UNIT 1 contd..

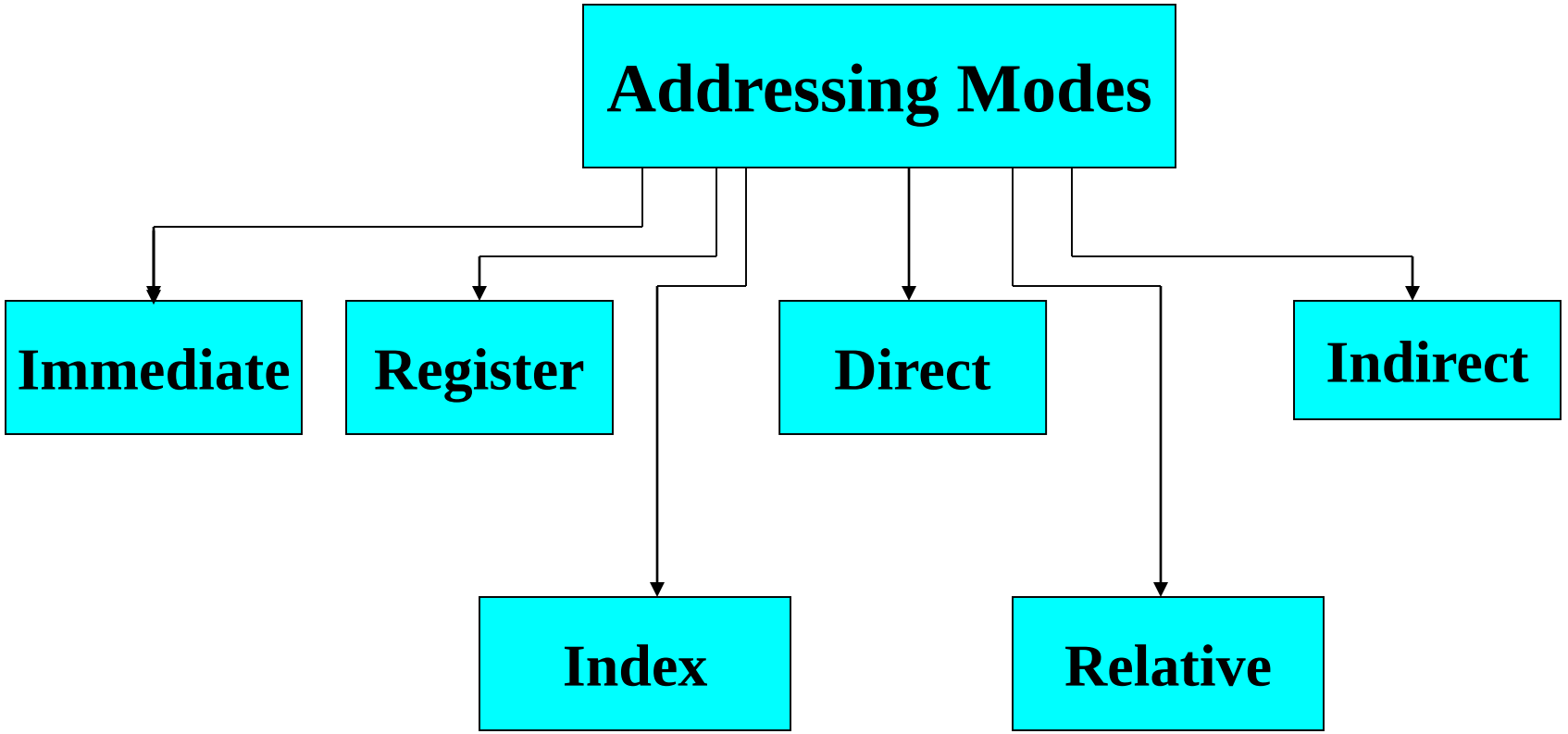
Machine Instructions And Programs

Various Types of Addressing Modes

Introduction

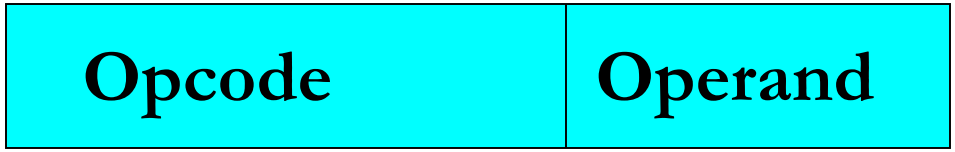
- bit, nibble, byte, word
  - single bit is either 0 or 1. Group of 4 bits is nibble, 8 bit is byte. Collection of n bits is word, n being the word length.
- Data and Address
  - memory of the computer is a collection of words, each memory has a distinct name or address.
- Addressing Modes
  - are the different ways in which the location of an operand (data value) is specified in an instruction

Types of Addressing Modes



Immediate Addressing Mode

Instruction



- Operand is given explicitly in the instruction
- Opcode is followed by operand rather than an address

Example

Move $50_{immediate}$, R0

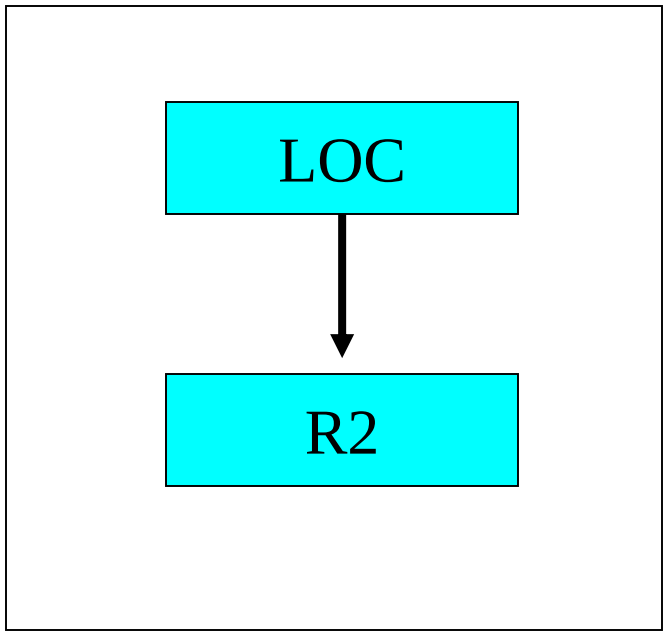
Places data item 50 in register R0

Contd…

- This mode is only used to indicate the value of the source operand
- Immediate operand is indicated by preceding the data by a #
  - **Move #50, R0**
- **Adv:**
  - Only one memory fetch cycle is enough to access both operand & opcode
  - Useful in initializing variables & registers
- **Disadv:**
  - Size of the immediate operand is restricted to the size of address field & opcode size of that instruction

Register Addressing Mode

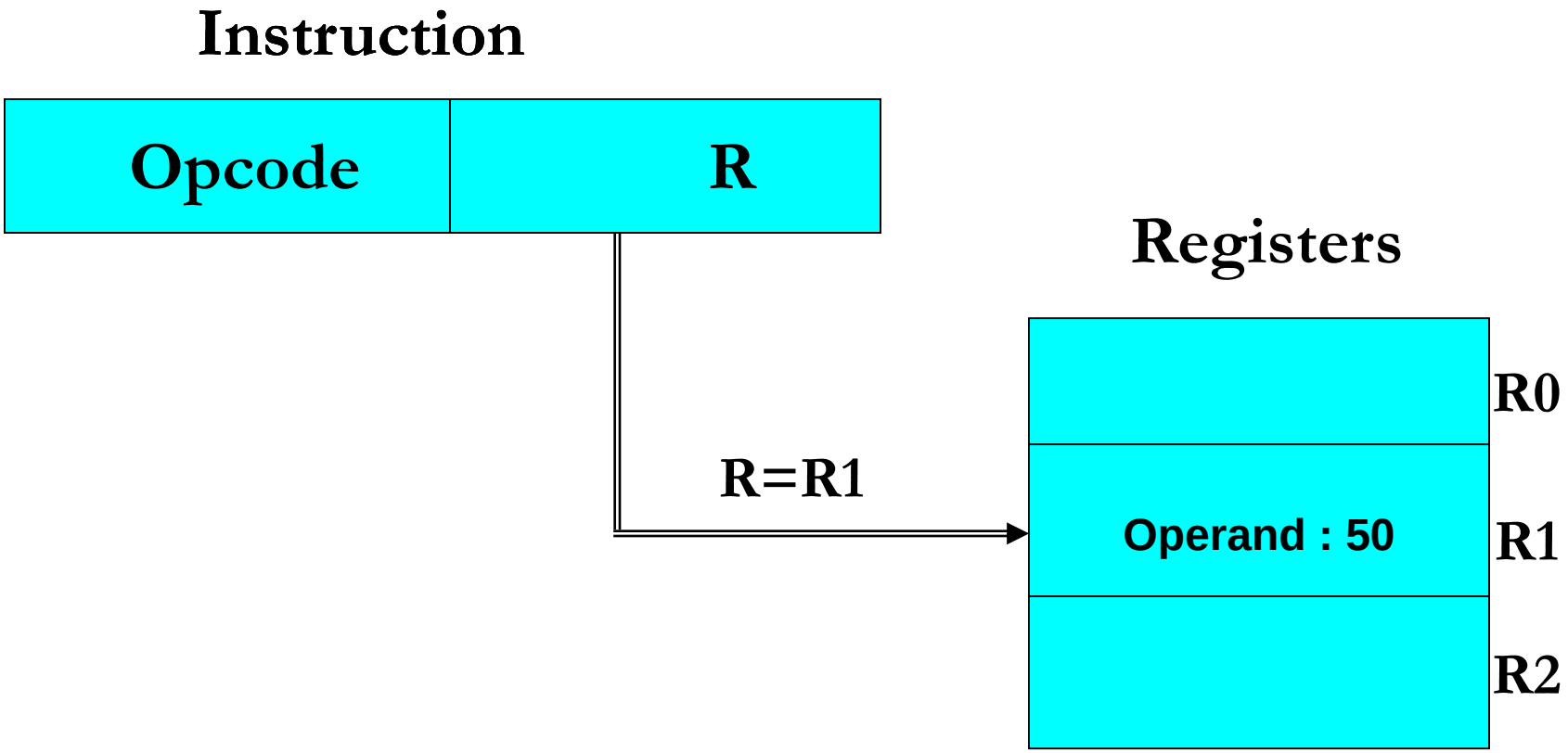
The operand is the contents of a processor register; the name (address) of the register is given in the instruction.



Instruction

Move LOC,R2

RegisterAddressing Mode



R is a general purpose register

Contd..

Advantages:

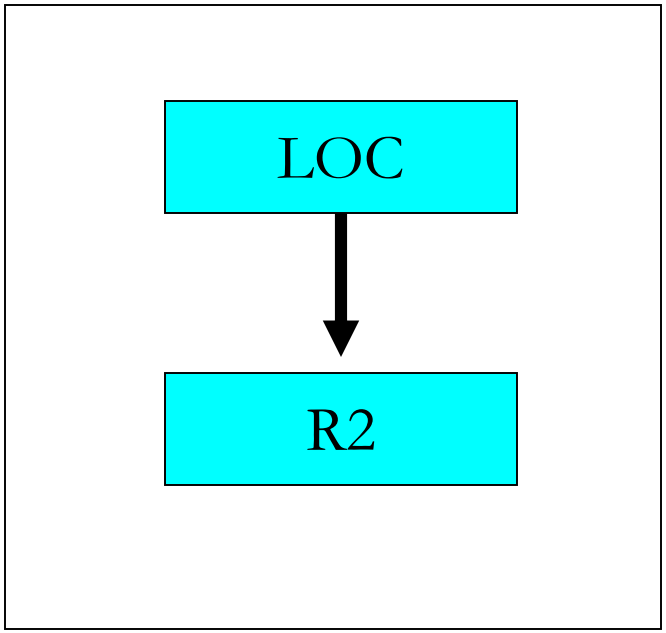
- No memory reference is required
- Shorter address field
- Speedy execution of instruction

Disadvantages:

- Limited registers - Limited address space

Direct Addressing Mode

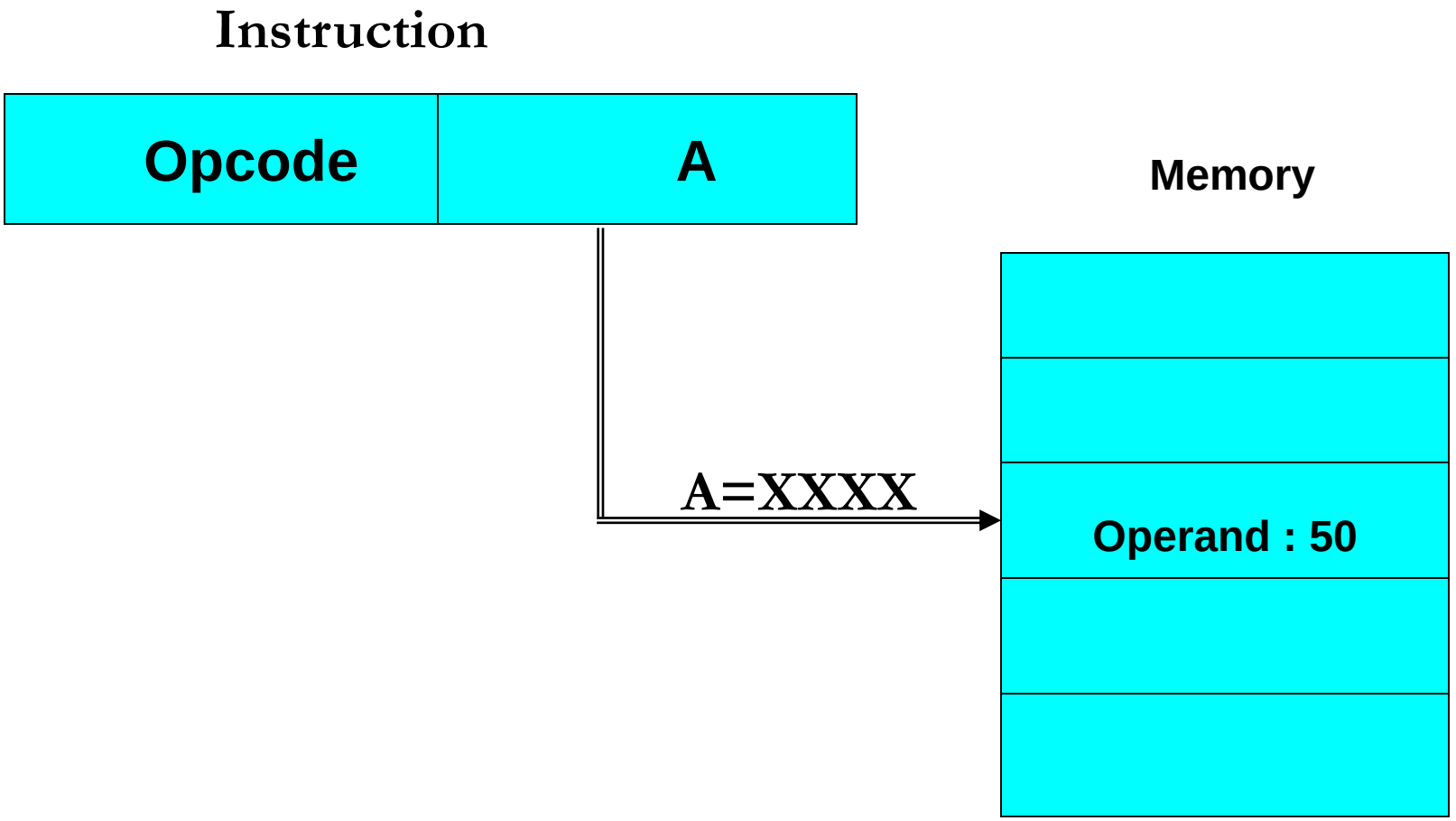
The operand is in a memory location; the address of this location is given explicitly in the instruction. (This mode is also called as *Absolute Addressing Mode*).



Instruction

Move LOC,R2

Direct/Absolute Addressing Mode



A = Address field

Contd..

Advantage:

- Implementation requires less hardware

Indirect Addressing Mode

- The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

- **Effective address** : is the address of the operand obtained from the computation dictated by the given addressing mode.

Contd..

- Address field of the instruction gives the address of a memory word in which effective or actual address of the operand is found

- Accessing an operand requires minimum of two memory fetches
  - Fetch the effective address of the operand
  - Fetch the operand from the effective address location

Contd..

- Indirect addressing mode is symbolically represented as (A)
- Effective address is calculated as EA=[A]
- The register or memory location inside the bracket gives the effective address of the operand.

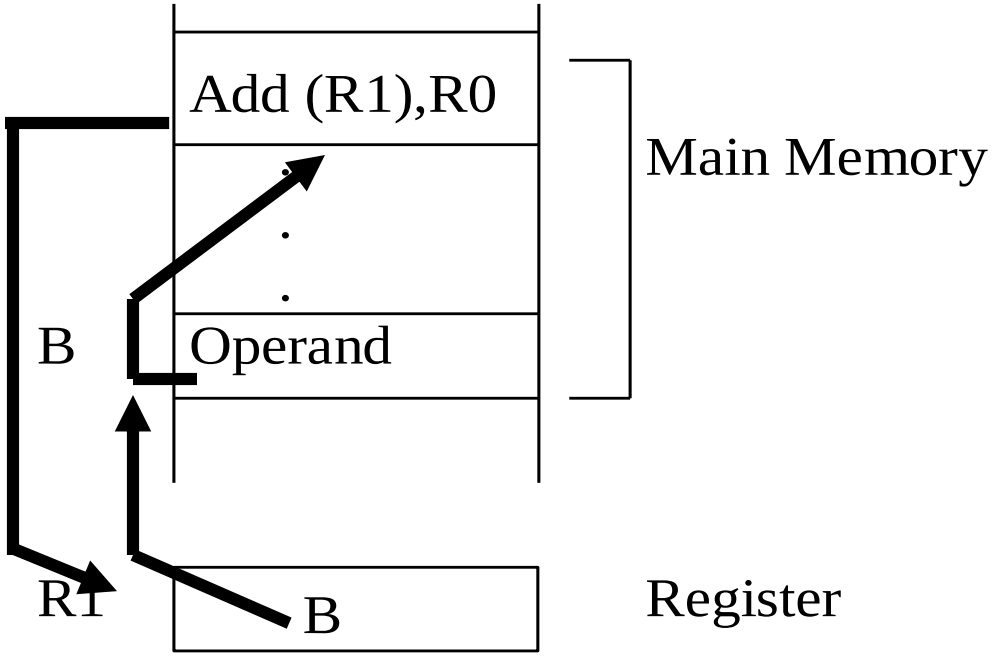
- The memory location or register containing the effective address of the operand is called as **Pointer**.
  - A is a pointer in the above example

Contd…

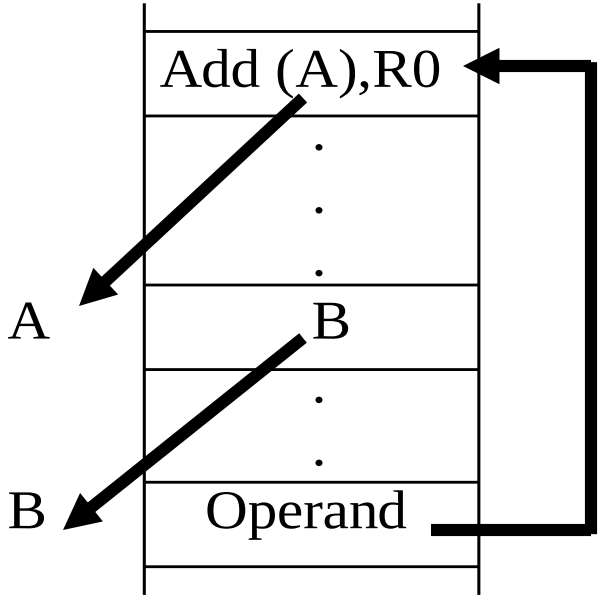
- Two types of indirect addr. modes :
- Through a general purpose register
  - ex: Add (R1), R0
  - Register R1 contains the effective address

- Through a memory location
  - ex: Add (A), R0
  - Memory location A contains the effective address of the operand

Indirect Addressing Mode



(a) Through general purpose register



(b) Through a memory location

Contd..

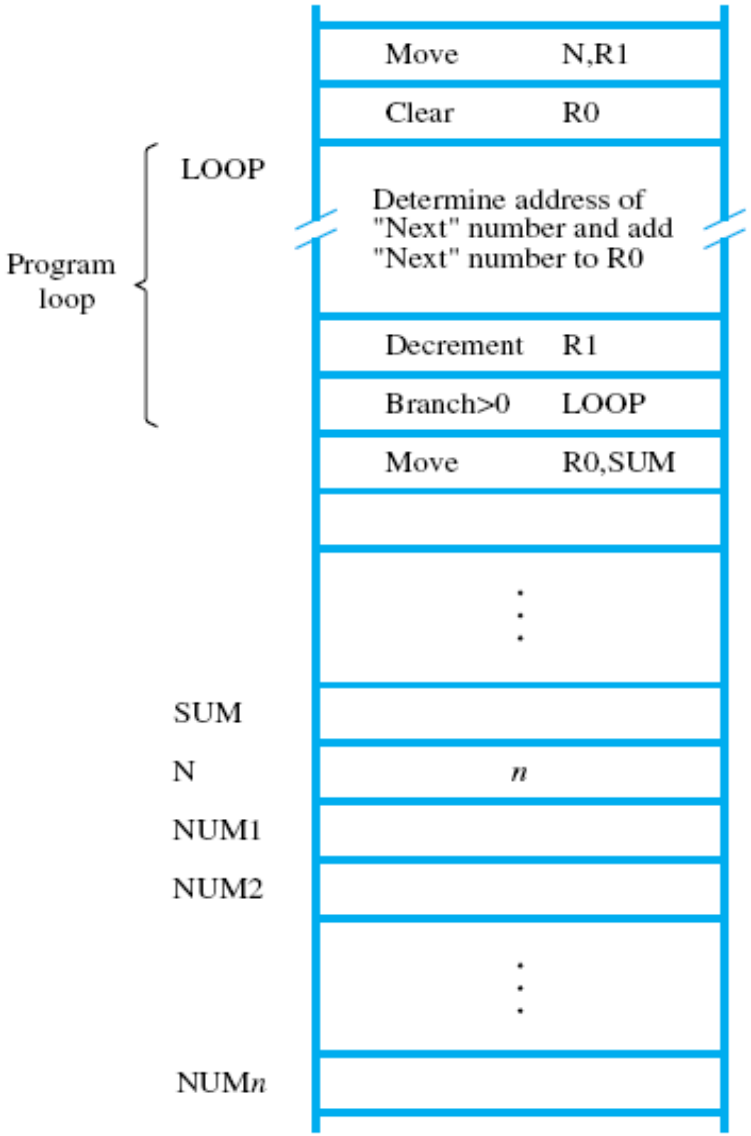
Advantage:

- Wider address range to refer to a large number of memory locations

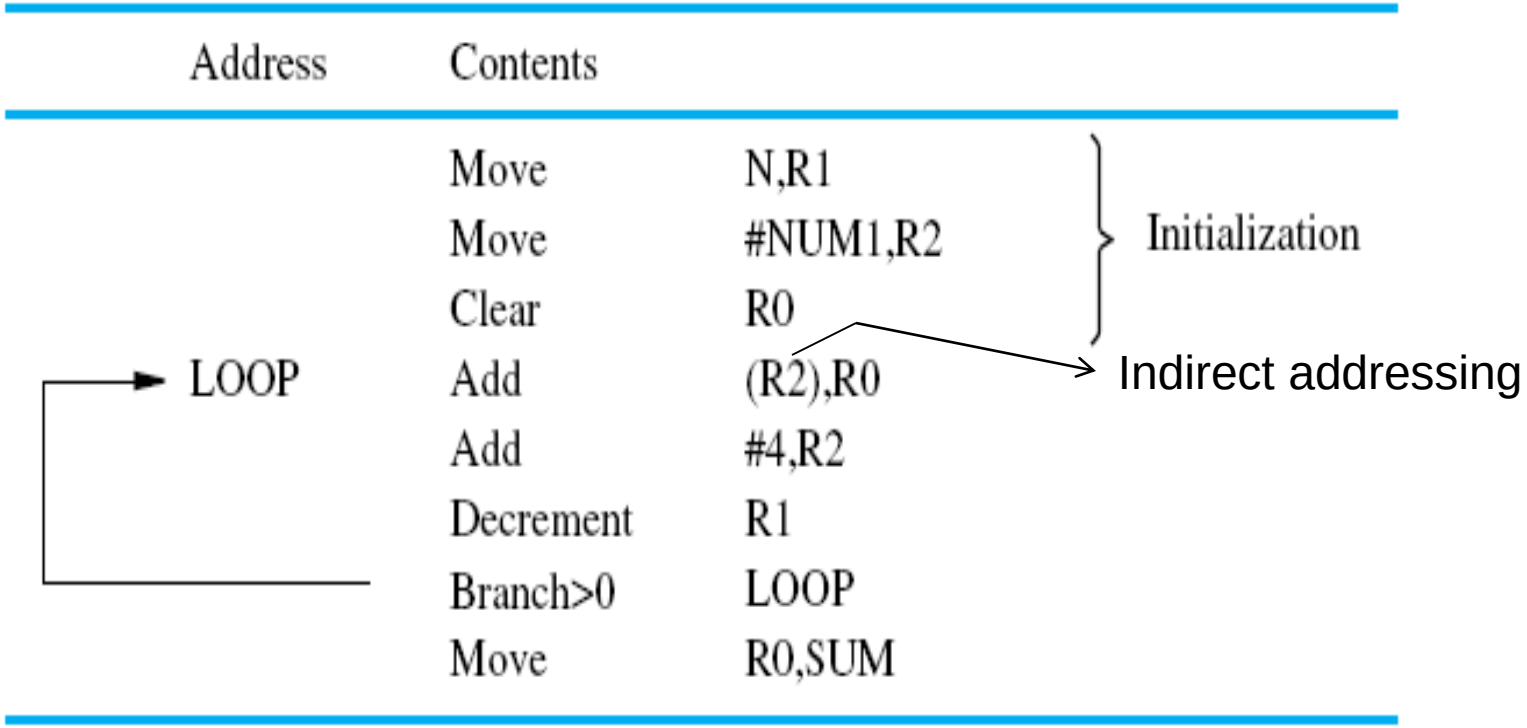
Disadvantage:

- 3 or more memory references to fetch the desired operand

Example: Pgm to add n numbers



The Pgm using indirect addressing



Indexed Addressing Mode

- The effective address of the operand is generated by adding a constant value to the contents of a register
- Symbolic Representation

  X(Ri)

  - X is a constant value given in the instruction
  - X may be either Offset Value or Beginning Address of the data array in memory
  - Ri is the name of the register involved
  - Ri is also called as **Index Register**

Contd..

- Generation of Effective Address:

$$\mathbf{EA = X + [Ri]}$$

- Contents of Index register is not changed during address generation

Contd..

Two ways of using Indexed Mode :

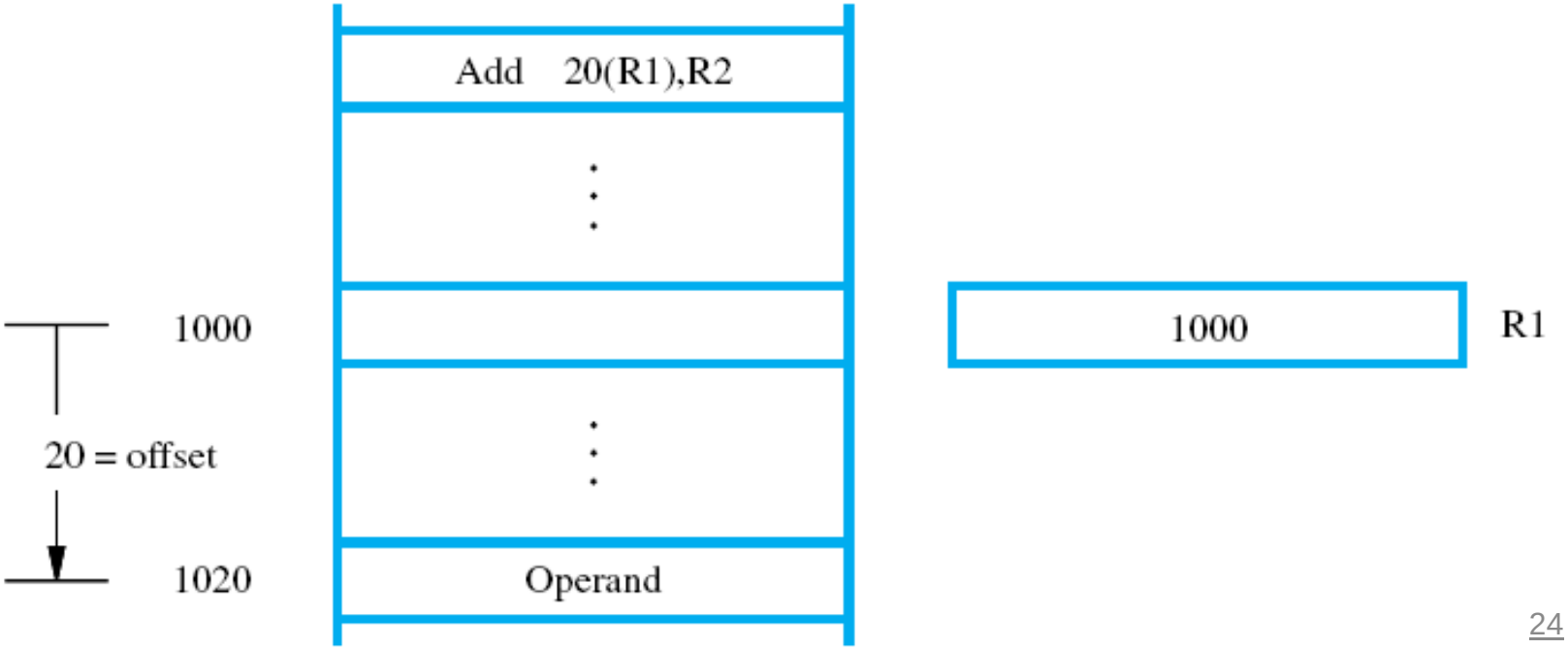
1. Index register Ri contains the beginning address of data array and Value of X defines the offset

2. Index register Ri contains the offset to the operand and Value of X defines the beginning address of data array

Indexed Mode - Type 1

1) Index register Ri contains the beginning address of data array and Value of X defines the offset

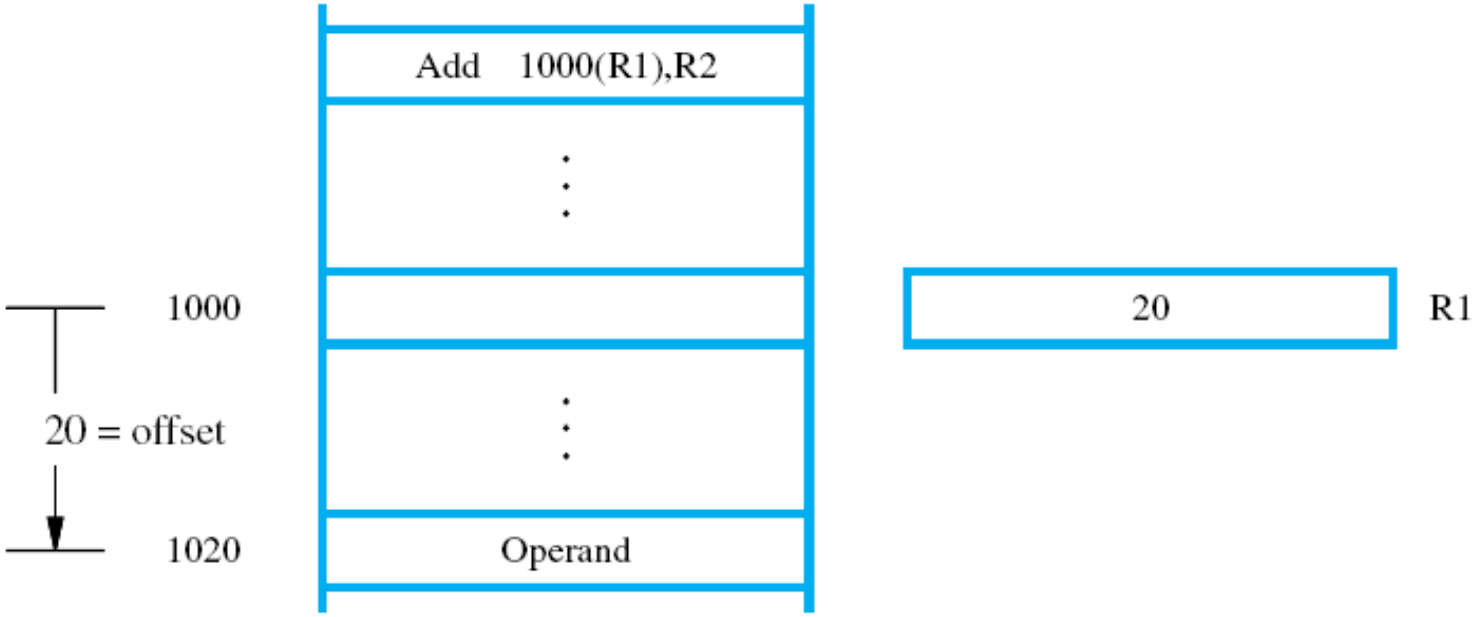
Ex: **Add 20(R1), R2**



Indexed Mode – Type 2

2) Index register Ri contains the offset to the operand and Value of X defines the beginning address of data array

Ex: **Add 1000(R1), R2**



Indexed Mode – contd..

Advantages:

Allows accessing operands whose locations are defined relative to a reference point

Disadvantage :

Complexity involved in computing the effective address

A variant of Indexed mode:

(Ri, Rj)

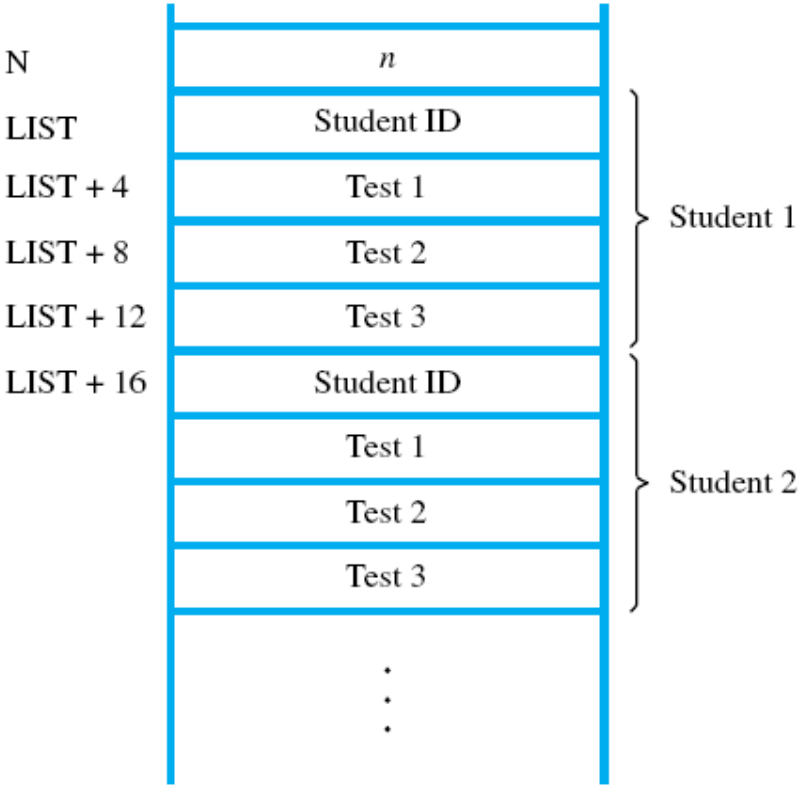
where EA = [Ri] + [Rj]

adv – contents of both registers can be changed

Ex: A pgm using Indexed Addressing

- N student records, starting at location LIST
- test1, test2, test3 – marks of 3 tests
- To find separate sums of marks of each test of all students
- Store these 3 sums at SUM1, SUM2, and SUM3

List of Students' Marks in Memory



The Pgm.

Move #LIST, R0 …… R0 points to ID location of 1st student
Clear R1 … stores the sum of Test 1 marks
Clear R2 … stores the sum of Test 2 marks
Clear R3 … stores the sum of Test 3 marks
Move N, R4 … keeps the count of student records

LOOP **Add 4(R0), R1** …. Acess the scores and
Add 8(R0), R2 …. add them to the running sum
Add 12(R0), R3 …. held in respective registers

Add # 16, R0 ..... Point to the ID location of next student
Decrement R4
Branch>0 LOOP …. Loop until R4 reaches 0

Move R1, Sum1
Move R2, Sum2 …. Store sums from register to memory
Move R3, Sum3

Relative Addressing Mode

- The effective address is determined by the index mode using the Program Counter in place of the general-purpose register Ri.
- Symbolic Representation

  X(PC)

  Addresses a memory location that is X bytes away from the location presently pointed to by the PC

  i.e., **EA = X + [PC]**

- Commonly used to specify the target address in branch instructions

Ex: Computation of branch target address using Relative Addressing mode

Address		Contents
.		Move N, R1
.		Move #NUM1, R2
.		Clear R0
1000	LOOP	Add (R2), R0
1004		Add #4, R2
1008		Decrement R1
1012		Branch>0 LOOP
1016		Move R0, SUM

- Content of PC at the time of branch target address is generated will be 1016
- Compute the offset value X needed to branch to location LOOP (i.e location 1000); i.e, X=-16
- Generate the branch target address as -16(PC)

Additional Modes

- Autoincrement Mode
- Autodecrement Mode
- Useful for accessing data items in successive memory locations

Autoincrement Mode

- Written as: **(Ri)+**
- **Content of the register Ri gives the effective address**
- After accessing the operand, the content of the register is automatically incremented to point to the next item in a list
- The increment is 1 for byte-sized operands, 2 for 16 bit operand & 4 for 32 bit operands

Pgm to add n numbers using-

1.Indirect addr.

```
        Move  N,R1
        Move  #NUM1, R2
        Clear  R0
LOOP    Add  (R2), R0
        Add  #4, R2
        Decrement  R1
        Branch > 0  LOOP
        Move  R0, SUM
```

2.Autoincrement addr.

```
        Move  N,R1
        Move  #NUM1, R2
        Clear  R0
LOOP    Add  (R2)+, R0
        Decrement  R1
        Branch > 0  LOOP
        Move  R0, SUM
```

Autodecrement Mode

- Written as: -**(Ri)**
- Content of the register Ri is **first automatically decremented** which gives the effective address of the operand
- Operands are accessed in descending address order
- Ex: Add –(R2), R0

Addressing Modes- Summary

Name	Assembler Syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Stacks and Queues

Stacks

- A list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only.

- Stack Pointer- points to the top element of the stack
  - It could be one of the general-purpose registers or a register dedicated to this function.

- Top – the last element
- Bottom - the first element
- Operating Mechanism - Last-In-First-Out (LIFO)
- Operations- Push / Pop

Stacks

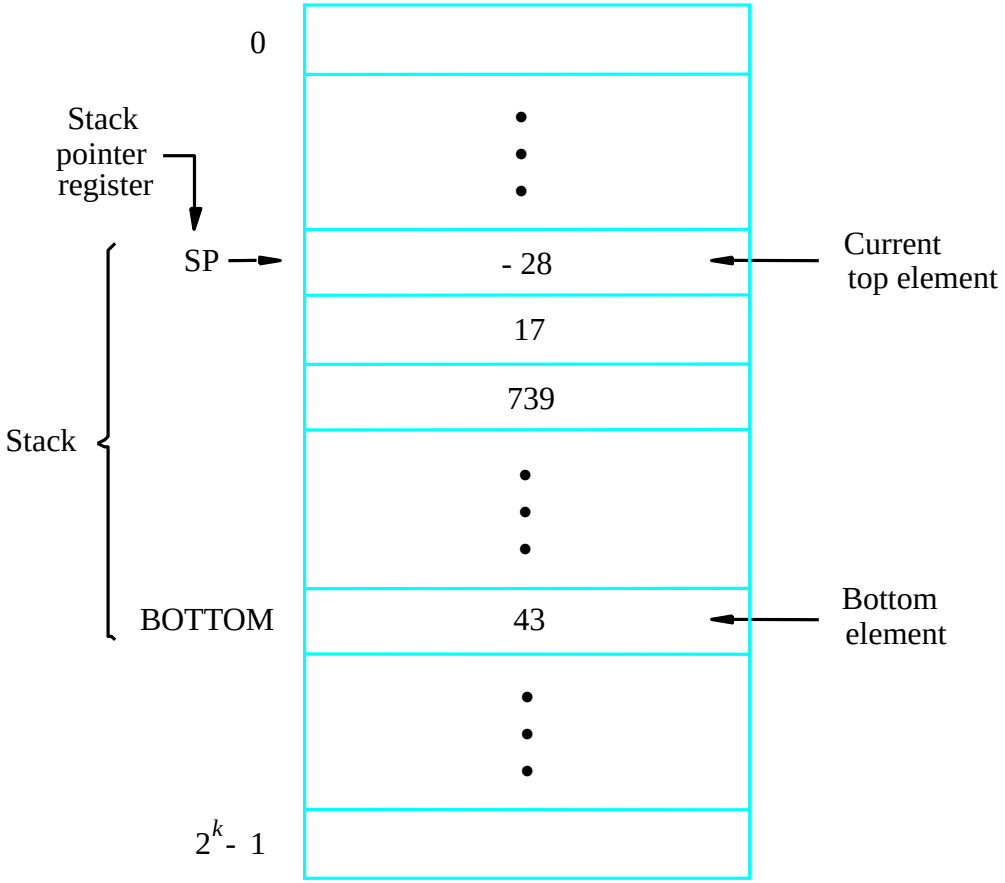


Figure 2.21 A stack of words in the memory.

Stacks

- Push operation on a 32-bit word-length memory

```
Subtract  #4, SP
Move  NEWITEM, (SP)
```

or

```
Move NEWITEM, -(SP)
```

- Pop operation

```
Move  (SP), ITEM
Add  #4, SP
```

or

```
Move (SP)+, ITEM
```

Stacks

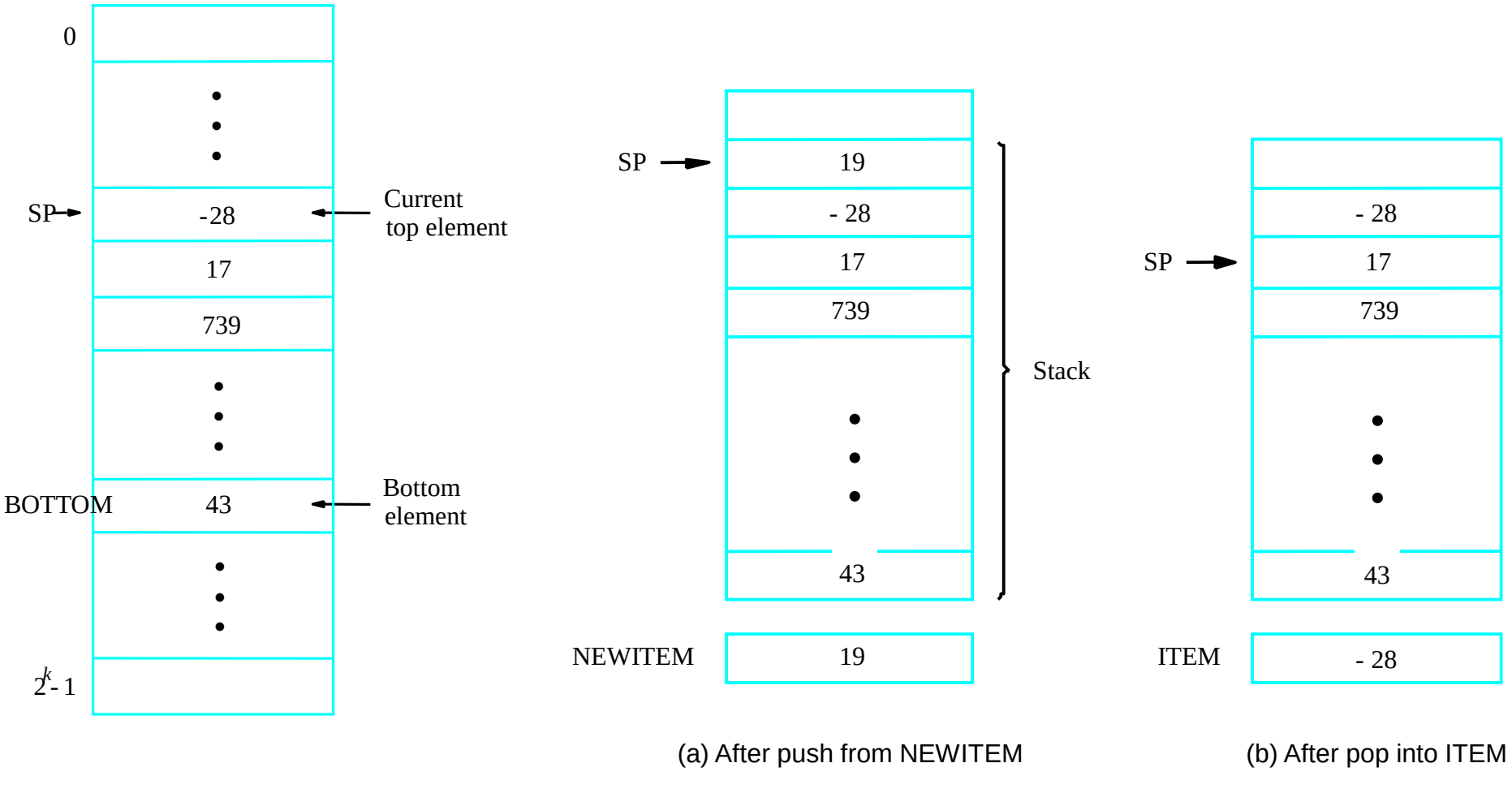


Figure 2.22. Effect of stack operations on the stack in Figure 2.21.

Stacks

- The size of stack in a program is fixed in memory.
- Need to avoid pushing if the maximum size is reached
- Need to avoid popping if stack is empty
- Compare instruction

Compare src, dst

- performs [dst] – [src] and sets the condition code flags according to the result
- Will not change the values of src and dst.

Stacks

SAFEPOP	Compare Branch >0	#2000,SP EMPTYERROR	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare Branch <=0	#1500,SP FULLERROR	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR forappropriate action.
	Move	NEWITEM,–(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

(b) Routine for a safe push operation

Figure 2.23. Checking for empty and full errors in pop and push operations.

Queues

- Data are stored in and retrieved from a queue on a First-In-First-Out (FIFO) basis.

- New data are added at the back (high-address end) and retrieved from the front (low-address end).

- Single pointer is needed for stack (TOS)
- two pointers is needed for queue (front & rear)
- Circular buffer- limits the queue to a fixed region in memory

Subroutines

- It is often necessary to perform a particular subtask (subroutine) many times on different data values.
- To save space, only one copy of the instructions that constitute the subroutine is placed in the memory.
- Any program that requires the use of the subroutine simply branches to its starting location (Call).
- After a subroutine has been executed, it is said to return to the program that called the subroutine, and the program resumes execution. (Return)

Subroutines

- Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location.
- Subroutine Linkage method: use link register to store the PC.

- Call instruction
  - Store the contents of the PC in the link register
  - Branch to the target address specified by the instruction
- Return instruction
  - Branch to the address contained in the link register

Subroutines

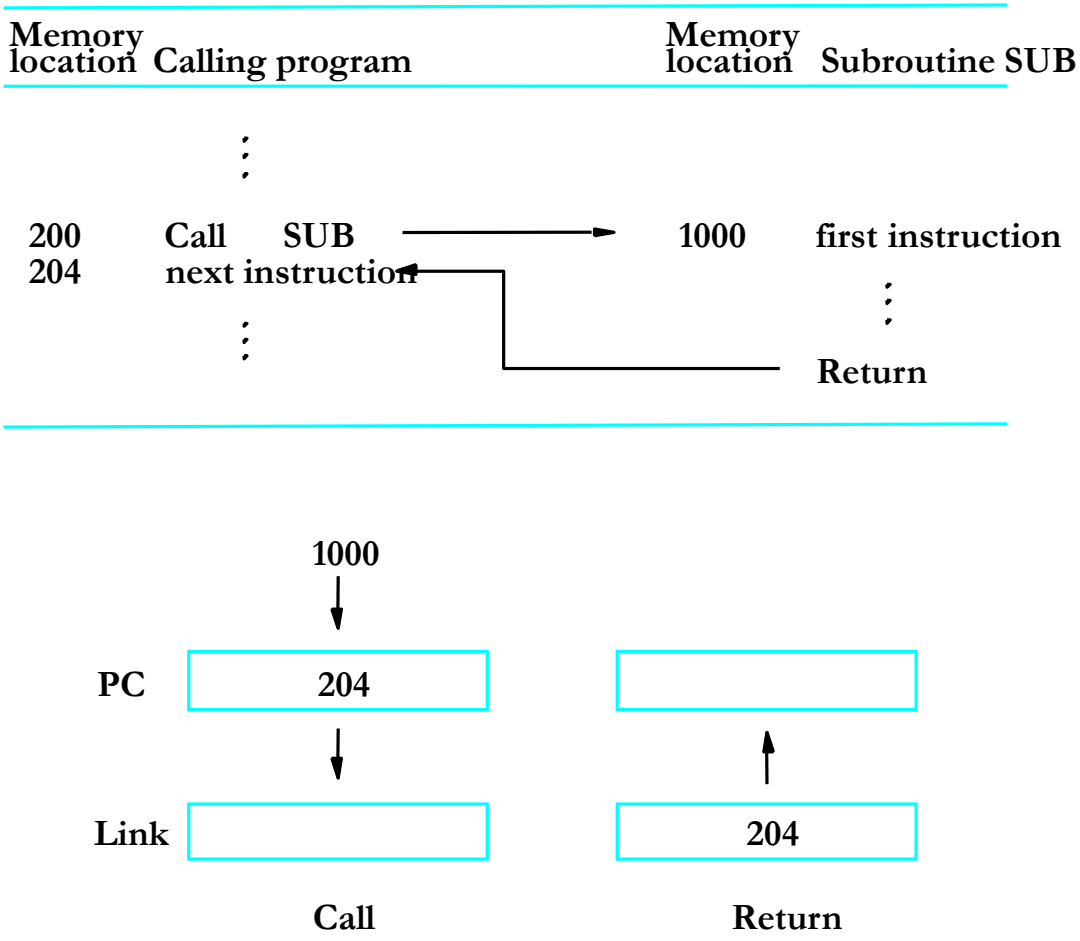


Figure 2.24. Subroutine linkage using a link register.

Subroutine Nesting and The Processor Stack

- If a subroutine calls another subroutine, the contents in the link register will be destroyed.

- If subroutine A calls B, B calls C, after C has been executed, the PC should return to B, then A

- LIFO – Stack
- Automatic process by the Call instruction
- Processor stack

Parameter Passing

- Exchange of information between a calling program and a subroutine.

- Several ways:
  - Through registers
  - Through memory locations
  - Through stack

Passing Parameters through Processor Registers

Calling program

```
            Move      N,R1          R1 serves as a counter.
            Move      #NUM1,R2      R2 points to the list.
            Call      LISTADD       Call subroutine.
            Move      R0,SUM        Save result.
            ⋮
```

Subroutine

```
LISTADD     Clear     R0            Initialize sum to 0.
LOOP        Add       (R2)+,R0      Add entry from list.
            Decrement R1
            Branch>0  LOOP
            Return                  Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

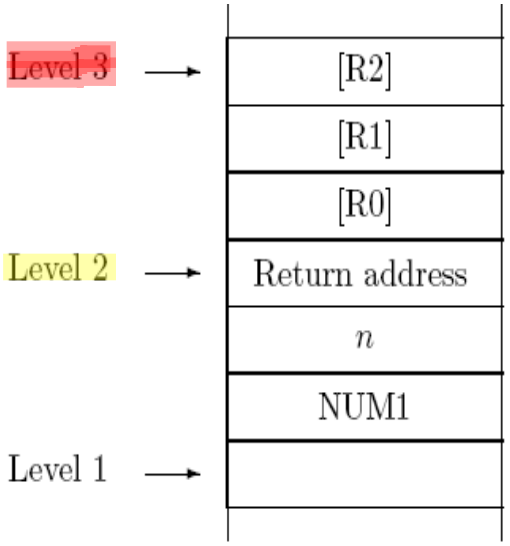
Passing Parameters through Stack

Assume top of stack is at level 1 below.

```
                Move        #NUM1,−(SP)     Push parameters onto stack.
                Move        N,−(SP)
                Call        LISTADD         Call subroutine
                                              (top of stack at level 2).
                Move        4(SP),SUM       Save result.
                Add         #8,SP           Restore top of stack
                                              (top of stack at level 1).
                ⋮

LISTADD         MoveMultiple R0−R2,−(SP)    Save registers
                                              (top of stack at level 3).
                Move        16(SP),R1       Initialize counter to n.
                Move        20(SP),R2       Initialize pointer to the list.
                Clear       R0              Initialize sum to 0.
LOOP            Add         (R2)+,R0        Add entry from list.
                Decrement   R1
                Branch>0    LOOP
                Move        R0,20(SP)       Put result on the stack.
                MoveMultiple (SP)+,R0−R2    Restore registers.
                Return                      Return to calling program.
```

(a) Calling program and subroutine



(b) Top of stack at various times

Figure 2.26 Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

Thank you

Additional Slides (not in the syllabus)

The Stack Frame

- Some stack locations constitute a private work space for a subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a stack frame.

- Frame pointer (FP) – used to access the parameters passed to the subroutine and the local memory variables used by the subroutine

The Stack Frame

- FP register points to the location just above the stored return address
- Use Index addressing to access data inside frame
  -4(FP), 8(FP), …
- Unlike SP, the contents of FP remain fixed throughout the execution of the subroutine

The Stack Frame – when a subroutine is called

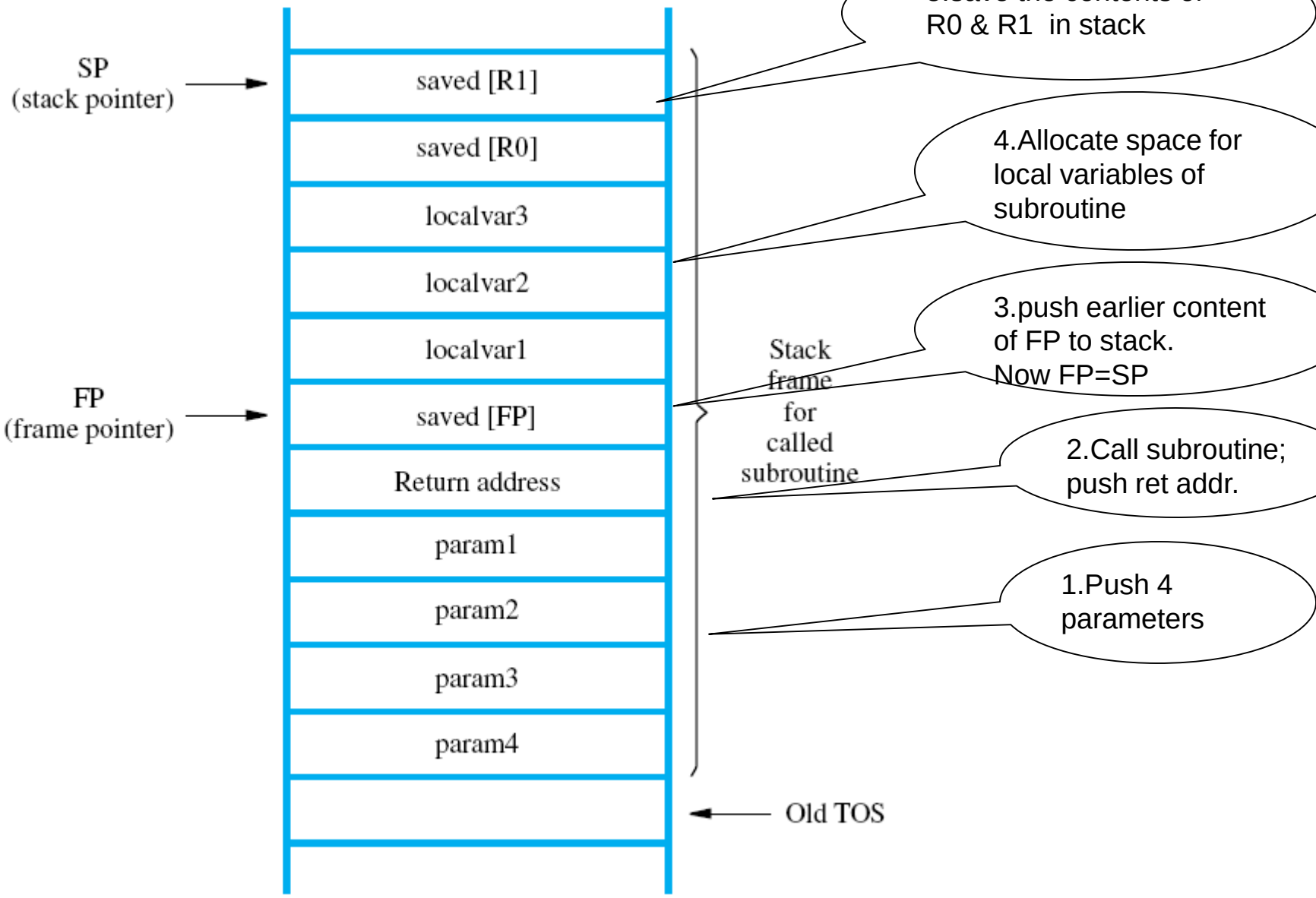


Figure 2.27 A subroutine stack frame example.

The Stack Frame – when a subroutine is completed

When the task is completed,

1. the subroutine pops the saved values of R1 and R0 back into those registers : Movemultiple (SP)+, R0-R1

2. removes the local variables from the stack frame by executing the instruction Add #12,SP

3. pops the saved old value of FP back into FP.

4. At this point, SP points to the return address, transferring control back to the calling program.

SP and FP

When sub is called:

Step 3

Move FP, -(SP)

Move SP, FP

Step 4

Subtract #12, SP

Step 5

Push R0 and R1

On completion of sub:

1. Pop R0 and R1
2. Add #12, SP
3. Move (SP)+, FP
4. Return

Stack Frames for Nested Subroutines

Memory location		Instructions		Comments
Main program				
		⋮		
2000		Move	PARAM2, – (SP)	Place parameters on stack.
2004		Move	PARAM1, – (SP)	
2008		Call	SUB1	
2012		Move	(SP),RESULT	Store result.
2016		Add	#8,SP	Restore stack level.
2020		next instruction		
		⋮		
First subroutine				
2100	SUB1	Move	FP,– (SP)	Save frame pointer register.
2104		Move	SP,FP	Load the frame pointer.
2108		MoveMultiple	R0 – R3, – (SP)	Save registers.
2112		Move	8(FP),R0	Get first parameter.
		Move	12(FP),R1	Get second parameter.
		⋮		
		Move	PARAM3, – (SP)	Place aparameter on stack.
2160		Call	SUB2	
2164		Move	(SP)+,R2	Pop SUB2 result into R2.
		⋮		
		Move	R3,8(FP)	Place answer on stack.
		MoveMultiple	(SP)+,R0 – R3	Restore registers.
		Move	(SP)+,FP	Restore frame pointer register.
		Return		Return to Main program.
Second subroutine				
3000	SUB2	Move	FP,– (SP)	Save frame pointer register.
		Move	SP,FP	Load the frame pointer.
		MoveMultiple	R0 – R1, – (SP)	Save registers R0 and R1.
		Move	8(FP),R0	Get the parameter.
		⋮		
		Move	R1,8(FP)	Place SUB2 result on stack.
		MoveMultiple	(SP)+,R0 – R1	Restore registers R0 and R1.
		Move	(SP)+,FP	Restore frame pointer register.
		Return		Return to Subroutine 1.

Figure 2.28. Nested subroutines.

Stack Frames for Nested Subroutines

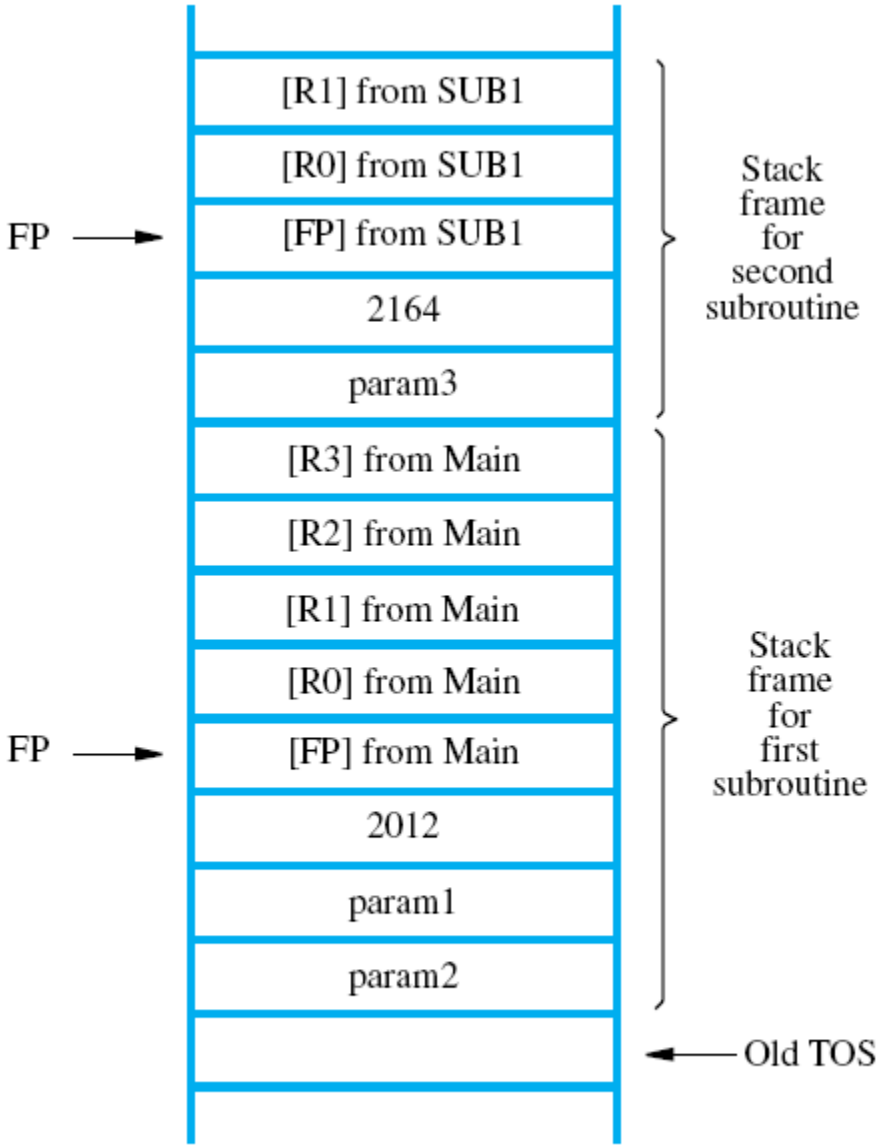


Figure 2.29 Stack frames for Figure 2.28.

Additional Instructions

Logic Instructions

- AND
- OR
- NOT
- (what's the purpose of the following)

```
Not  R0
Add  #1, R0
```

- Determine if the leftmost character of the four ASCII characters stored in the 32-bit register R0 is Z (01011010)

```
And  #$FF000000, R0
Compare  #$5A000000, R0
Branch=0  YES
```

Logical Shifts: shifting left (LShiftL) and shifting right (LShiftR)

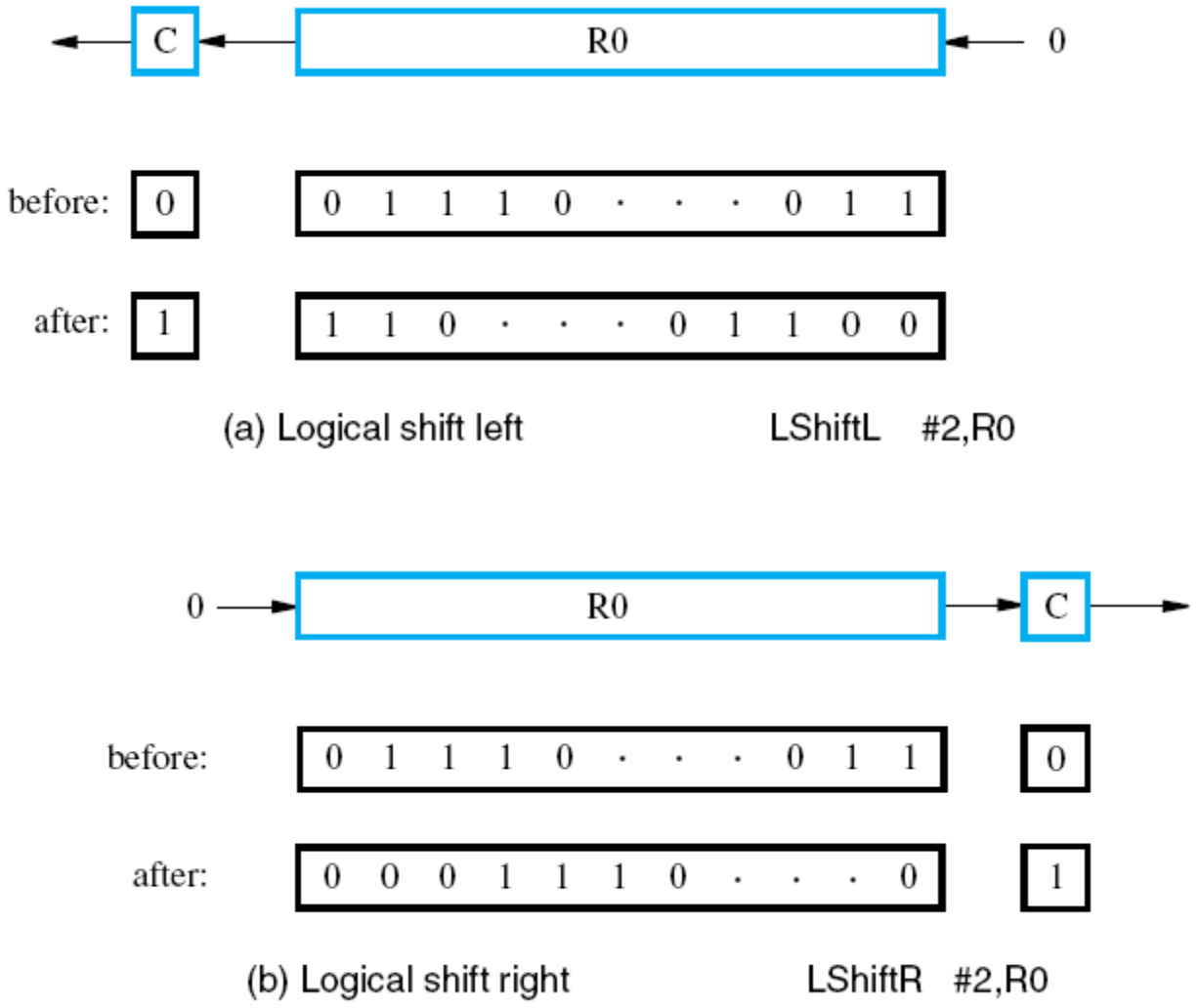


Figure 2.30 Logical and arithmetic shift instructions.

Arithmetic Shifts

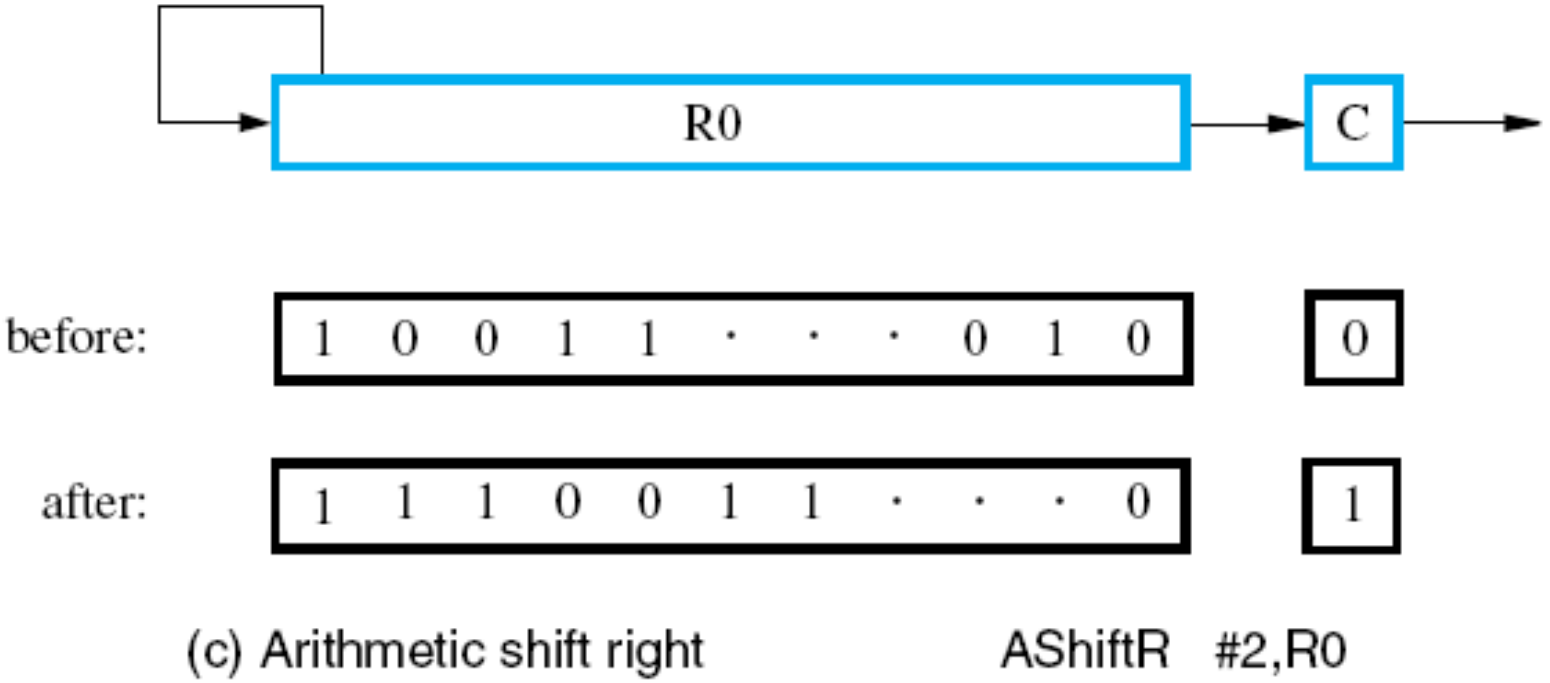
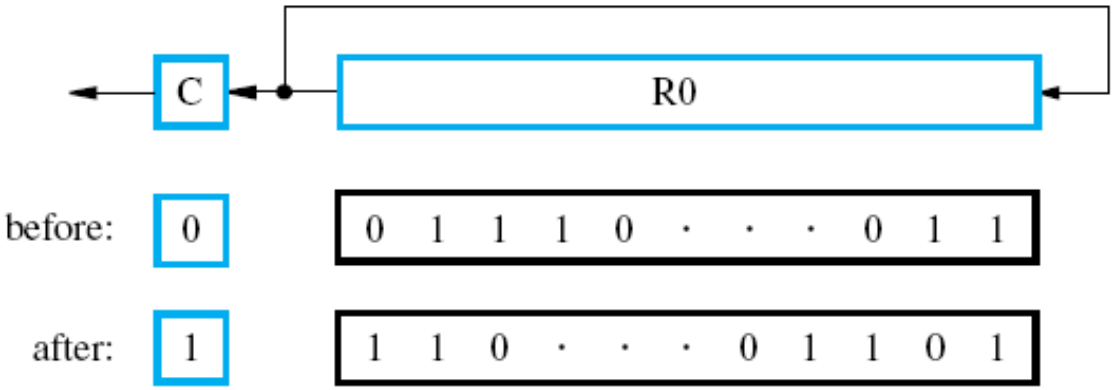
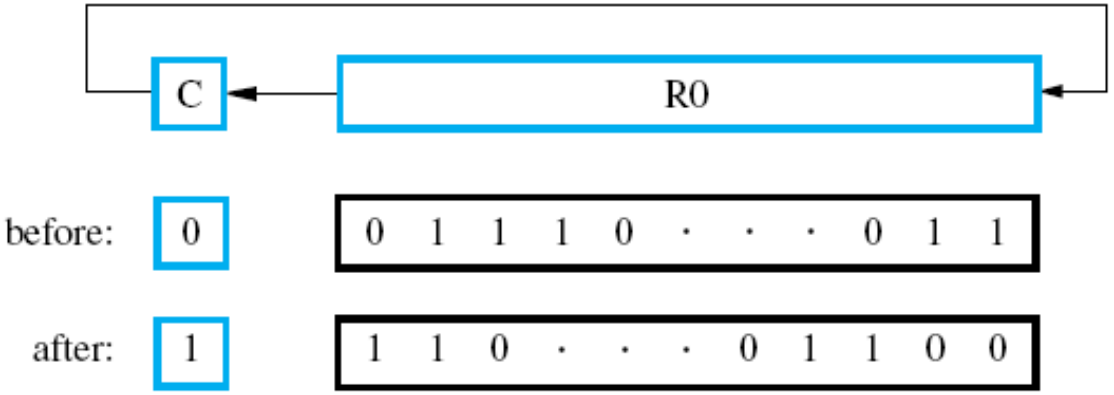


Figure 2.30 Logical and arithmetic shift instructions.

Rotate



(a) Rotate left without carry RotateL #2,R0



(b) Rotate left with carry RotateLC #2,R0

Figure 2.32 Rotate instructions.

Logical Shifts

- Two decimal digits represented in ASCII code are located at LOC and LOC+1. Pack these two digits in a single byte location PACKED.

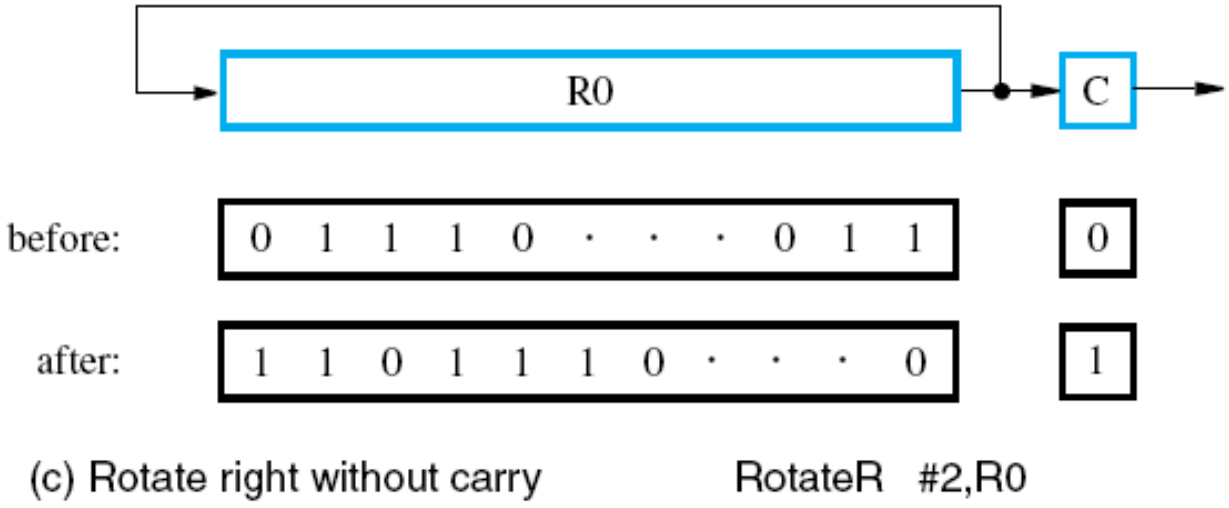
```
Move        #LOC,R0       R0 points to data.
MoveByte    (R0)+,R1      Load first byte into R1.
LShiftL     #4,R1         Shift left by 4 bit positions.
MoveByte    (R0),R2       Load second byte into R2.
And         #$F,R2        Eliminate high-order bits.
Or          R1,R2         Concatenate the BCD digits.
MoveByte    R2,PACKED     Store the result.
```

Figure 2.31 A routine that packs two BCD digits.

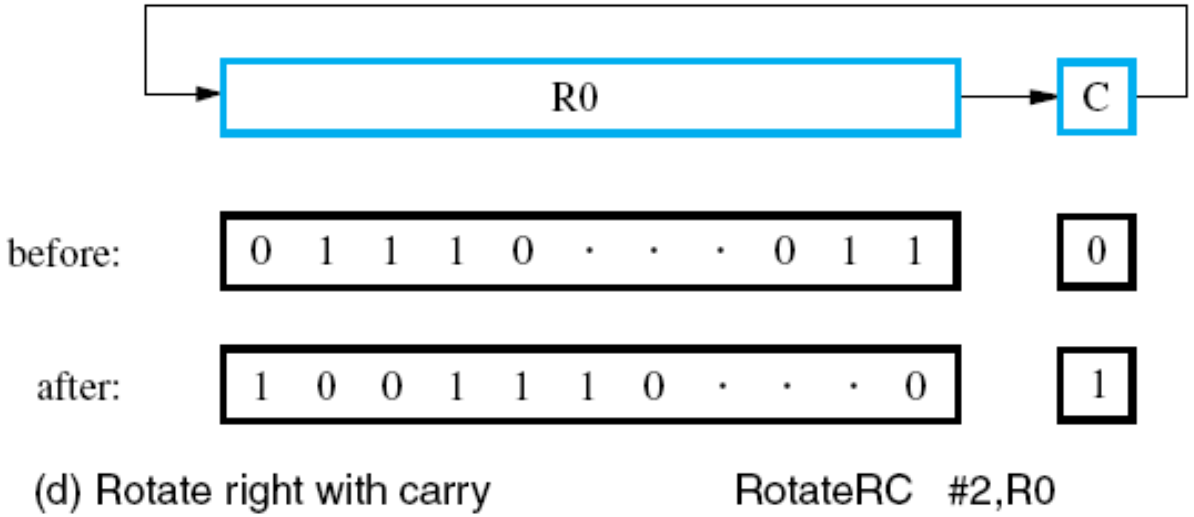
ASCII TABLE

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Rotate



(c) Rotate right without carry RotateR #2,R0



(d) Rotate right with carry RotateRC #2,R0

Figure 2.32 Rotate instructions.

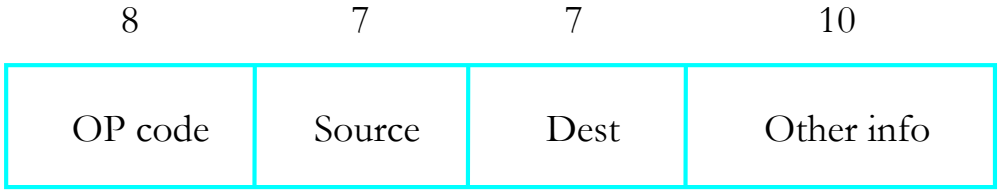
Encoding of Machine Instructions

Encoding of Machine Instructions

- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)

- In the previous section, an assumption was made that all instructions are one word in length.

- Suppose 32-bit word length, 8-bit OP code (how many instructions can we have?), 16 registers in total (how many bits?), 3-bit addressing mode indicator.

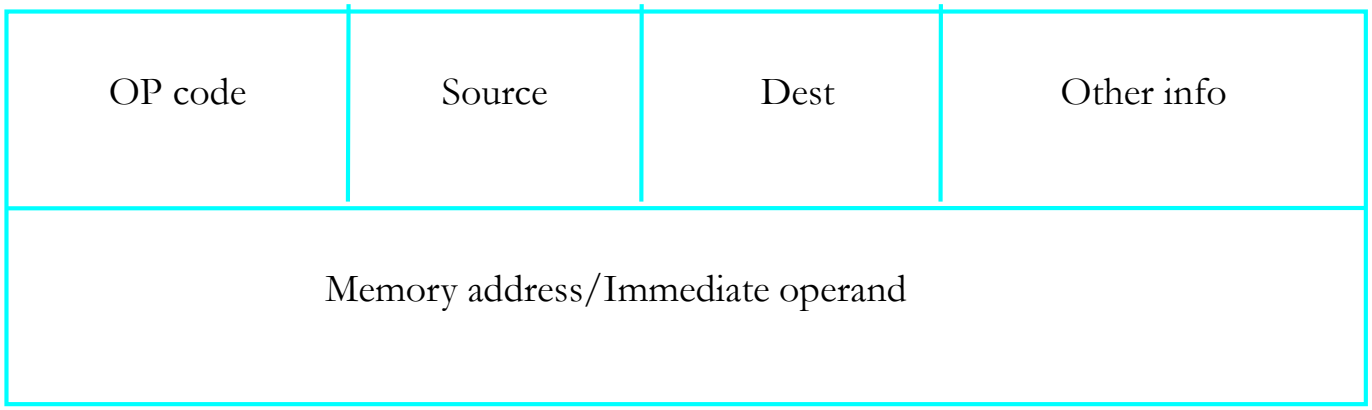


(a) One-word instruction

- Add R1, R2
- Move 24(R0), R5
- LshiftR #2, R0
- Move #$3A, R1
- Branch>0 LOOP

Encoding of Machine Instructions

- What happens if we want to specify a memory operand using the Absolute addressing mode?
- Move R2, LOC
- 14-bit for LOC – insufficient
- Solution – use two words



(b) Two-word instruction

Encoding of Machine Instructions

- Then what if an instruction in which two operands can be specified using the Absolute addressing mode?
- Move LOC1, LOC2
- Solution – use two additional words
- This approach results in instructions of variable length. Complex instructions can be implemented, closely resembling operations in high-level programming languages – Complex Instruction Set Computer (CISC)

Encoding of Machine Instructions

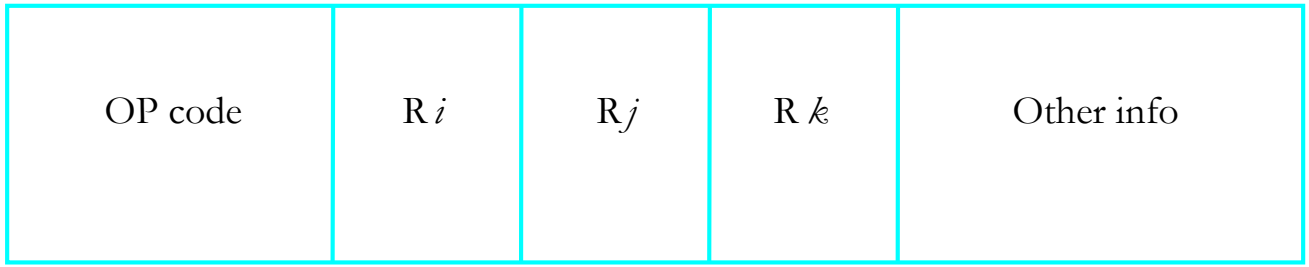
- If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction.
- It is still possible to define a highly functional instruction set, which makes extensive use of the processor registers.
- Add R1, R2 ----- yes
- Add LOC, R2 ----- no
- Add (R3), R2 ----- yes

Encoding of Machine Instructions

- How to load a 32-bit address into a register that serves as a pointer to memory locations?
- Solution #1 – direct the assembler to place the desired address in a word location in a data area close to the program, so that the Relative addressing mode can be used to load the address.
- Solution #2 – use logical and shift instructions to construct the desired 32-bit address by giving it in parts that are small enough to be specifiable using the Immediate addressing mode.

Encoding of Machine Instructions

- An instruction must occupy only one word – Reduced Instruction Set Computer (RISC)
- Other restrictions



(c) Three-operand instruction

Number representation

- For computer arithmetic we use **binary number system.**

- The binary number system is said to be of base 2 or radix 2, because it uses two digits and the coefficients are multiplied by power of 2.

- The binary number 110011 represents the quantity equal to:

  $1x2^5+1x2^4+0x2^3+0x2^2+1x2^1+1x2^0 = 51$ in decimal

Representation of Unsigned Integers

- What is the size of Integer that can be stored in a Computer?
  - It depends on the word size of the Computer
  - Word size indicates the number of bits used to represent the number
  - We call a computer as 8-bit computer if it uses 8 bits to store or represent the number
  - With 8 bits lowest value that can be represented is 00000000 (i.e. 0) , and highest value is 11111111 (i.e. 255)
  - In general, for n-bit number, the range for natural number is from 0 to 2^n-1

Representation of Unsigned Integers

- Consider the addition of the following numbers

10000001	129
10101010	178
----------	------
100101011	307

  - the result is a 9-bit number; but we can store only 8 bits (00101011 =43 which is wrong)
  - Hence this case is known as overflow.

Hexadecimal number

- The hexadecimal number system is said to be of base, or radix 16,
  - because it uses 16 symbols and the coefficients are multiplied by power of 16.
- Sixteen digits used in hexadecimal system are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

- Example:

Binary	Hexadecimal	Decimal
01101000	68	104
00111010	3A	58
----------	------	-----
10100010	A2	162

Hexadecimal number

- There are three different schemes to represent negative number:
  - Signed-Magnitude form.
  - 1’s complement form.
  - 2’s complement form.

Signed magnitude form

- For an n-bit number, one bit (generally, the MSB) is used to indicate the sign of the number and remaining (n-1) bits are used to represent magnitude.

- Therefore, the range is from : $-2^{n-1} - 1$ to $+2^{n-1} - 1$

- 0 in sign bit indicates positive number and 1 in sign bit indicates negative number.

- For example, 01011001 represents + 169 and

  11011001 represents - 169

- What is 00000000 and 10000000 in signed magnitude form?

Representation of Signed integer in 1's complement form

- Consider the eight bit number 01011100,
- 1's complements of this number is 10100011.
- If we perform the following addition:

  0 1 0 1 1 1 0 0

  1 0 1 0 0 0 1 1

  ----

  1 1 1 1 1 1 1 1
- If we add 1 to the number, the result is 100000000.

Representation of Signed integer in 1's complement form

- Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored.
- Therefore, the final result is 00000000.
- Since the addition of two number is 0, so one can be treated as the negative of the other number.
- So, 1's complement can be used to represent negative number.

Representation of Signed integer in 2's complement form

- