D) using a single accumulator CPU Organization
LOAD [A] @ AC <- [A]
ADD [B] @ AC <- AC + [B], AC now contains sum A + B
STORE [T] @ [T] <- AC
LOAD [C] @ AC <- [C]
ADD [D] @ AC <- AC + [D], AC now contains sum C + D
MULTIPLY [T] @ AC <- AC * [T], AC now contains product (A+B)*(C+D)
STORE [RESULT] @ [RESULT] <- AC
```

EXERCISE

- Evaluate the expression **X = A + (B-C) * D - A/B** using a single accumulator CPU Organization.



SINGLE ACCUMULATOR ORGANIZATION

- **Advantages -**
  - One of the operands is always held by the accumulator register. This results in short instructions and less memory space.
  - Execution of single instruction takes less time, since you need to fetch less data from memory.
- **Disadvantages -**
  - When complex expressions are computed, program size increases due to the usage of many short instructions to execute it. There more memory is needed to store the program.
  - As the number of instructions increases for a program, the execution time increases.

STACK ORGANIZATION

- The computers which use Stack-based CPU Organization are based on a data structure called a **stack**.
- The stack is a list of data words. It uses the **Last In First Out (LIFO)** access method which is the most popular access method in most of the CPU.
- A register is used to store the address of the topmost element of the stack which is known as **Stack pointer (SP)**.
- In this organization, ALU operations are performed on stack data. It means both the operands are always required on the stack. After manipulation, the result is placed in the stack.
- **PDP-11, Intel's 8085, and HP 3000** are some examples of stack-organized computers.

STACK ORGANIZATION

- Data transfer operations which can be performed in stack organization are following –
  - **PUSH X** – This instruction will first increment the SP and then store data of memory location [X] in the stack. SP will point to this new data at the top of stack i.e. **SP ← SP + 1; SP ← [X]**
  - **POP Y** – This instruction will transfer data on the top of stack, to the memory location [Y] and then will decrement SP by 1. i.e. **M[Y] ← SP; SP ← SP – 1**
- Data processing operations can be also be performed in stack organization. These instruction will be 0-address instructions. They directly operates on the contents of stack and results get pushed back into the stack.
  - **SUBTRACT** – This instruction pops the two top data from the stack, subtracting the data, and pushing the result into the stack at the top.

EXERCISE

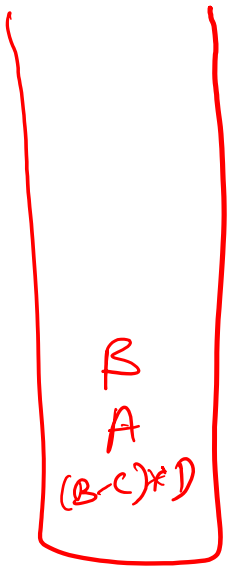
- Evaluate the expression (A + B) * (C + D) using a Stack based CPU organization.

```
@ Program to evaluate (A+B)*(C+D) using a stack based CPU Organization
PUSH [A] @ Pushes data [A] in the stack; SP <- [A]
PUSH [B] @ Pushes data [B] in the stack; SP <- [B]
ADD @ Pops top two entries from stack and pushes back sum of those entries; SP <- [A] + [B]
PUSH [C] @ Pushes data [C] in the stack; SP <- [C]
PUSH [D] @ Pushes data [D] in the stack; SP <- [D]
ADD @ Pops top two entries from stack and pushes back sum of those entries; SP <- [C] + [D]
MULTIPLY @ Pops top two entries from stack and pushes back multiplication of those entries; SP <- (A+B)*(C+D)
POP [RESULT] @ Pops the top entry from stack and stores it at memory location [RESULT]; [RESULT] <- SP
```

EXERCISE

- Evaluate the expression **X = A + (B-C) * D – A/B** using a stack based CPU organization.

PUSH [B]
PUSH [C]
SUBTRACT
PUSH [D]
MULTIPLY
PUSH [A]
PUSH [B]
DIVIDE



STACK BASED CPU ORGANIZATION

- Advantages -
  - Efficient computation of complex arithmetic expressions.
  - Execution of instructions is fast because operand data are stored in consecutive memory locations.
  - The length of instruction is short as they do not have an address field.
- **Disadvantages –**
  - The size of the program increases.

GENERAL REGISTER BASED CPU ORGANIZATION

- When we are using multiple general-purpose registers, instead of a single accumulator register, in the CPU Organization then this type of organization is known as General register-based CPU Organization.
- If many CPU registers are available for heavily used variables and intermediate results, we can avoid memory references much of the time, thus vastly increasing program execution speed, and reducing program size.

EXERCISE

- Evaluate the expression **Y = (A + B) * (C + D)** using a General Purpose Registers CPU Organization.
- Evaluate the expression **X = A + (B-C) * D - A/B** using a General Purpose Registers CPU Organization.

GENERAL PURPOSE REGISTER CPU ORGANIZATION

- Advantages –
  - The efficiency of the CPU increases as large number of registers are used in this organization.
  - Less memory space is used to store the program since the instructions are written in a compact way.
- Disadvantages –
  - Care should be taken to avoid unnecessary usage of registers. Thus, compilers need to be more intelligent in this aspect.
  - Since a large number of registers are used, thus extra cost is required in this organization.

EXERCISE

- Write assembly program to solve expression $Z = (A - B + C) * (D / E)$ using –
  - Single Accumulator Based CPU Organization
  - Stack Based CPU Organization
  - General Purpose Based CPU Organization

ADDRESSING MODES

- So far we have studied how to write a simple assembly program.
- Each program needs to perform some operations on data; the data can be presented in either memory or registers.
- In an assembly statement, we define different types of operands based on where the data is present and how to access it.
