COMPUTER ORGANIZATION AND ARCHITECTURE

Computer Instruction Set

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

- **Instruction Set Architecture (ISA)**
  - An instruction set architecture (ISA) defines the set of basic operations a computer must support
  - ISA is the part of the processor architecture related to programming, including:
    - ✓ the native data types
    - ✓ Instructions
    - ✓ Registers
    - ✓ addressing modes
    - ✓ memory architecture
    - ✓ interrupt and exception handling
    - ✓ external I/O
  - Essentially, an ISA provides a blueprint that describes how software controls the hardware to perform computations.

Instruction

- A programmable system uses a sequence of instructions to control its operation
- A typical instruction specifies:
  - Operation to be performed
  - Operands to use, and
  - Where to place the result, or
  - Which instruction to execute next
- Instructions are stored in RAM or ROM as a program
- The addresses for instructions in a computer are provided by a program counter (PC) that can
  - Count up
  - Load a new address based on an instruction and, optionally, status information
- The PC and associated control logic are part of the Control Unit
- Executing an instruction - activating the necessary sequence of operations specified by the instruction
- Execution is controlled by the control unit and performed:
  - In the datapath
  - In the control unit
  - In external hardware such as memory or input/output

Instruction Formats

- Computer instructions are a set of machine language instructions that a particular processor understands and executes.
- Instruction is represented as a sequence of bits.
- A computer performs tasks on the basis of the instruction provided.
- An instruction comprises of groups called fields. These fields include:
  - **Operation code (Opcode) field:** which specifies the operation to be performed.
  - **Address field:** contains the location of the operand, i.e., register or memory location.
  - **Mode field:** specifies how the operand will be located.
- Other special fields are sometimes employed under certain circumstances
  - For example a field that gives the number of shifts in a shift-type instruction.

Instruction Formats

- The ***operation code field*** of an instruction is a group of bits that define various processor operations: *add, subtract, complement, and shift*.
- The bits that define the ***mode field*** of an instruction code specify a variety of alternatives for choosing the operands from the given address.
- The various ***addressing modes that*** have been formulated for digital computers
- Operations specified by computer instructions are executed on some data stored in memory or processor registers.
- Operands residing in memory are specified by their memory address.
- Operands residing in processor registers are specified with a register address.
- A register address is a binary number of **k bits** that defines one of $\mathbf{2^k}$ registers in the CPU.
- For example: CPU with 16 processor registers RO through R15 will have a register address field of four bits.
- The binary number 0101, for example, will designate register R5.

Instruction Formats

- Computers may have instructions of several different lengths containing varying number of addresses.
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers.
- Most computers fall into one of three types of CPU organizations:
  1. Single accumulator organization.
  2. General register organization.
  3. Stack organization.
- Depending on number of addresses, the instructions are classified into following categories:
  - Three Address Instructions
  - Two Address Instructions
  - One Address Instruction
  - Zero Address Instructions

Instruction Formats

Three Address Instructions

- Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

- Evaluation of expression using three address instructions: **X = (A + B) • (C + D)**

```
ADD    R1, A, B     R1 ← M[A] + M[B]
ADD    R2, C, D     R2 ← M[C] + M[D]
MUL    X, R1, R2    M[X] ← R1 * R2
```

- It is assumed that the computer has two processor registers, R 1 and R2.
- The symbol M [A] denotes the operand at memory address symbolized by A.
- **Advantage:** it results in short programs when evaluating arithmetic expressions
- **Disadvantage:** the binary-coded instructions require too many bits to specify three addresses

Instruction Formats

Two Address Instructions

- Two-address instructions are the most common in commercial computers.
- Here again each address field can specify either a processor register or a memory word.
- The program to evaluate **X = (A + B) • (C + D)** is as follows:

```
MOV   R1, A     R1 ← M[A]
ADD   R1, B     R1 ← R1 + M[B]
MOV   R2, C     R2 ← M[C]
ADD   R2, D     R2 ← R2 + M[D]
MUL   R1,R2     R1 ← R1*R2
MOV   X, R1     M[X] ← R1
```

- The MOV instruction moves o r transfers the operands t o and from memory and processor registers.
- The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

Instruction Formats

One Address Instruction

- One-address instructions use an implied accumulator (AC) register for all data manipulation.
- For multiplication and division there is a need for a second register.
- However, here we will neglect the second register and assume that the AC contains the result of all operations.
- The program to evaluate **X = (A + B) • (C + D)** is

```
LOAD    A    AC ← M[A]
ADD     B    AC ← AC + M[B]
STORE   T    M[T] ← AC
LOAD    C    AC ← M[C]
ADD     D    AC ← AC + M[D]
MUL     T    AC ← AC * M[T]
STORE   X    M[X] ← AC
```

- All operations are done between the AC register and a memory operand.
- T is the address of a temporary memory location required for storing the intermediate result.

Instruction Formats

Zero Address Instructions

- A stack-organized computer does not use an address field for the instructions ADD and MUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- The following program shows how **X = (A + B) • (C + D)** will be written for a stack organized computer. (TOS stands for top of stack.)

```
PUSH    A    TOS←A
PUSH    B    TOS←B
ADD          TOS←(A + B)
PUSH    C    TOS←C
PUSH    D    TOS←D
ADD          TOS←(C + D)
MUL          TOS←(C + D)*(A + B)
POP     X    M[X]←TOS
```

- The name "***zero-address***" is given to this type of computer because of the absence of an address field in the computational instructions.

Addressing Modes

- **Operation field:** specifies the operation to be performed.
- **Operation** must be executed on some data stored in computer registers or memory words.
- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
  1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  2. To reduce the number of bits in the addressing field of the instruction.

Addressing Modes

- Different types of addressing modes are:
  1. Implied / Implicit Addressing Mode
  2. Immediate Addressing Mode
  3. Register Addressing Mode
  4. Register Indirect Addressing Mode
  5. Auto-Increment or Auto-Decrement Addressing Mode
  6. Direct Addressing Mode
  7. Indirect Addressing Mode
  8. Relative Addressing Mode
  9. Indexed Addressing Mode
  10. Base Register Addressing Mode

Addressing Modes

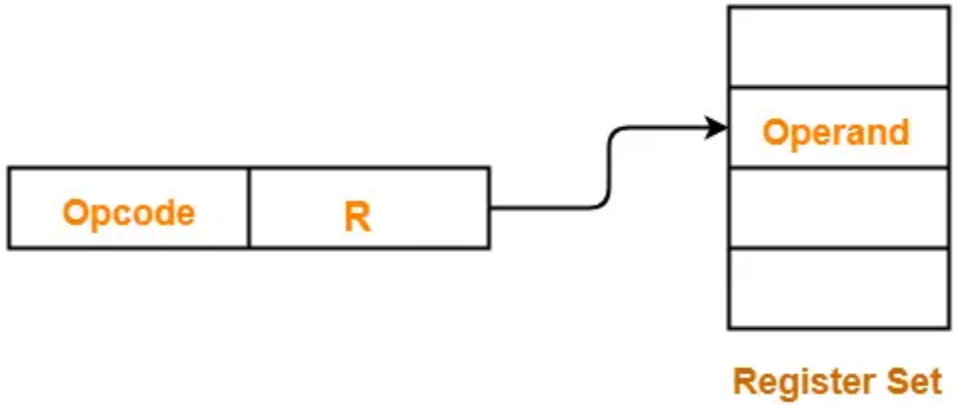
1. Implied / Implicit Addressing Mode

- In this mode the operands are specified implicitly in the definition of the instruction.
- For example, the instruction "complement accumulator“ is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- **Example: CMA**
- In fact, all register reference instructions that use an accumulator are implied-mode instructions
- Zero-address instructions in a stack-organized computer are implied-mode instructions
- **Example: MUL** *Pop top two items from stack and add*

Addressing Modes

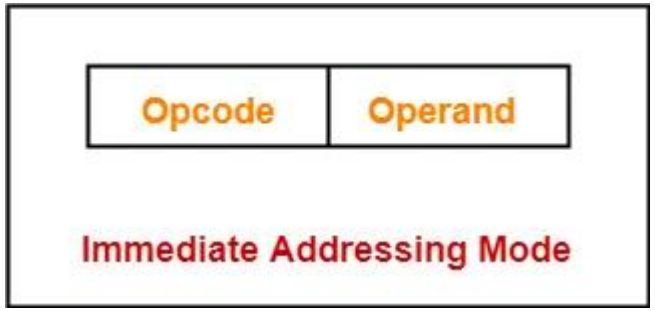
2. Register Addressing Mode

- In this mode the operands are in registers that reside within the CPU.
- The particular register is selected from a register field in the instruction.
- **Example: ADD R0, R1**



3. Immediate Addressing Mode

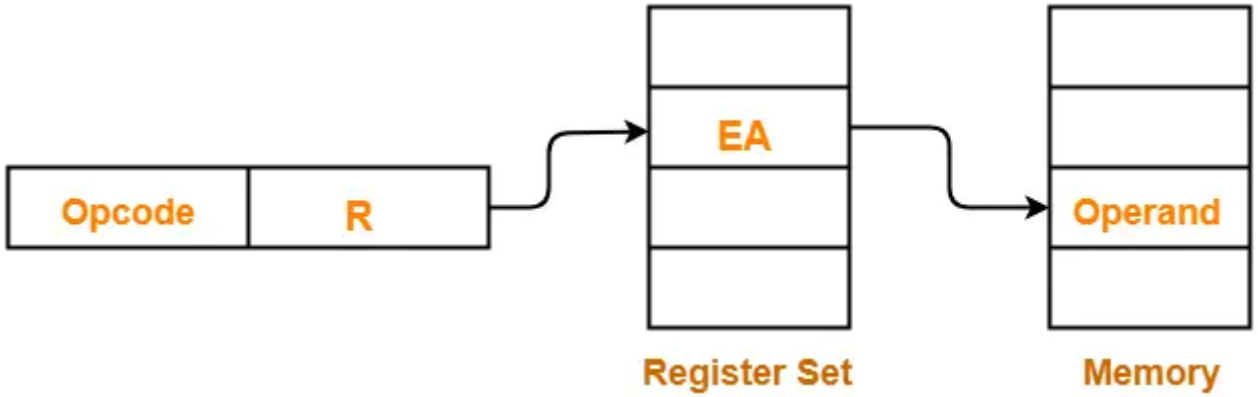
- In this mode the operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand field rather than an address field.
- **Example: ADD A, #0x31**



Addressing Modes

4. Register Indirect Addressing Mode

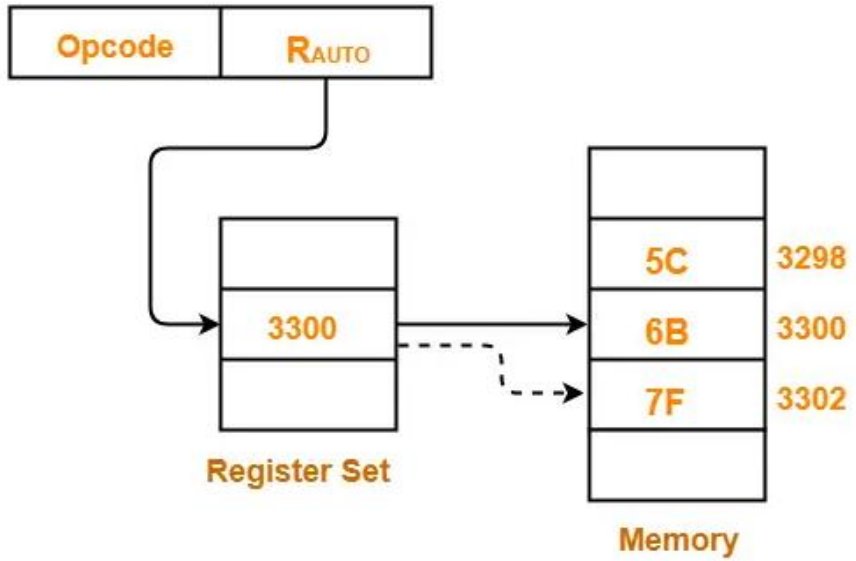
- In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory
- This method allows for the execution of the same set of instructions for different memory locations by incrementing the register content and pointing to the new location each time.
- The indirect mode is denoted by enclosing the given register in parentheses.
- In this scenario, the effective address (EA) is the content of the memory location found in the register.
- EA=(R)
- **Example: LOAD R1, (R2)**



Addressing Modes

5. Auto-Increment or Auto-Decrement Addressing Mode

- This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.
- This can be achieved by using the increment or decrement instruction.
- The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.
- **Example of ARM processor instructions**
- --- Auto pre increment/decrement addressing mode
- LDR r0, [r1, #+/-4]!
- Auto post increment/decrement addressing mode
- LDR r0, [r1], #+/-4

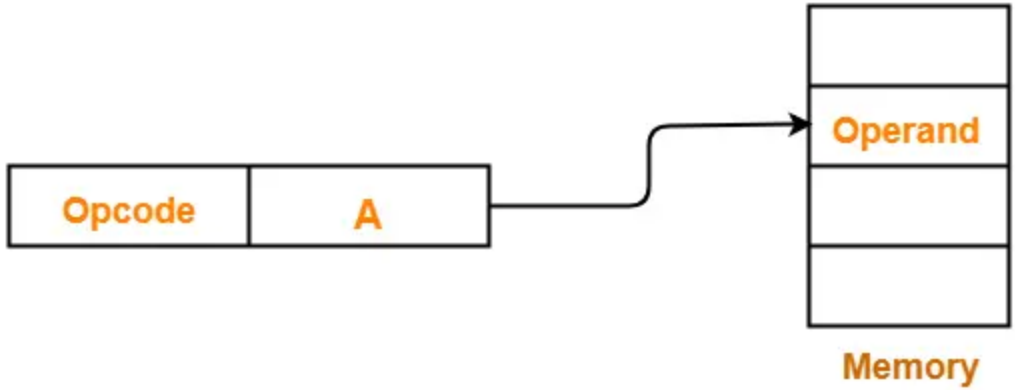


Addressing Modes

6. Direct Addressing Mode

- In this mode the effective address is equal to the address part of the instruction.
- The operand resides in memory and its address is given directly by the address field of the instruction.
- The Direct addressing mode is one that contains the actual address of the data.
- In a branch-type instruction the address field specifies the actual branch address
- The direct addressing mode is also known as the ***absolute addressing mode***.

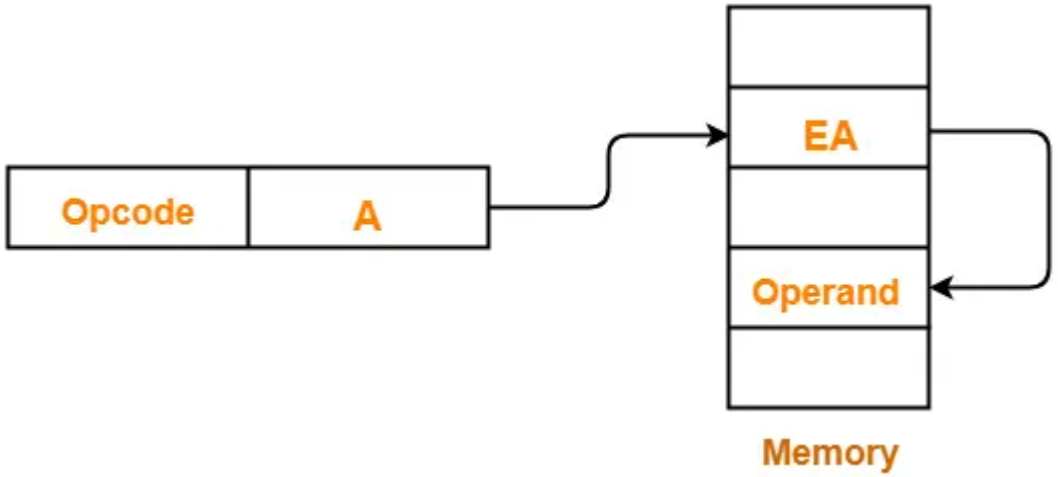
- **Example:**
- MOV A, 81H ; 8051 microcontroller instruction
- JMP 5000H
- MOV R1, @6000H



Addressing Modes

7. Indirect Addressing Mode

- In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- Two references to memory are required to fetch the operand.
- **Example:**
- ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.
- AC ← AC + ((X))



Addressing Modes

- A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- The effective address in these modes is obtained from the following computation:

  effective address(EA) = address part of instruction + content of CPU register

- The CPU register used in the computation may be the:
  - ✓program counter
  - ✓An index register, or
  - ✓a base register.
- In either case we have a different addressing mode which is used for a different application.

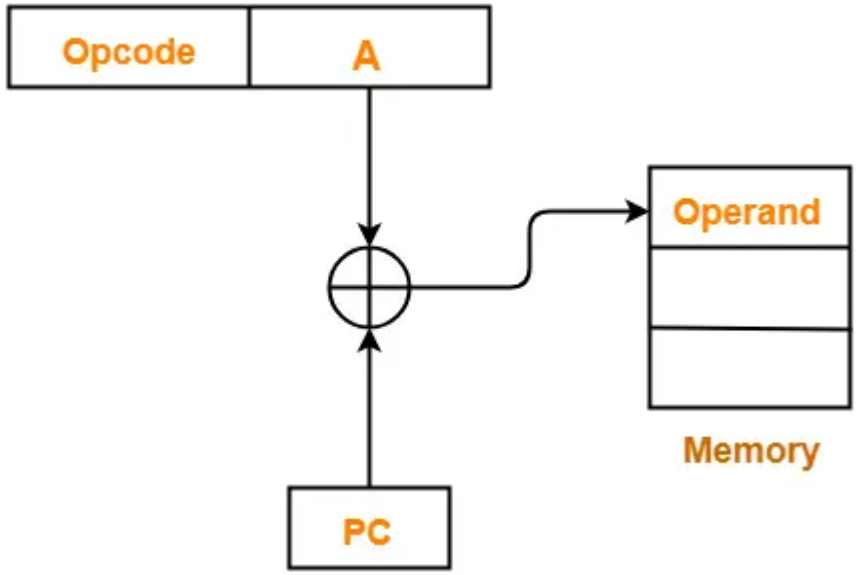
Addressing Modes

8. Relative Addressing Mode

- In this mode the content of the program counter/specified register is added to the address part of the instruction in order to obtain the effective address.
- The address part of the instruction is usually a signed number (in 2' s complement representation) which can be either positive or negative.
- When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

- **Example:**

- JMP 20H ; 8051 instruction

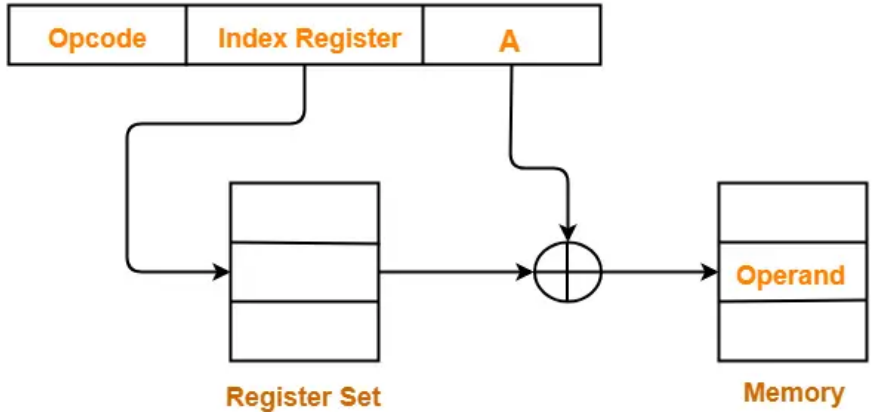


Addressing Modes

9. Indexed Addressing Mode

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- The index register is a special CPU register that contains an index value.
- The address field of the instruction defines the beginning address of a data array in memory.
- Each operand in the array is stored in memory relative to the beginning address.
- The distance between the beginning address and the address of the operand is the index value stored in the index register.
- **Effective Address = Content of Index Register + Address part of the instruction**

- **Example:**
- MOV AX, [SI+2000H] ; 8086 instructions



Instruction types

- Depending on operation they perform, all instructions are divided in several groups
  - Data transfer instructions
  - Data manipulation instructions
    - ✓ Arithmetic instructions
    - ✓ Logical and bit manipulation instructions
    - ✓ Shift instructions
  - Program control instructions
  - Input/Output Instruction

Instruction types

Data transfer Instructions

- Data transfer instructions transfer the data between
  - ✓ memory and processor registers, processor registers, I/O devices, and from one processor register to another.

Name	Mnemonics	Operations
Load	LD	transfer data from the memory to a processor register, which is usually an accumulator
Store	ST	transfers data from processor registers to memory
Move	MOV	transfers data from processor register to memory or memory to processor register or between processor registers itself
Exchange	XCH	swaps information either between two registers or between a register and a memory word
Input	IN	transfers data between the processor register and the input terminal
Output	OUT	transfers data between the processor register and the output terminal
Push	PUSH	transfer data between a processor register and memory stack
Pop	POP	transfer data between a processor register and memory stack

Instruction types

Arithmetic Instructions

- The four basic arithmetic operations are addition, subtraction, multiplication, and division.
- Most computers provide instructions for all four operations.
- Some small computers have only addition and possibly subtraction instructions.
- The multiplication and division must then be generated by means of software subroutines.

Name	Mnemonics	Operations
Increment	INC	adds 1 to the value stored in the register or memory word
Decrement	DEC	subtracts 1 from the contents stored in the register or memory word
Add	ADD	adds register/memory with accumulator
Subtract	SUB	subtract register/memory with accumulator
Multiply	MUL	multiply register/memory with accumulator
Divide	DIV	Divide register/memory with accumulator
Add with carry	ADDC	adds register/memory with accumulator with carry
Subtract with borrow	SUBB	subtract register/memory with accumulator with borrow
Negate (2's comp)	NEG	2's complement of accumulator

Instruction types

Logical and bit manipulation Instructions

- Logical instructions carry out binary operations on the bits stored in the registers.
- In logical operations, each bit of the operand is treated as a Boolean variable.
- Logical instructions can change bit value, clear a group of bits, or can even insert new bit value into operands that are stored in registers or memory words.

Name	Mnemonics	Operations
Clear	CLR	Clear register or memory
Complement	COM	Complement register or memory
AND	AND	Logical Bitwise AND operations with register or memory
OR	OR	Logical Bitwise OR operations with register or memory
Exclusive-OR	XOR	Logical Bitwise Exclusive OR operations with register or memory
Clear carry	CLRC	Carry ← 0
Set carry	SETC	Carry ← 1
Complement carry	COMC	Carry ← (complement of Carry)
Enable interrupt	EI	Enable maskable interrupt
Disable interrupt	DI	Disable maskable interrupt

Instruction types

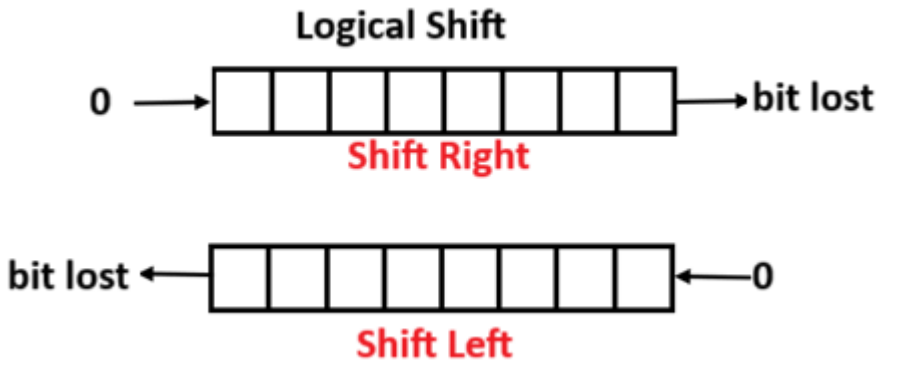
Shift instructions

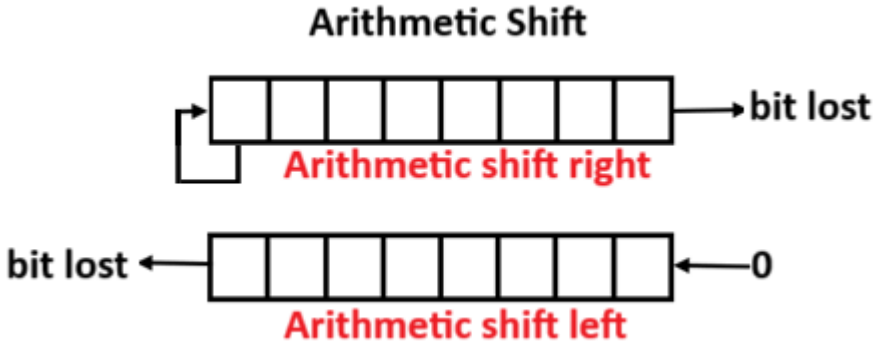
- Shifts are operations in which the bits of a word are moved to the left or right.
- The bit shifted in at the end of the word determines the type of shift used.
- Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations.
- In either case the shift may be to the right or to the left.

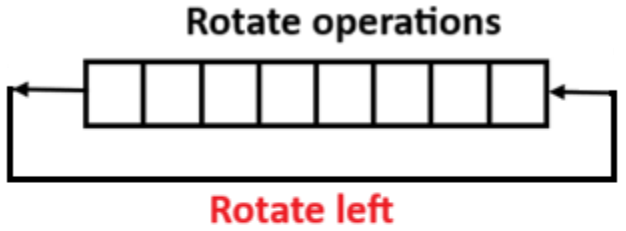
Name	Mnemonics	Operations
Logical shift left	SHL	Shift one bit position to left
Logical shift right	SHR	Shift one bit position to right
Arithmetic shift left	SHLA	Arithmetic shift one bit position to left
Arithmetic shift right	SHRA	Arithmetic shift one bit position to right
Rotate right	ROR	Rotate one bit position to right
Rotate left	ROL	Rotate one bit position to left
Rotate right through carry	RORC	Rotate one bit position to right through carry
Rotate left through carry	ROLC	Rotate one bit position to left through carry

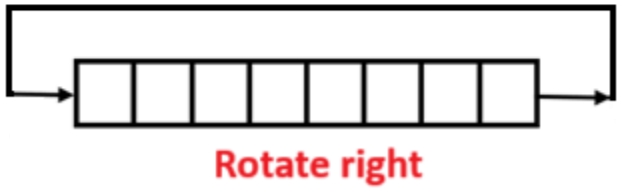
Instruction types

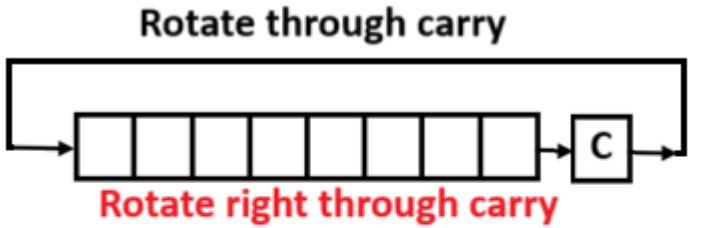
Shift operations

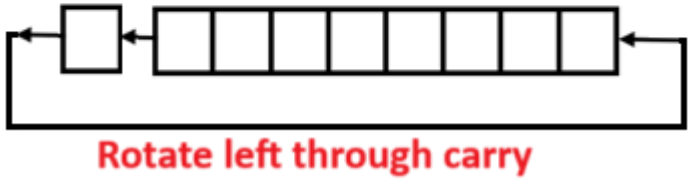












Instruction types

Program control instructions

- Instructions are always stored in successive memory locations.
- When processed in the CPU, the instructions are fetched from consecutive memory locations and executed.
- Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence.
- A ***program control type of instruction***, when executed, may change the address value in the program counter and cause the flow of control to be altered
- The change in value of ***the program counter*** as a result of the execution of a program control instruction ***causes a break*** in the sequence of instruction execution.
- This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Instruction types

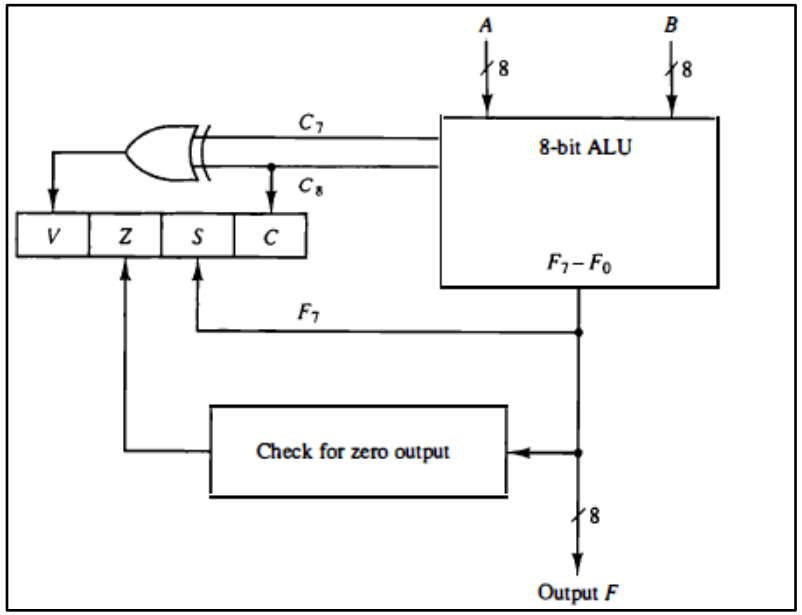
Program control instructions

Name	Mnemonics	Operations
Branch	BR	Jump to a specified memory locations
Jump	JMP	Jump to a specified memory location
Skip	SKP	Skip the next instruction
Call	CALL	Call subroutine
Return	RET	Return from subroutine
Compare (by subtraction)	CMP	Compare two operands
Test (by ANDing)	TST	Logically AND bits in the operands

Instruction types

Status register

- Stores status of the result produced by the ALU used for further analysis.
- Status bits are also called ***condition-code bits or flag bits***.
- The four status bits: C. S, Z, and V.
- The bits are set or cleared as a result of an operation performed in the ALU.



- ✓ ***Bit C (carry):*** set to 1 if the end carry C8 is 1 otherwise it is cleared to 0
- ✓ ***Bit S (sign):*** set to 1 if MSB of the result is 1 otherwise it is cleared to 0
- ✓ ***Bit Z (zero):*** set to 1 if the result contains all O's, otherwise it is cleared to 0
- ✓ ***Bit V (overflow):*** set to 1 if the result exceds the given range otherwise it is cleared to 0

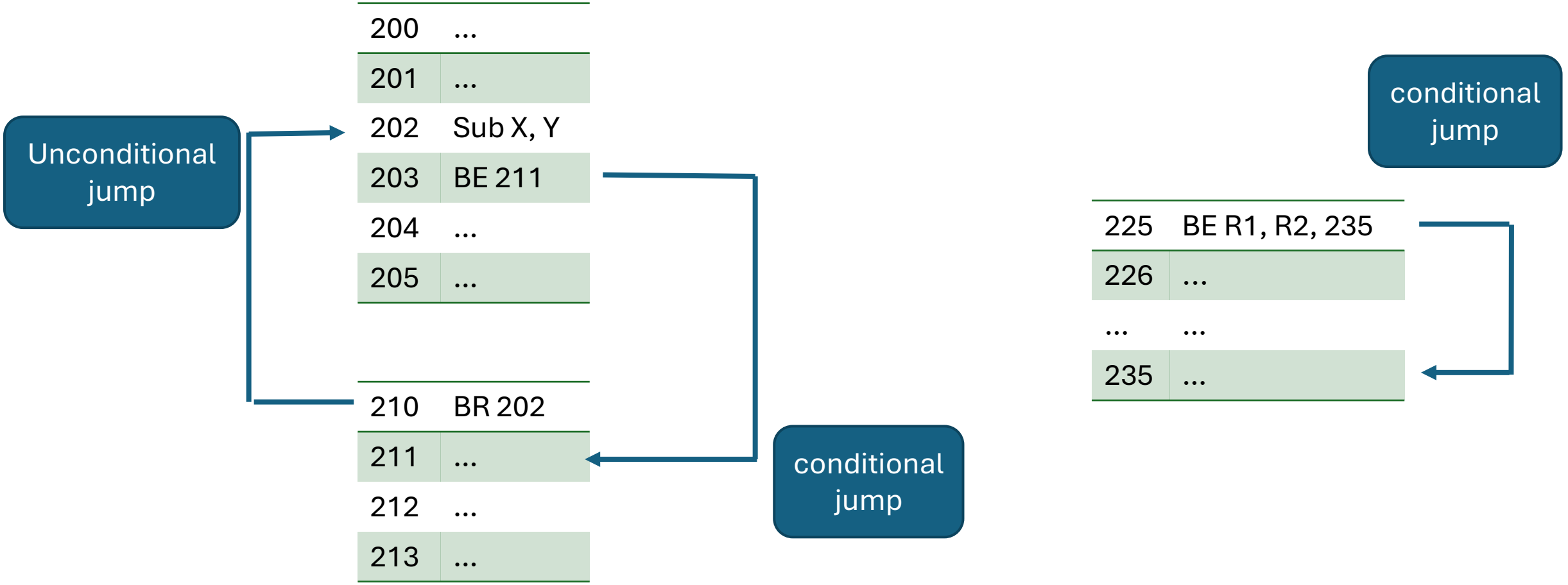
Instruction types

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	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Instruction types

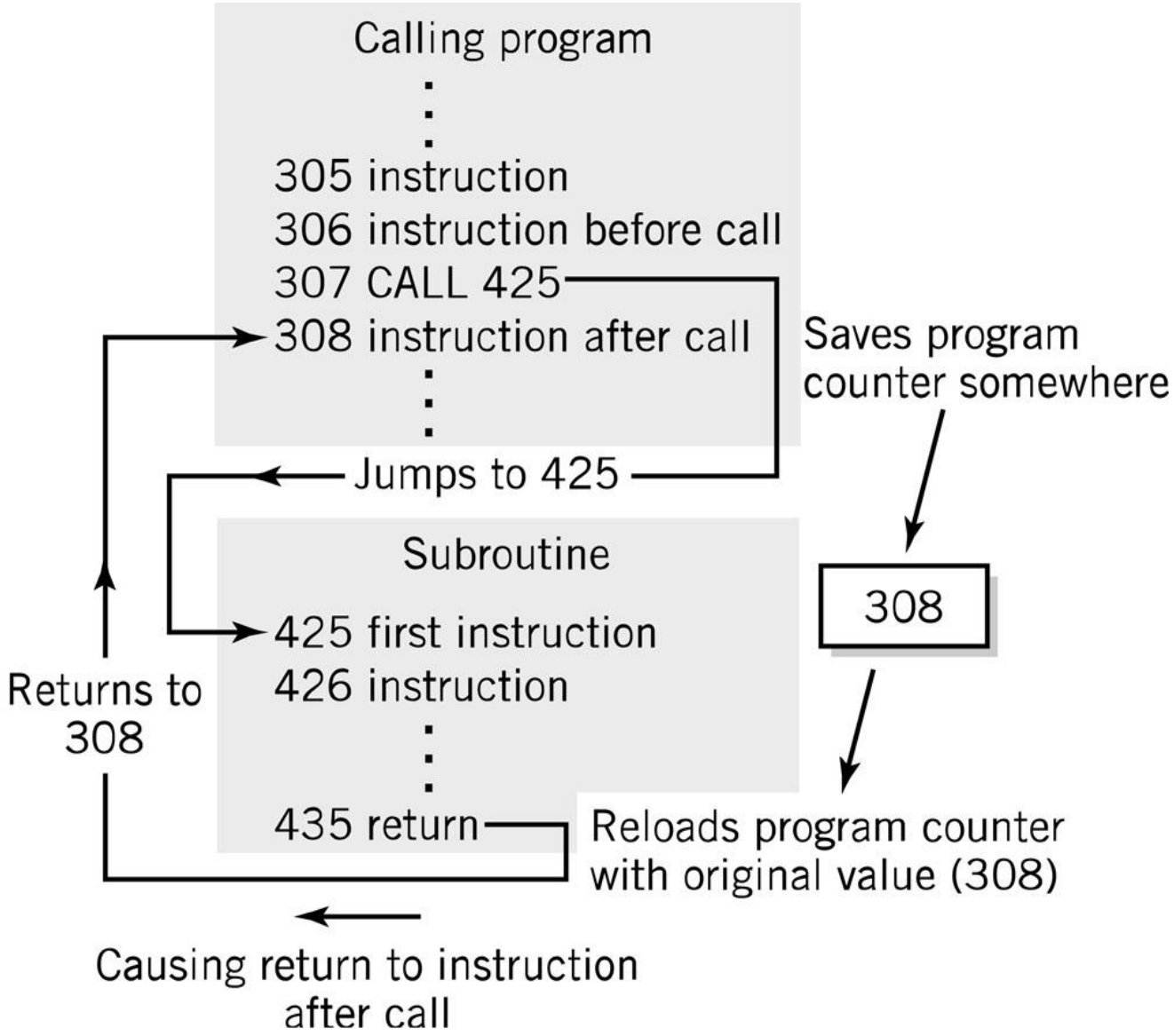
- **unconditional and conditional jumps**



Instruction types

CALL and RET illustration

- Call => Push PC followed by a jump to
- RET => Pop PC



Instruction types

Nested Procedure Calls

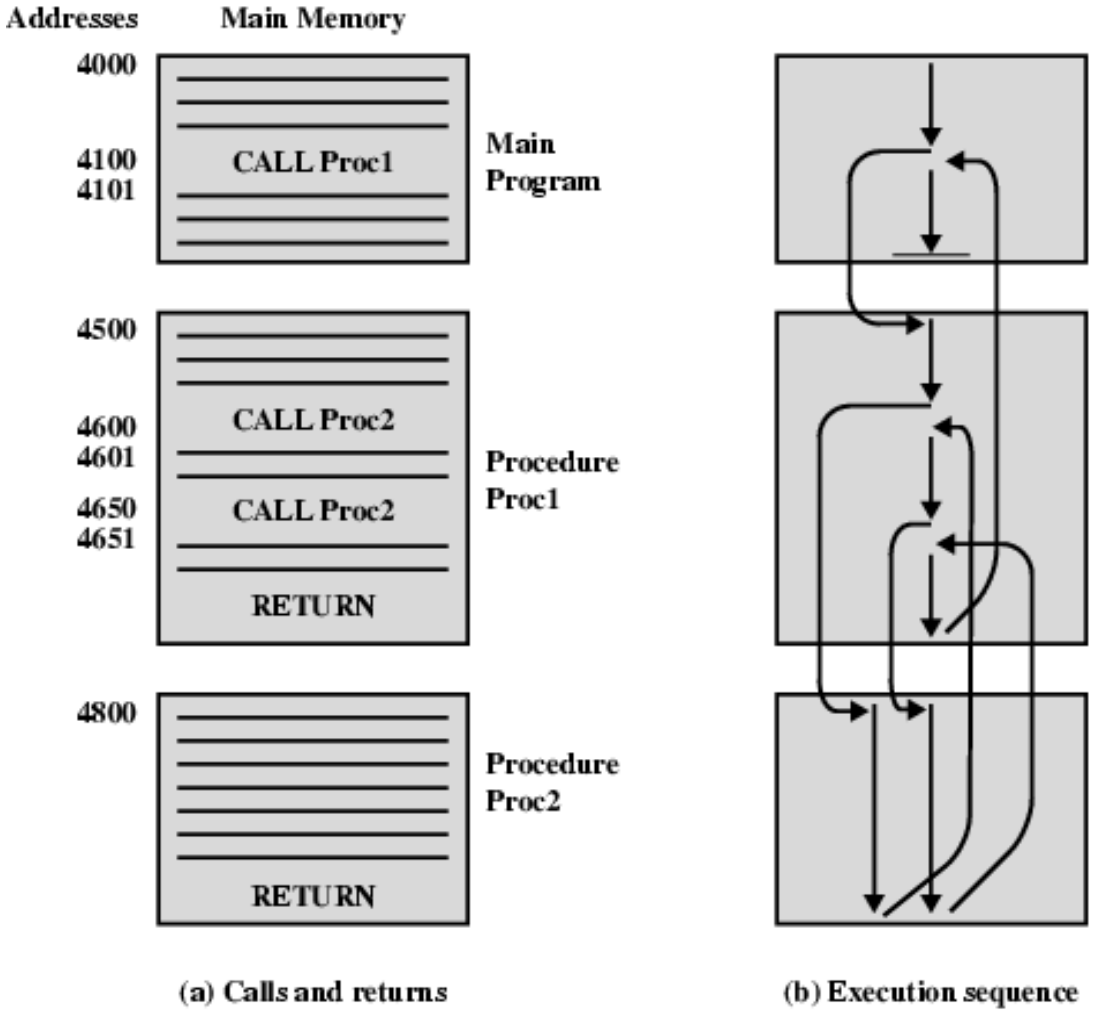
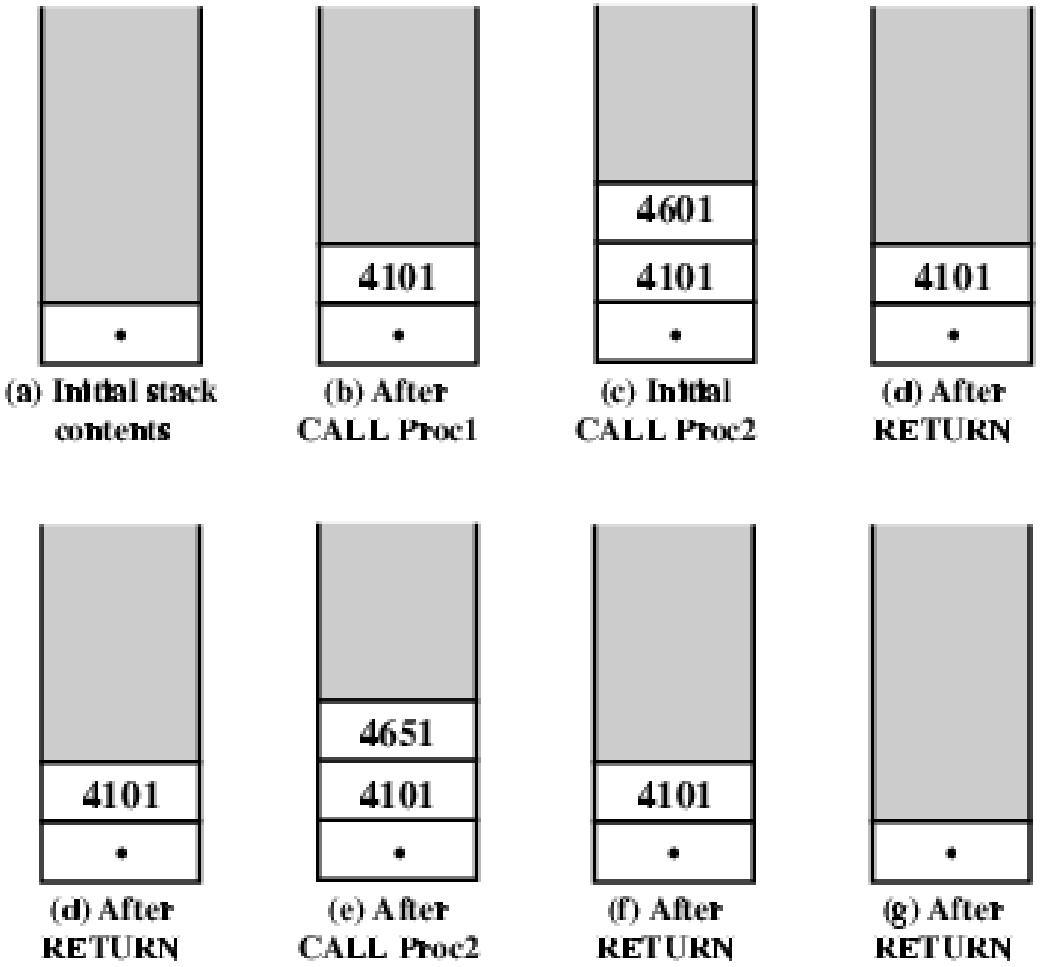


Illustration of function call's effect on stack



Instruction types

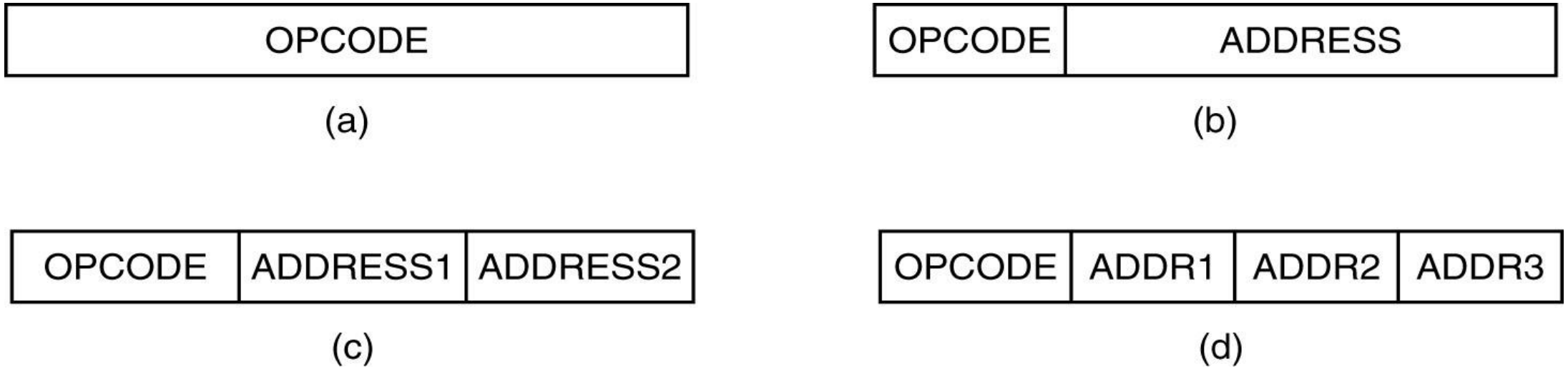
Input/Output Instruction

- The I/O devices are given specific addresses.
- The processor similarly views the I/O operations as memory operations.
- It include the address for the device.
- transfer data to or from the AC register.

Symbol	Description
INP	The INP instruction address the information from the INPR to AC which has 8 low order bits. It also clears the input flag to 0.
OUT	It can send the 8 low order bits from AC into output register OUTPR. It also clears the output flag to 0.

Opcode/Instruction Encoding

- A processor can execute an instruction only if it is represented as a binary sequence
- A unique binary pattern must be assigned to each opcode
- Process is known as opcode encoding
- Some popular upcoding techniques are:
  - Block code technique
  - Expanding opcode technique
  - Huffmann encoding

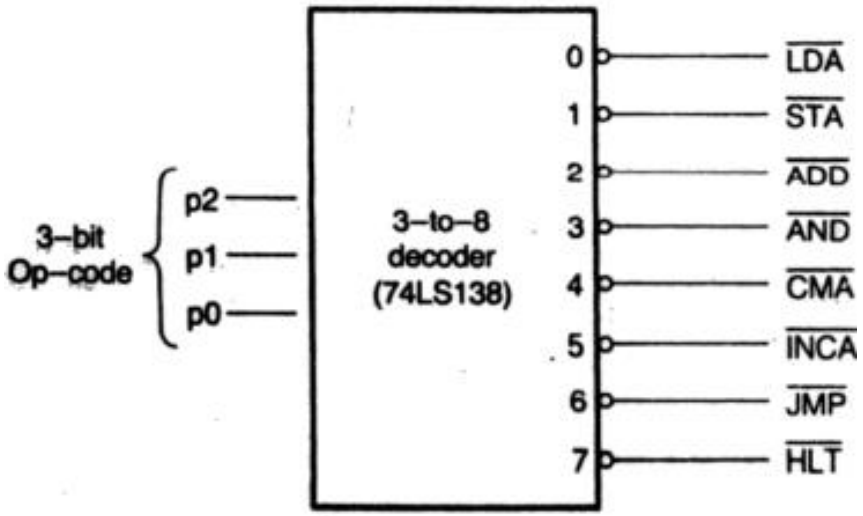


Opcode/Instruction Encoding

Block code technique

- Simplest way to carry out output encoding
- Assign a ***fixed length*** of binary pattern to each opcode
- For example: a k-bit binary pattern can represent 2^k distinct opcodes
- This method is known as a block code encoding
- **Example:** Hypothetical instruction set shown in table has 8 different instructions encoded using 3-bit binary pattern

Op-Code	Binary Pattern
	$p_2p_1p_0$
LDA	000
STA	001
ADD	010
AND	011
CMA	100
INCA	101
JMP	110
HLT	111



Opcode/Instruction Encoding

Expanding opcode technique

- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as ***HALT***, necessarily waste some space when fixed length instructions are used.
- One way to recover some of this space is to use expanding opcodes.
- The idea of expanding opcodes is to make some opcodes short, but have a means to provide longer ones when needed.
- When the opcode is short, a lot of bits are left to hold operands
- So, we could have two or three operands per instruction
- If an instruction has no operands (such as Halt), all the bits can be used for the opcode
- Many unique instructions are hence available
- In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands.

Opcode/Instruction Encoding

Expanding opcode technique

- **Example:** consider an instruction set of length 16-bit
- **Option 1**

4-bit	12-bit
Opcode	Address

- Possible operations = $2^4=16$
- Memory locations accessible: $2^{12}=4096$
- **Option 2**

3-bit	13-bit
Opcode	Address

- Possible operations = $2^3=8$
- memory locations accessible: $2^{13}=8192$

Expanding opcode technique

- **Option 3**

5-bit	11-bit
Opcode	Address

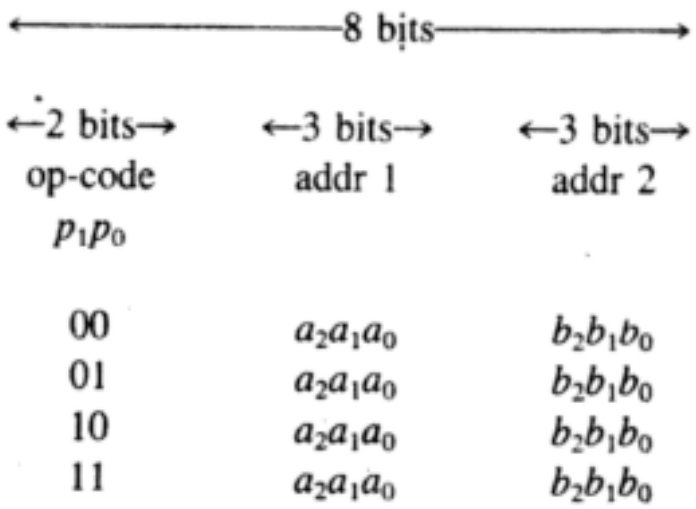
- Possible operations = $2^5=32$
- Memory locations accessible: $2^{11}=2048$
- **Option 2**
- Reduces the no. of possible operations by 50% compared to option 1
- Increase the memory accessibility by 100%
- **Option 3**
- Increases the no. of possible operations by 100% compared to option 1
- Decrease the memory accessibility by 50%

Opcode/Instruction Encoding

Expanding opcode technique

- Instruction format with an instruction length 8-bits and address field is 3-bit.
- **Two address instruction format**

8-bits		
3-bits	3-bits	3-bits
Opcode	**Address 1**	**Address 2**

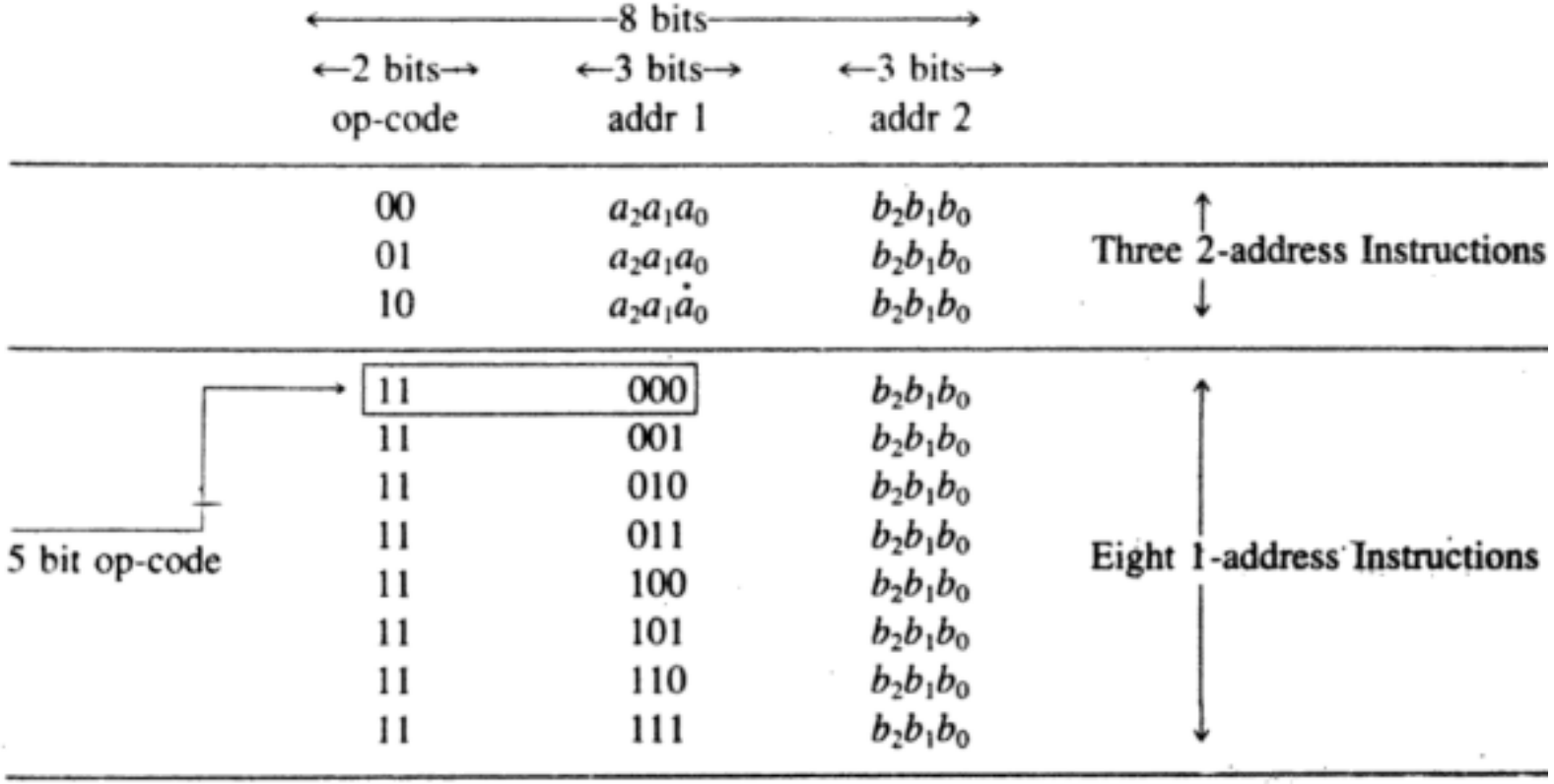


- ***4 two-address instructions can be obtained***

Opcode/Instruction Encoding

Expanding opcode technique

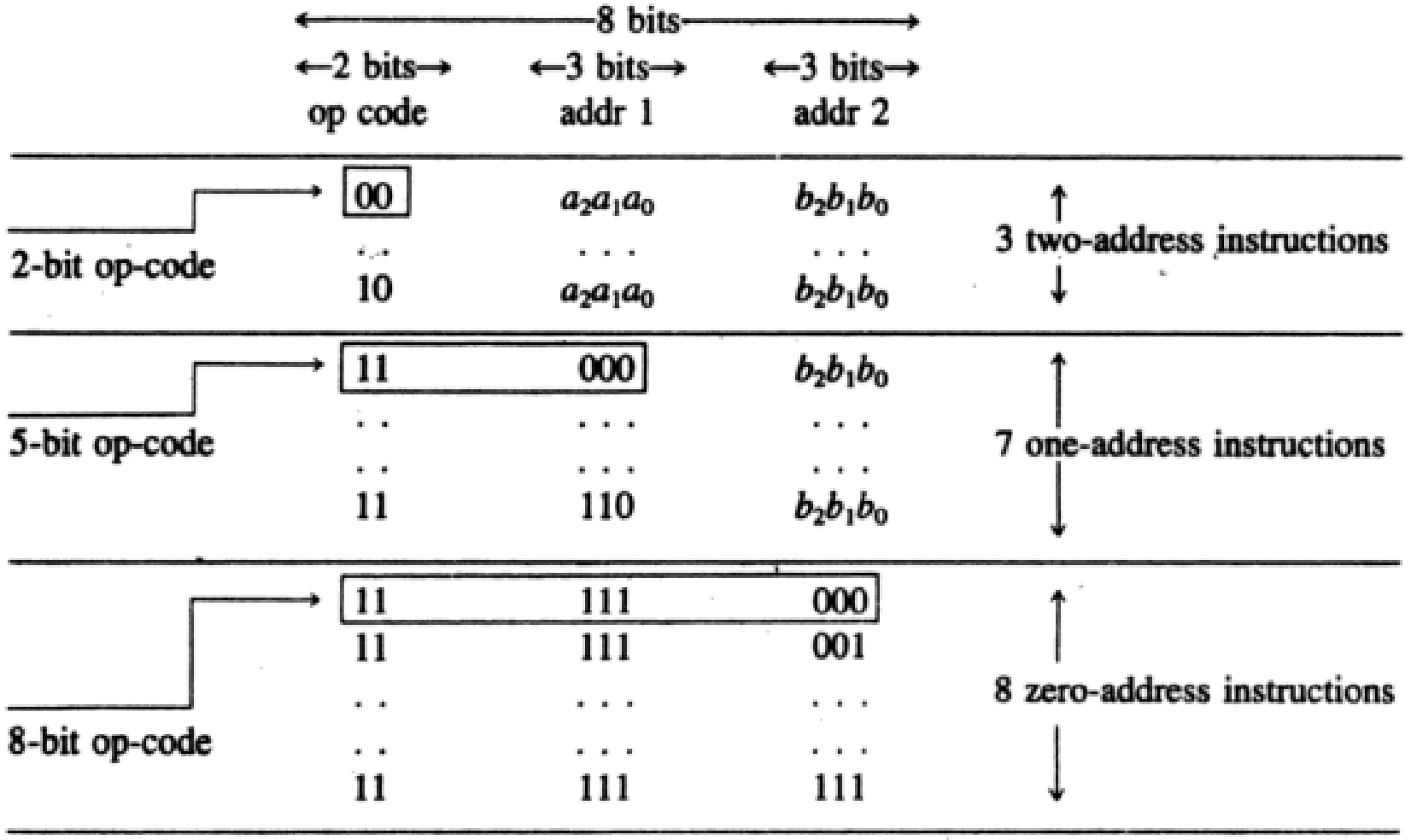
- **Example: 3 two-address instructions and 8 one-address instructions**
- Each one-address instruction requires only one address field, so opcode bits + one address bits are used to form 8 opcodes i.e. opcode for each one-address instruction is 5-bits
- ***This technique is referred to as expanding opcode technique***



Opcode/Instruction Encoding

Expanding opcode technique

- **Example: 3 two-address, 7 one-address and 8 zero address instructions**



Opcode/Instruction Encoding

- **Expanding opcode technique**
  - *How do we know if the instruction set we want is possible when using expanding opcodes?*

Ans: We must determine if we have enough bits to create the desired number of bits patterns

Opcode/Instruction Encoding

Expanding opcode technique

- Consider a machine with 16-bit instructions and 16 registers. We wish to encode the following instructions:
  - ✓ 15 instructions with 3 addresses
  - ✓ 14 instructions with 2 addresses
  - ✓ 31 instructions with 1 address
  - ✓ 16 instructions with 0 addresses

Can we encode this instruction set in 16 bits?

Opcode/Instruction Encoding

Expanding opcode technique

- Having a total of 16-bits we can create 2^{16} = 65536 bit patterns

- 15 instructions with 3 addresses
  - $15x2^4x2^4x2^4 = 15 \times 212 = 61440$ bit patterns
- 14 instructions with 2 addresses
  - $14 \times 2^4 \times 2^4 = 14 \times 2^8 = 3584$ bit patterns
- 31 instructions with 1 address
  - $31 \times 2^4 = 496$ bit patterns
- 16 instructions with 0 addresses
  - The last 16 instructions account for 16-bit patterns

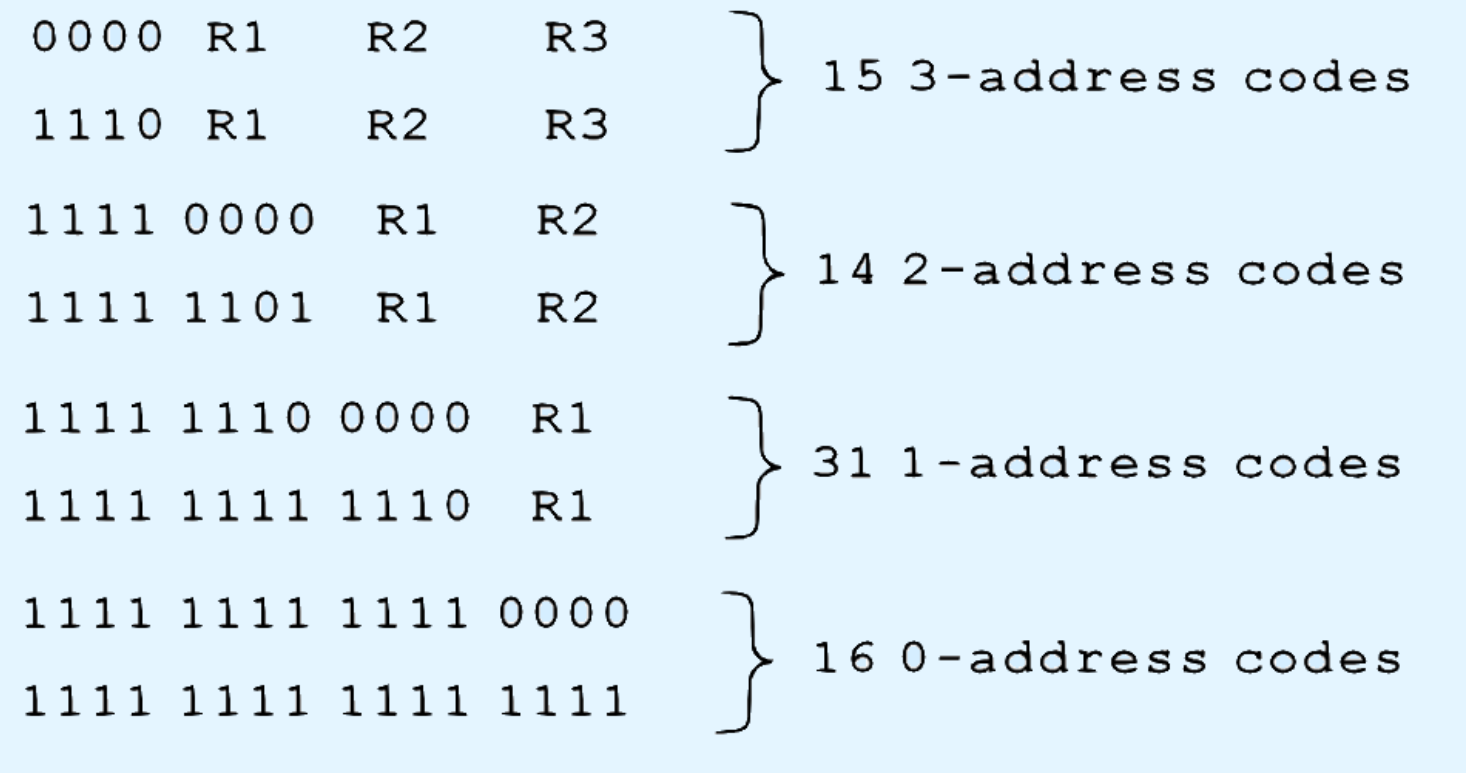
- In total we need 61440 + 3584 + 496 + 16 = 65536 different bit patterns

We have an exact match with no wasted patterns. So our instruction set is possible.

Opcode/Instruction Encoding

Expanding opcode technique

- Consider a machine with 16-bit instructions and 16 registers. We wish to encode the following instructions:
  - ✓ 15 instructions with 3 addresses; 14 instructions with 2 addresses
  - ✓ 31 instructions with 1 address; 16 instructions with 0 addresses



Opcode/Instruction Encoding

Expanding opcode technique

- Is it possible to design an expanding opcode to allow the following to be encoded with a 12-bit instruction? Assume a register operand requires 3 bits.
  - ✓ 4 instructions with 3 registers
  - ✓ 255 instructions with 1 register
  - ✓ 16 instructions with 0 register

- **Solution:**
  - The first 4 instructions account for: 4x23x23x23= 4 x 29= 2048 bit patterns
  - The next 255 instructions account for: 255 x 23= 2040 bit patterns
  - The last 16 instructions account for: 16 bit patterns

  - In total we need 2048 + 2040 + 16 = 4104 bit patterns
  - With 12-bit instruction, we can only have 2^{12} = 4096 bit patterns

Required bit patterns (4104) is more than what we have (4096), so this instruction set is not possible with only 12 bits.

Opcode/Instruction Encoding

Huffman encoding technique

- The block code technique assumes all instructions are used with equal probability
- In practice, not all instructions are used with the same relative frequency count
- On an average, 40% of the instructions used in a program are load and store instructions
- It is required an encoding scheme that will encode the opcode of the ***most frequently used instructions with fewer bits*** and the ***least frequently used instructions with more bits***
- This allows the average number of bits required to encode a typical program to be optimal
- **Huffman coding techniques is used to encode instructions with variable bit lengths**
- It uses variable length encoding.
- It assigns variable length code to all the instructions.
- The code length of an instruction depends on how frequently it occurs in the given text.
- The instruction which occurs most frequently encoded with fewer bits.
- The instruction which occurs least frequently encoded with more bits

Opcode/Instruction Encoding

Building Huffman tree

1. Sort the instructions in ascending order of frequency.
2. Create a leaf node for every unique instruction and its frequency from the given table of instruction set.
3. Extract the two minimum frequency nodes and the sum of these frequencies is made the new root of the tree.
4. Make the first extracted node its left child and the other extracted node as its right child.
5. Repeat **steps 3 and 4** until we left with only one node. The remaining node is the root node and the tree is complete.

Opcode/Instruction Encoding

Building Huffman tree

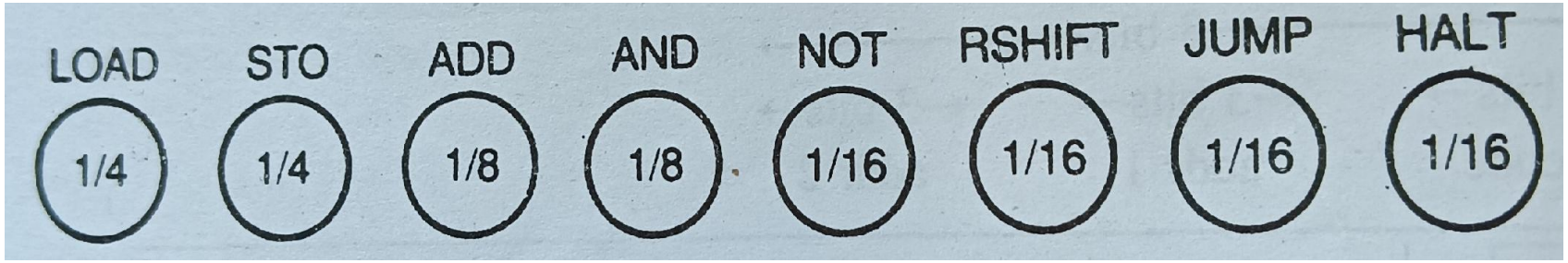
- **Encode the following hypothetical instructions using huffman coding technique**

Instruction	Probability
LOAD	1/4
STO	1/4
ADD	1/8
AND	1/8
NOT	1/16
RSHIFT	1/16
JUMP	1/16
HALT	1/16

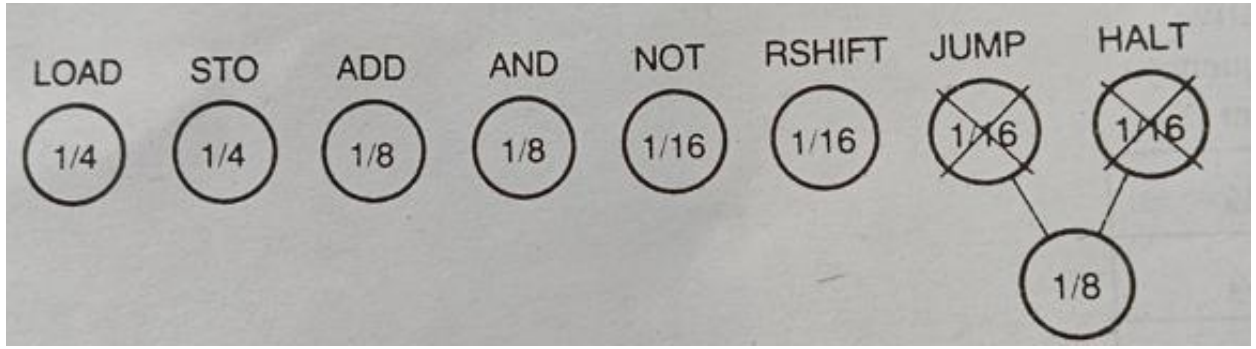
Opcode/Instruction Encoding

Building Huffman tree

- **Arrange the instructions in the ascending order**



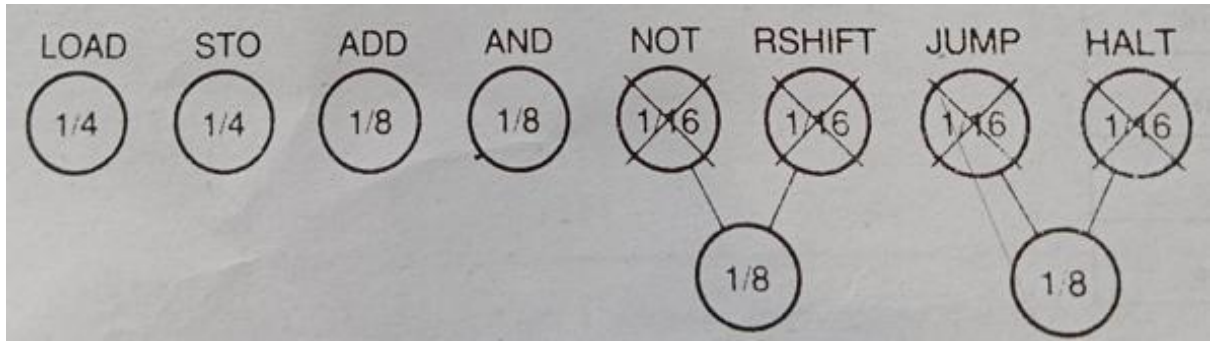
- Select the leaf nodes with least probability and create a new node and assign the weight



Opcode/Instruction Encoding

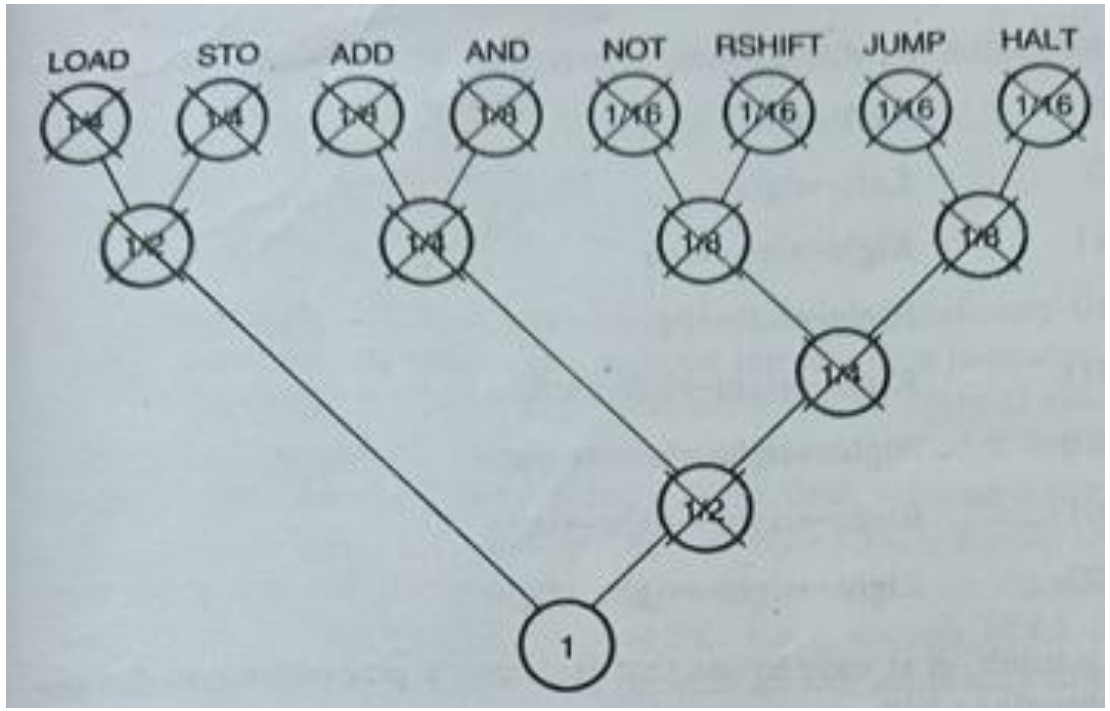
Building Huffman tree

- Select the next two leaf nodes with least probability and create a new node and assign the weight



Building Huffman tree

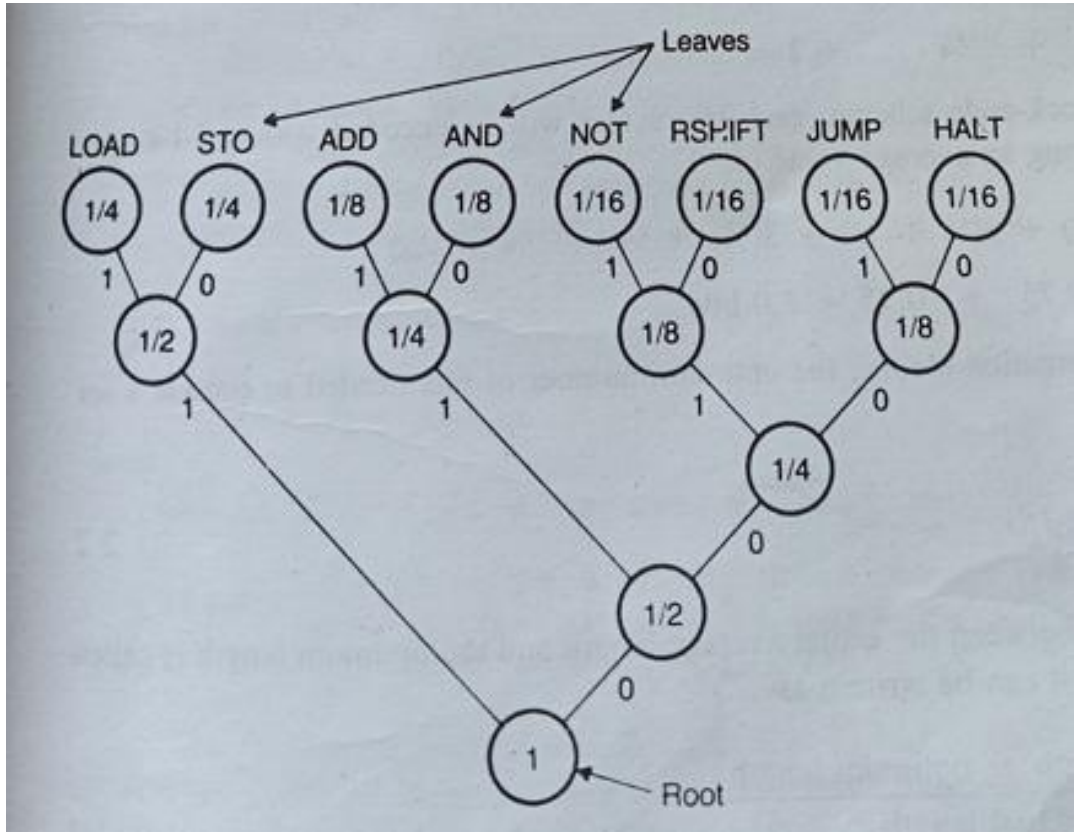
- Continue till only one node left



Opcode/Instruction Encoding

Building Huffman tree

- Label the left and right branches of the tree
- **Assign '1' to the left branches**
- **Assign '0' to the right branches**



Building Huffman tree

- Determining the opcode
- **Path from the root node to the desired leaf node will give you the opcode for the instruction**

MNEMONIC	OP-CODE	PATH FROM THE ROOT
LOAD	11	Left→left
STO	10	Left→right
ADD	011	Right→left→left
AND	010	Right→left→right
NOT	0011	Right→right→left→left
RSHIFT	0010	Right→right→left→right
JUMP	0001	Right→right→right→left
HALT	0000	Right→right→right→right

Opcode/Instruction Encoding

Huffman encoding

- Average number of bits per instructions using Huffman technique

$$Average\ no.of\ bits_{Huffman} = \sum_{i=1}^{n} l_i f_i = 2.75 \text{ bits}$$

- Where l_i is opcode length and f_i relative frequency of the ith instruction

- Average number of bits per instructions using Block code technique

$$Average\ no.of\ bits_{Block\ code} = \sum_{i=1}^{n} l_i f_i = 3.0 \text{ bits}$$

- Optimum number of bits needed to encode

$$Optimum\ length = -\sum_{i=1}^{n} f_i log_2(f_i) = 2.75 \text{ bits}$$

- Redundancy bits calculated as

$$R = \frac{actual\ length - optimum\ length}{actual\ length}$$

$$R_{Huffman} = 0 \quad \text{and} \quad R_{Block\ code} = \frac{1}{12} or\ 8.33\%$$

Opcode/Instruction Encoding

Exercise:

- The hypothetical instruction set of a computer consists of eight instructions, I_0, I_1, I_2,,,,,,I_7. The relative frequency of these instructions are as follows:

Instruction	Relative frequency
I_0	1/4
I_1	1/8
I_2	1/8
I_3	1/8
I_4	1/8
I_5	1/8
I_6	1/16
I_7	1/16

- Encode these instructions using Huffman's method
- Calculate the redundancy introduced by Huffman's and Block code scheme