- These different ways of specifying operands in an assembly statement are called ***addressing modes.***

ADDRESSING MODES

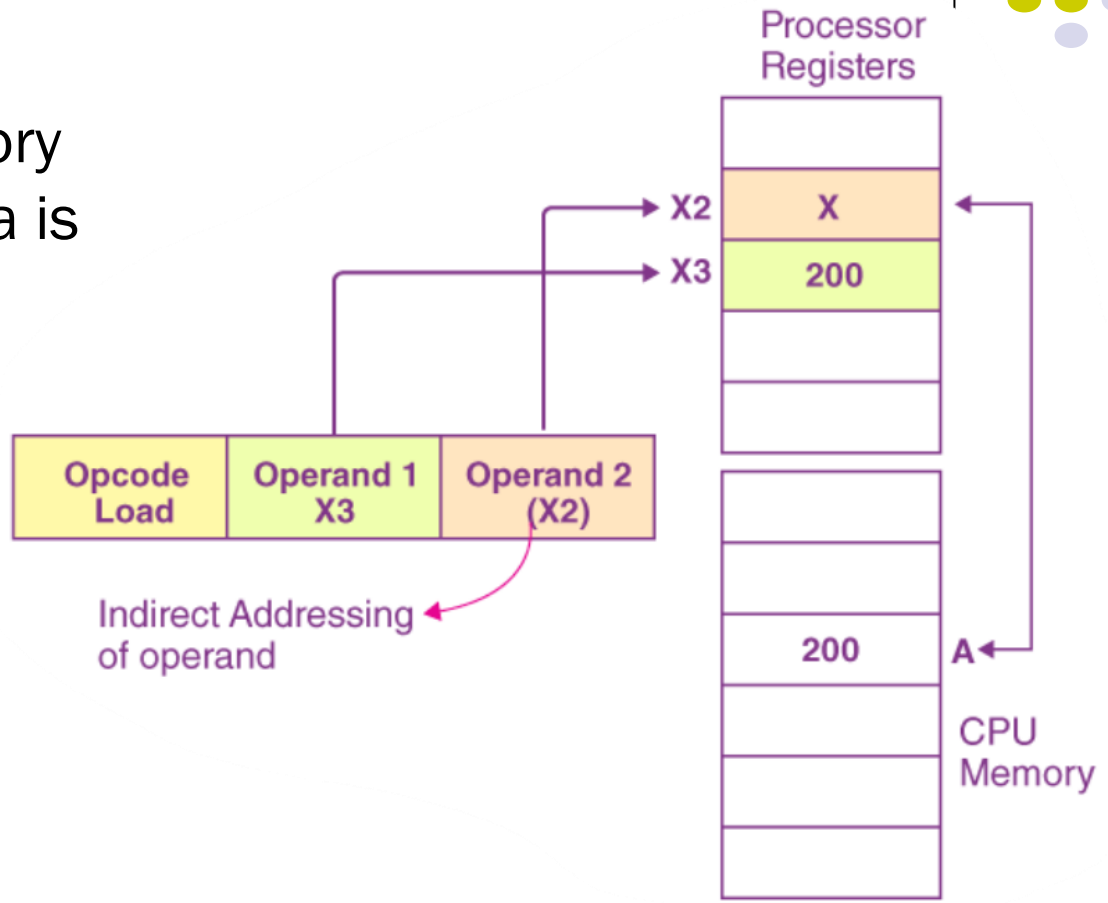
- A variable is a placeholder which can store some data temporarily.
- In assembly programming, variable is either stored at some memory location or stored in a register.
- **Register Mode** – The operand is the contents of a processor register; the name of the register is given in the instruction.
- **Absolute Mode or Direct Mode** – The operand is in a memory location; the address of this location is given explicitly in the instruction.
- **Immediate Mode** – The operand is a constant value directly given in instruction.

ADDRESSING MODES

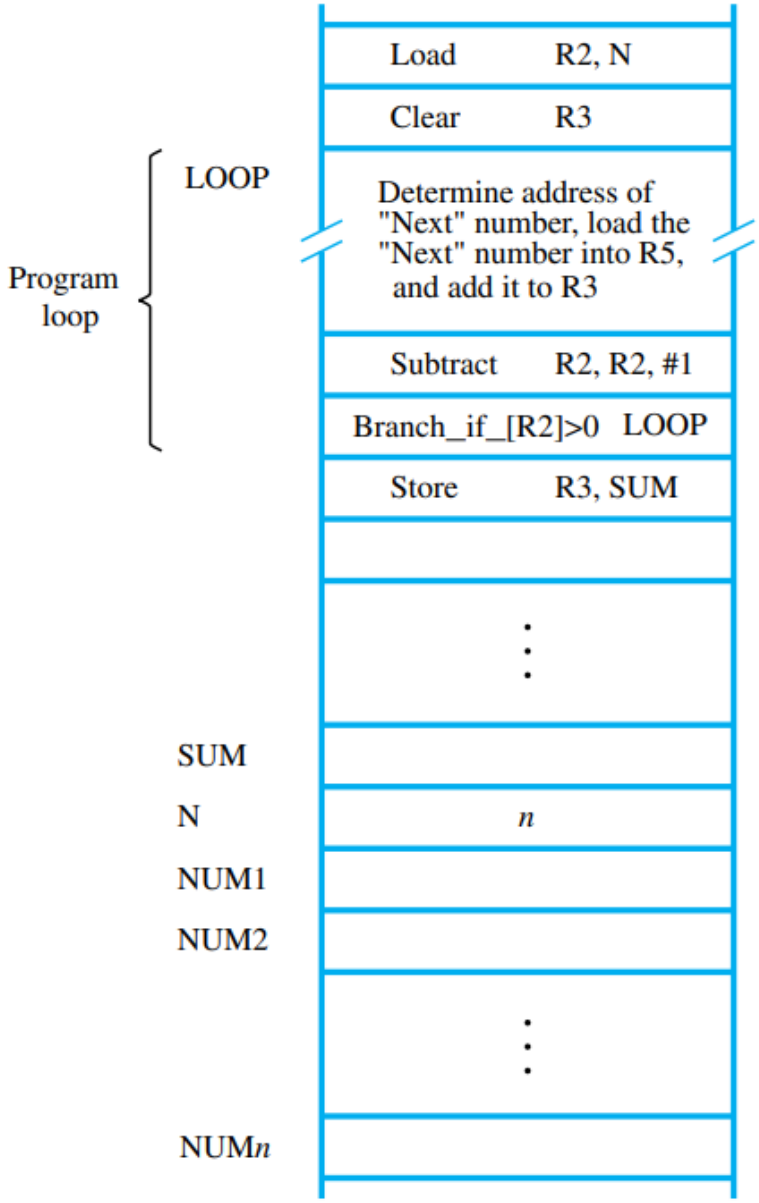
- **Implied**
  - **AC** is implied in “ADD M[AR]” in “**One-Address**” instruction or Single accumulator based CPU Organization.
  - **SP** is implied in “ADD” in “**Zero-Address**” instruction or Stack based CPU Organization.
- **Immediate**
  - Constant or Immediate value is encoded in the instruction. For example the use of a constant in “**MOVE R1, #5**”, i.e. R1 ← 5. In assembly language, to identify an immediate value, usually # symbol is written before immediate value.
- **Register**
  - Indicate which register holds the operand. For example **ADD R1, R2, R3** here all the operands are stored in registers.

Addressing Modes

- **Register Indirect**
  - Register stores the address of memory location where actual operand / data is stored.
    - **MOVE R1, (R2)**



ADDRESSING MODE: REGISTER INDIRECT



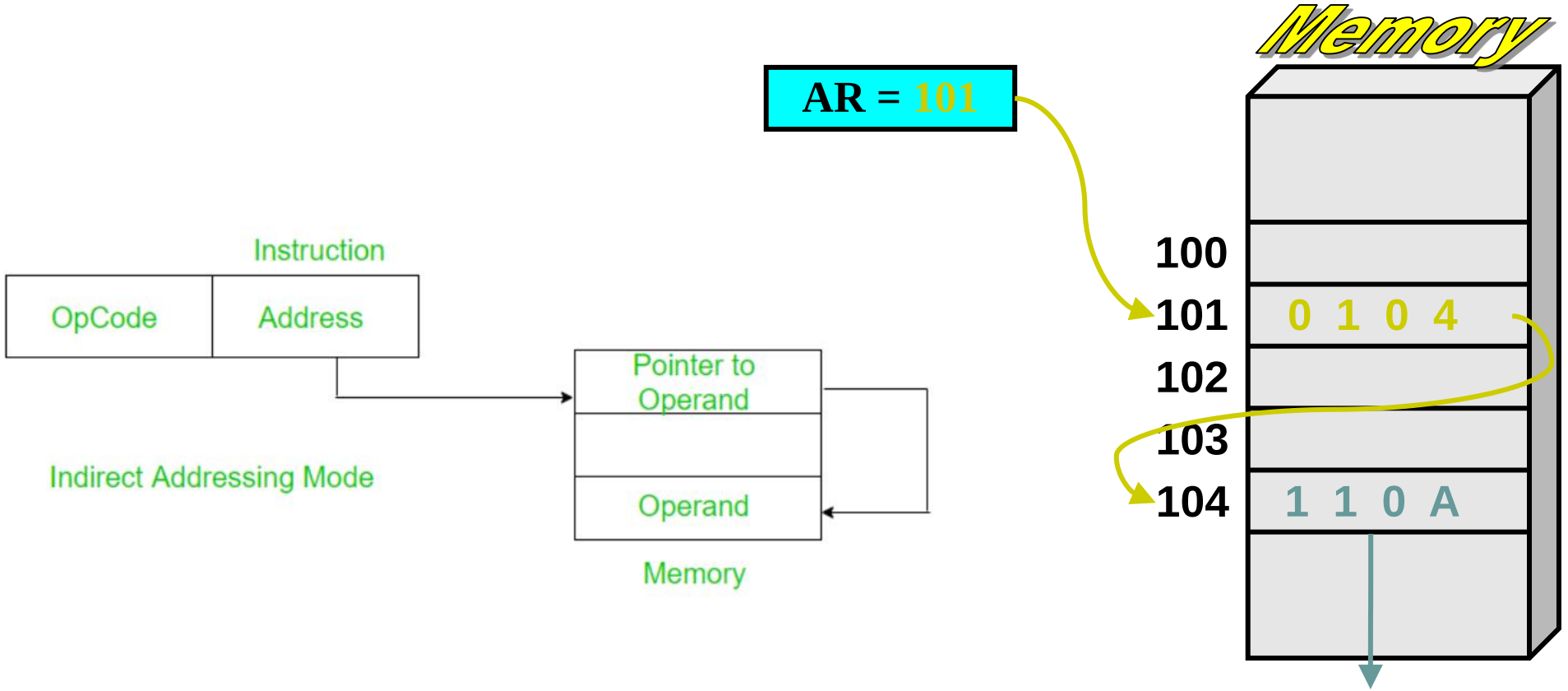
EXERCISE

- Compute sum of N numbers which are stored in array **nums[N].**

```
@ Program to compute sum of N numbers which are stored in a array
LOAD R1, =NUMS @ Stores address of first element of array in R1
LOAD R2, #0 @ Initialize R2 with 0 to store sum
LOAD R3, [N] @ Load N in R3

loop:
    ADD R2, R2, (R1) @ Adds content of R2 with content at memory location [R1] and stores the sum in R2
    ADD R1, R1, #4 @ Update address
    SUBTRACT R3, R3, #1 @ Update counter
    BRANCH_IF_[R3 > 0] loop
```

Addressing Modes

- **Memory Indirect Address**
  - Indicate the memory location that holds the address of the memory location that holds the data.



EXERCISE: POINTERS

- Write an assembly program to represent pointers like **A = *B.**

```
Load    R2, B
Load    R3, (R2)
Store   R3, A
```

ADDRESSING MODES: INDEX MODE

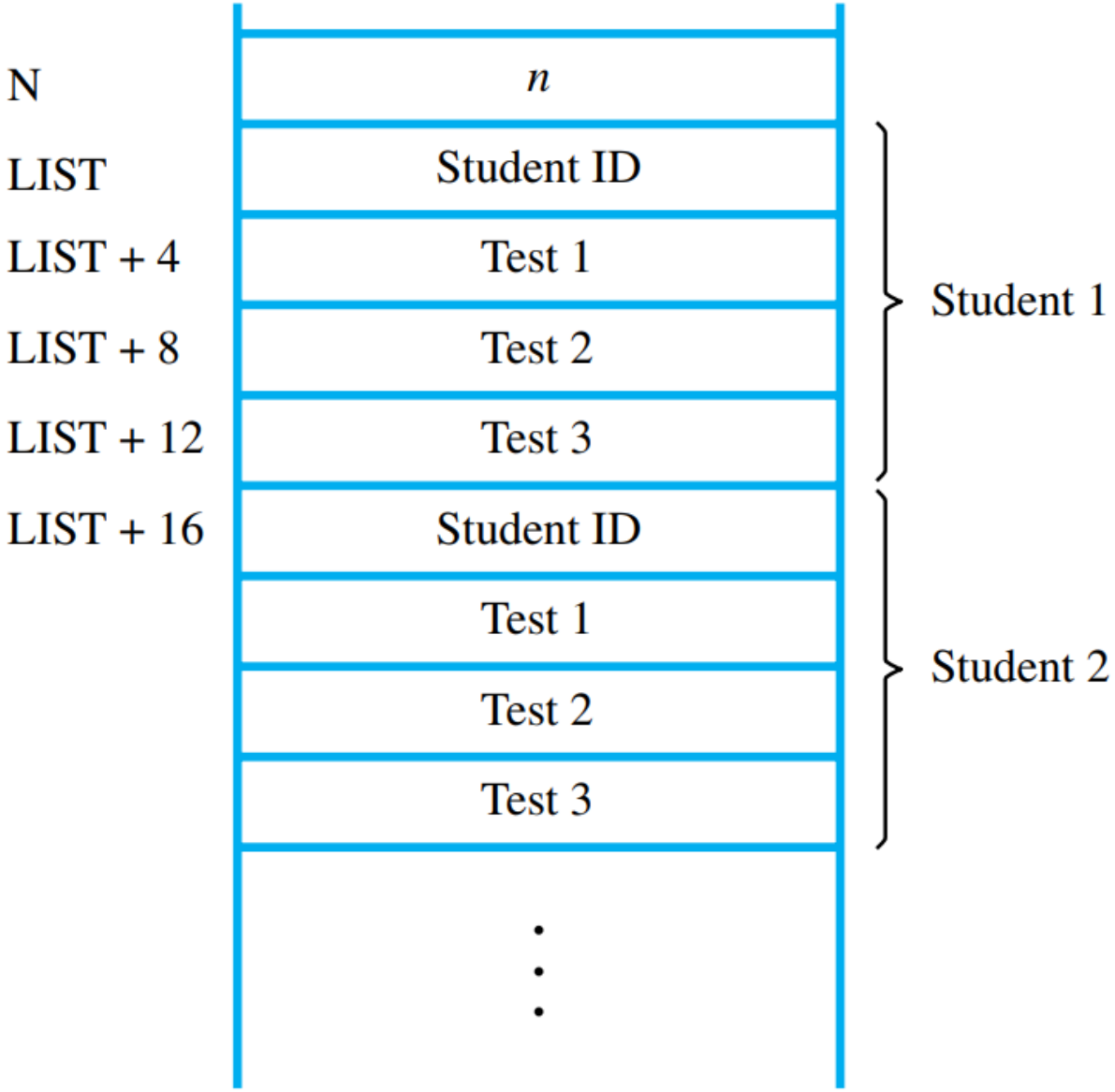
- **Index Mode** – Memory address of operand is generated by adding a constant value to the contents of a register.
- For example, **LOAD R1, X(Rx)** will copy data at memory location **X + [Rx]** into register R1. Here register Rx is called *index register.*
- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.

ADDRESSING MODES: INDEX MODE

- Index mode can be written in other forms also like –
- **(Ri, Rj)** represents that **Effective Address (EA)** of operand is **[Ri] + [Rj].** Here Ri is called **index register** and Rj is called **base register.**
- **X(Ri, Rj)** represents **EA = X + [Ri] + [Rj].**

EXERCISE

- Suppose you have students' data which is stored in 2-D array. Each row in matrix represents data of one student. Each row has 4 values, student ids, marks of subject1, marks of subject2, marks of subject3. Now we want to compute average of total marks of students. Write an assembly program for this.



ADDRESSING MODES: SUMMARY

Addressing Mode	Syntax	Effective Address (EA)
Immediate	#Value	Operand = Value
Register	Ri	Operand = Ri
Absolute / Direct	LOC	EA = LOC
Register Indirect	(Ri)	EA = [Ri]
Memory Indirect	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with Index	(Ri, Rj)	EA = [Ri] + [Rj]
Base with index and offset	X(Ri, Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Auto Increment	(Ri)+	EA = [Ri] then Increment Ri
Auto Decrement	-(Ri)	Decrement Ri then EA = [Rj]

Assembly Language

- Mnemonics (LD/ADD instead of Load/Add) used when programming specific computers
- The mnemonics represent the OP codes
- Assembly language is the set of mnemonics and rules for using them to write programs
- The rules constitute the language syntax

Assembler Directives

- Other information also needed to translate source program to object program
- How should symbolic names be interpreted?
- Where should instructions/data be placed?
- Assembler directives provide this information
- ORIGIN defines instruction/data start position
- RESERVE and DATAWORD define data storage
- EQU associates a name with a constant value

Assembler Directives

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Program Assembly & Execution

- From source program, assembler generates machine-language object program
- Assembler uses ORIGIN and other directives to determine address locations for code/data
- For branches, assembler computes ±offset from present address (in PC) to branch target
- Loader places object program in memory
- Debugger can be used to trace execution

Number Notation

- Decimal numbers used as immediate values:

  ADDI R2, R3, 93

- Assembler translates to binary representation
- Programmer may also specify binary numbers:

  ADDI R2, R3, %01011101

- Hexadecimal specification is also possible:

  ADDI R2, R3, 0x5D

- Note that $93 = 1011101_2 = 5D_{16}$

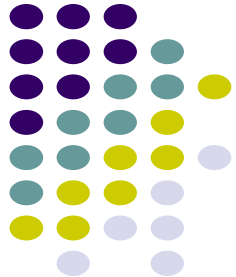
Types of Instructions

- Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data value is not modified

Data Transfer Instructions



Mode	Assembly	Register Transfer
Direct address	LD ADR	*AC* ← *M*[*ADR*]
Indirect address	LD @ADR	*AC* ← *M*[*M*[*ADR*]]
Relative address	LD $ADR	*AC* ← *M*[*PC*+*ADR*]
Immediate operand	LD #NBR	*AC* ← NBR
Index addressing	LD ADR(X)	*AC* ← *M*[*ADR*+*XR*]
Register	LD R1	*AC* ← *R1*
Register indirect	LD (R1)	*AC* ← *M*[*R1*]
Autoincrement	LD (R1)+	*AC* ← *M*[*R1*], *R1* ← *R1*+1

Data Manipulation Instructions

- Arithmetic
- Logical & Bit Manipulation
- Shift

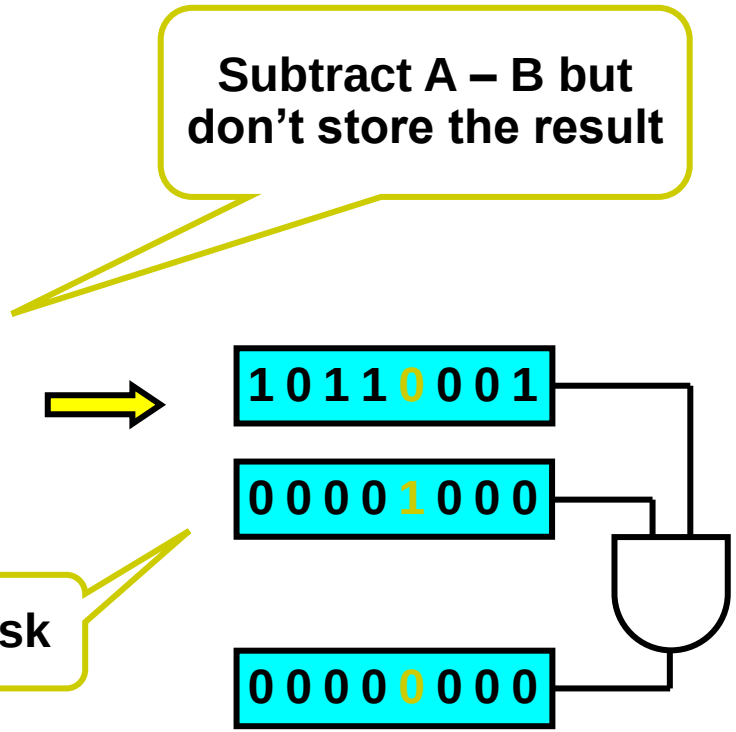
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate	NEG

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

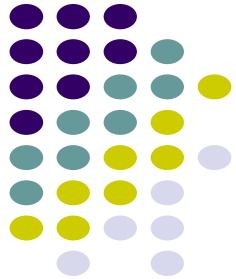
Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (Subtract)	CMP
Test (AND)	TST

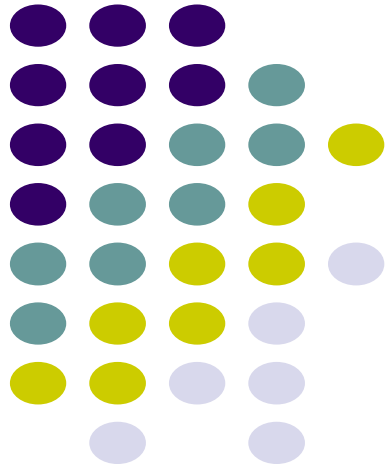


Conditional Branch Instructions

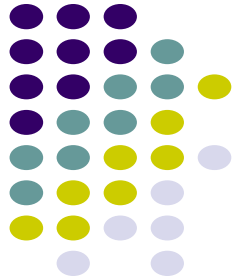


Mnemonic	Branch Condition	Tested Condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0

Basic Input/Output Operations

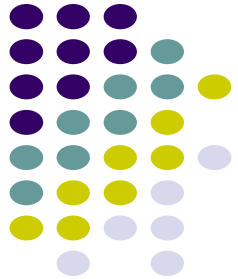


I/O



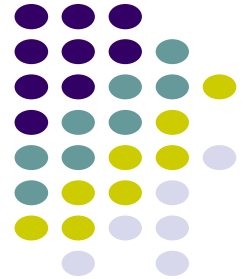
- The data on which the instructions operate are not necessarily already stored in memory.
- Data need to be transferred between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

Program-Controlled I/O Example



- Read in character input from a keyboard and produce character output on a display screen.
  - Rate of data transfer (keyboard, display, processor)
  - Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.
  - A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way.

Program-Controlled I/O Example



- Registers
- Flags
- Device interface

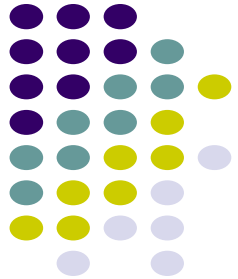
PROGRAM-CONTROLLED I/O: TAKING INPUT

- Suppose we want to read some characters from keyboard and show them on display.
- When we press a key on keyboard, corresponding character code is stored in an 8-bit buffer register associated with keyboard. Let's call this register **DATAIN.**
- To inform the processor that a valid character is in **DATAIN,** a status control flag, **SIN,** is set to 1.
- A program monitors SIN and when SIN is set to 1, the processor reads the content of DATAIN.
- When the character is transferred to the processor, SIN is automatically cleared to 0, and same process gets repeated for next character.

PROGRAM CONTROLLED I/O: SENDING OUTPUT

- Similar to taking input, when characters are transferred from the processor to the display, A buffer register **DATAOUT** and status control flag **SOUT** are used.
- When SOUT is set to 1, the display is ready to receive a character.
- Under program controlled I/O, processor monitors SOUT and when SOUT is set to 1, it transfers a character code to DATAOUT.
- The transfer of a character to DATAOUT clears SOUT to 0 and when display is ready to receive a second character, SOUT is again set to 1.

Program-Controlled I/O Example



- Machine instructions that can check the state of the status flags and transfer data:

```
READWAIT  Branch to READWAIT if SIN = 0
          Input from DATAIN to R1

WRITEWAIT Branch to WRITEWAIT if SOUT = 0
          Output from R1 to DATAOUT
```

MEMORY-MAPPED I/O

- Until now, we have assumed that the address issued by a processor to access instructions and operands always refer to memory locations.
- Many computers use an arrangement called *memory-mapped I/O* in which some memory address values are used to refer to peripheral device buffer registers such as **DATAIN** and **DATAOUT.**
- Therefore we can use general instructions like move to transfer data between processor and I/O devices.
- For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction -

  MoveByte R1, DATAIN

- MoveByte is used to denote transfer of single byte instead of MOVE instruction which transfers a word.
- The status flags SIN and SOUT are automatically cleared when the buffer registers DATAIN and DATAOUT are referenced, respectively.

MEMORY-MAPPED I/O

- It is quite common to include SIN and SOUT flags in *device status registers.* Let's assume that bit b3 in registers **INSTATUS** and **OUTSTATUS** corresponds to SIN and SOUT, respectively.
- We can implement read and write operations as follows -

```
READWAIT:
        TESTBIT INSTATUS, #3 @ It will check if 3rd bit of INSTATUS register is 0.
        BRANCH=0 READWAIT @ If 3rd bit is 0, then branch to READWAIT
        MOVEBYTE R1, DATAIN @ When there is data in DATAIN, transfer that data to R1


WRITEWAIT:
        TESTBIT OUTSTATUS, #3 @ It will check if 3rd bit of OUTSTATUS register is 0.
        BRANCH=0 WRITEWAIT @ If 3rd bit is 0, then branch to WRITEWAIT
        MOVEBYTE R1, DATAOUT @ When display is ready, transfer contents of R1 into DATAOUT
```

I/O OPERATIONS

- Program-Controlled I/O requires continuous involvement of the processor in the I/O operations.
- Since I/O operations are usually slow, it affects overall performance of processor also.
- Other I/O techniques, based on the use of interrupts may be used to improve the utilization of processor.

Subroutines

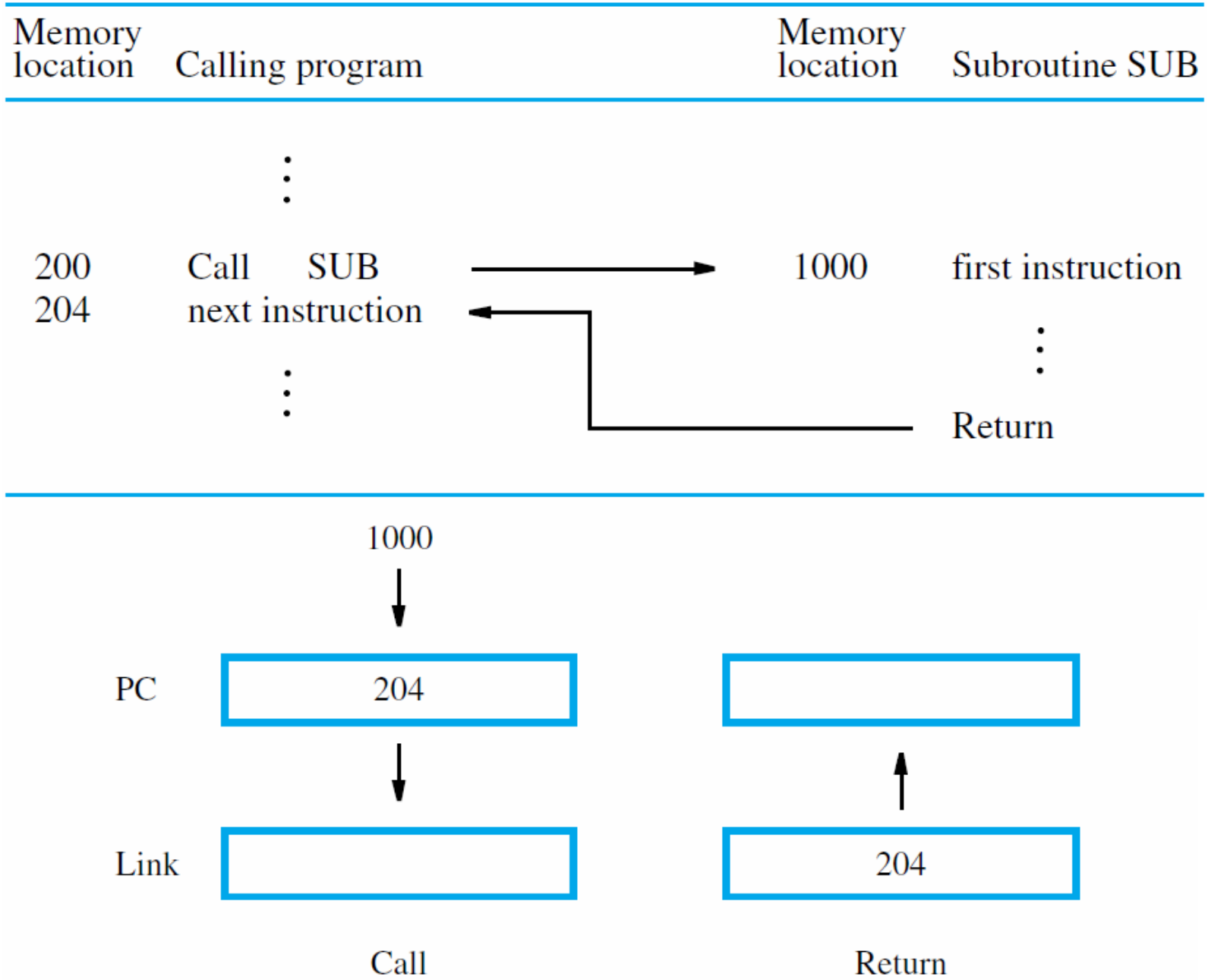
- In a given program, a particular task may be executed many times using different data
- Examples: mathematical function, list sorting
- Implement task in one block of instructions
- This is called a subroutine
- Rather than reproduce entire subroutine block in each part of program, use a subroutine call
- A Call instruction is a special type of branch

Subroutines

- Branching to same block of instructions
  saves space in memory, but must branch back
- The subroutine must return to calling program
  after executing last instruction in subroutine
- This branch is done with a Return instruction
- Subroutine can be called from different places
- How can return be done to the correct place?
- This is the issue of subroutine linkage

Subroutine Linkage

- During execution of Call instruction,
  PC upated to point to instruction after Call
- Save this address for Return instruction to use
- Simplest method: place address in link register
- Call instruction performs two operations:
  store updated PC contents in link register,
  then branch to target (subroutine) address
- Return just branches to address in link register


Memory location
Calling program
Memory location
Subroutine SUB
200
Call SUB
204
next instruction
1000
first instruction
Return
1000
PC
204
Link
204
Call
Return

Subroutine Nesting and the Stack

- We can permit one subroutine to call another,
  which results in subroutine nesting
- Link register contents after first subroutine call are overwritten
  after second subroutine call
- First subroutine should save link register
  on the processor stack before second call
- After return from second subroutine,
  first subroutine restores link register

Parameter Passing

- A program may call a subroutine many times with different data to obtain different results
- Information exchange to/from a subroutine is called parameter passing
- Parameters may be passed in registers
- Simple, but limited to available registers
- Alternative: use stack for parameter passing, and also for local variables & saving registers

Subroutine

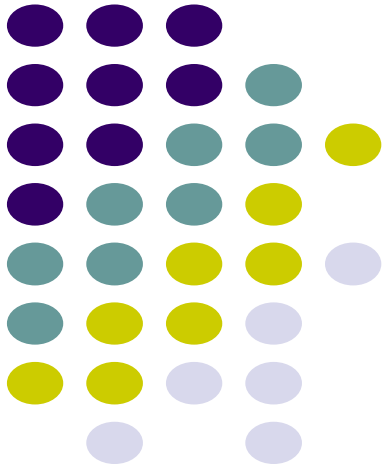
Calling program

```
            Load        R2, N          Parameter 1 is list size.
            Move        R4, #NUM1      Parameter 2 is list location.
            Call        LISTADD        Call subroutine.
            Store       R3, SUM        Save result.
            ⋮
```

Subroutine

```
LISTADD:    Subtract    SP, SP, #4     Save the contents of
            Store       R5, (SP)         R5 on the stack.
            Clear       R3             Initialize sum to 0.
LOOP:       Load        R5, (R4)       Get the next number.
            Add         R3, R3, R5     Add this number to sum.
            Add         R4, R4, #4     Increment the pointer by 4.
            Subtract    R2, R2, #1     Decrement the counter.
            Branch_if_[R2]>0 LOOP
            Load        R5, (SP)       Restore the contents of R5.
            Add         SP, SP, #4
            Return                     Return to calling program.
```

Additional Instructions



LOGICAL INSTRUCTIONS

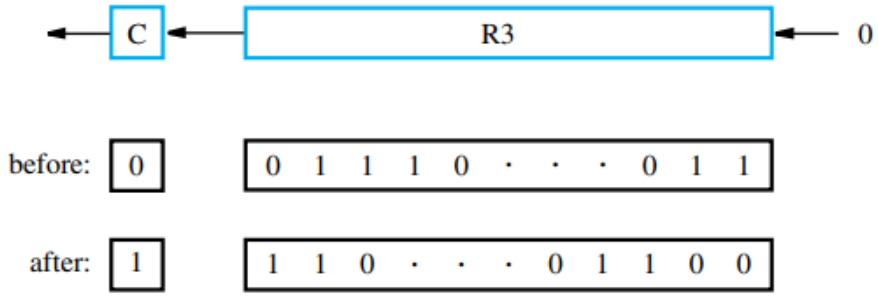
- AND, OR, and NOT operations on single bits are basic building blocks of digital circuits.
- Almost all Instruction Set Architectures provides, instructions to perform AND, OR and NOT operations on register data.

```
AND R1, R2, R3 @ Performs AND between R2 and R3 and stores the result into R1

OR R1, R2, R3 @ Performs OR between R2 and R3 and stores the result into R1

NOT R1, R2 @ Performs NOT operation on content of R2 and stores the result in R1
```

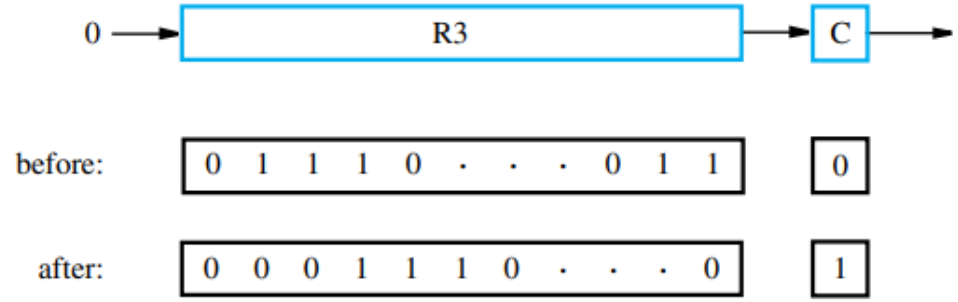
SHIFT INSTRUCTIONS

- Usually following shift instructions are provided by any ISA –
  - Logical Shift Left – **LShiftL R1, R0, #2 @** Equivalent to multiplication by 2
  - Logical Shift Right – **LShiftR R1, R0, #2 @** Equivalent to division by 2
  - Arithmetic Shift Right – **AShiftR R1, R0, #2 @** Equivalent to division by 2

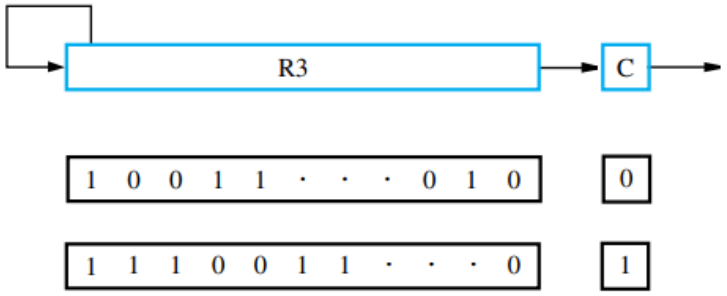
SHIFT INSTRUCTIONS



(a) Logical shift left LShiftL R3, R3, #2



(b) Logical shift right LShiftR R3, R3, #2

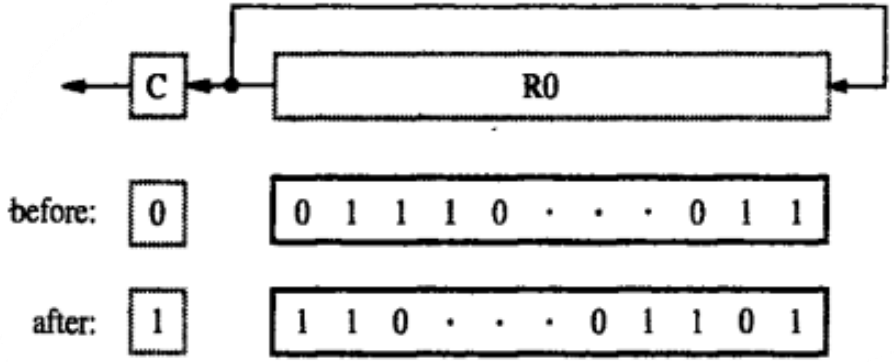


(c) Arithmetic shift right AShiftR R3, R3, #2

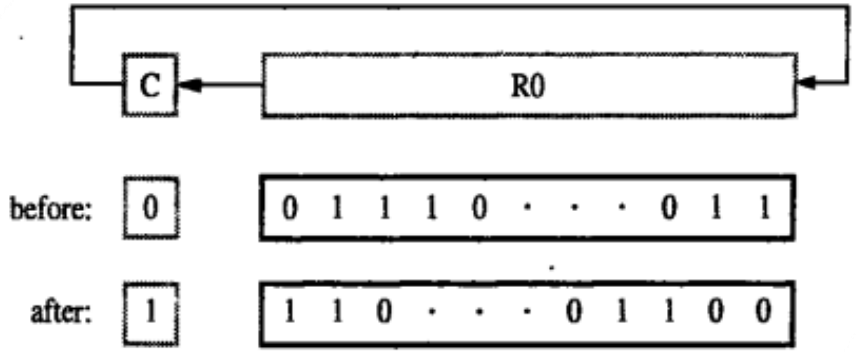
ROTATE INSTRUCTIONS

- Usually following rotate instructions are provided by any ISA –
  - Rotate Left without Carry – **RotateL R0, R0, #2**
  - Rotate Left with Carry – **RotateLC R0, R0, #2**
  - Rotate Right without Carry – **RotateR R0, R0, #2**
  - Rotate Right with Carry – **RotateRC R0, R0, #2**

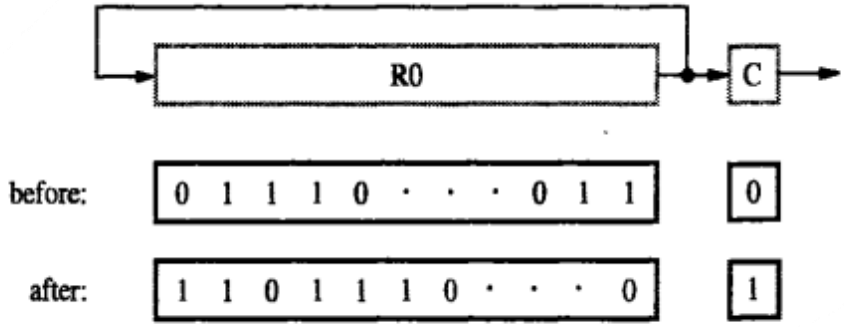
ROTATE INSTRUCTIONS



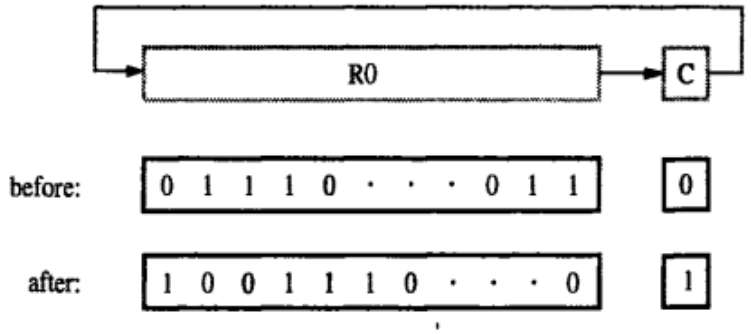
(a) Rotate left without carry



(b) Rotate left with carry



(c) Rotate right without carry



(d) Rotate right with carry

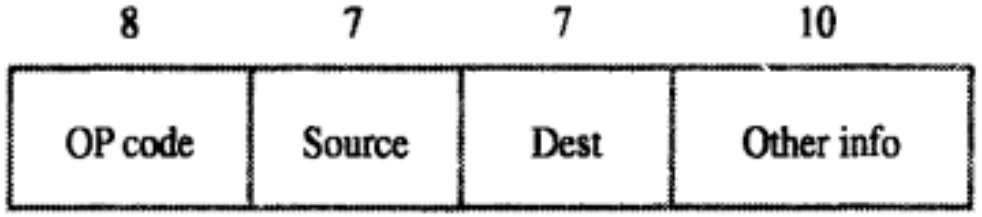
MULTIPLICATION & DIVISION INSTRUCTIONS

- Signed integer multiplication of *n*-bit numbers produces a product with up to 2*n* bits
- Processor truncates product to fit in a register:

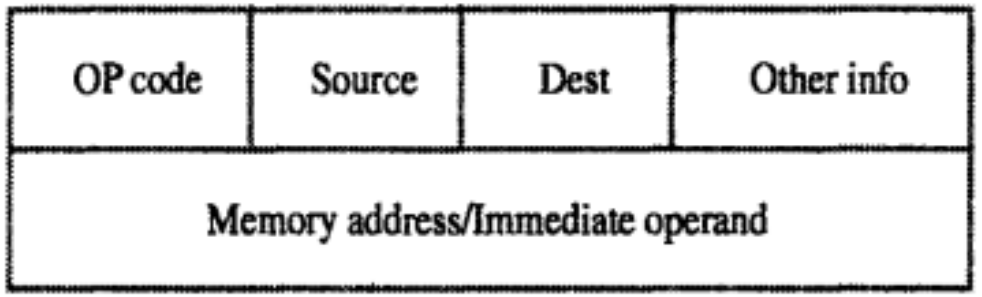
  Multiply R*k*, R*i*, R*j* (R*k* ← [R*i*] × [R*j*])
- For general case, 2 registers may hold result
- Integer division produces quotient as result:

  Divide R*k*, R*i*, R*j* (R*k* ← [R*i*] / [R*j*])
- Remainder is discarded or placed in a register

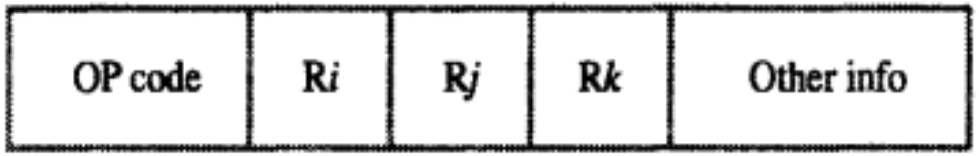
ENCODING OF MACHINE INSTRUCTIONS



(a) One-word instruction



(b) Two-word instruction



(c) Three-operand instruction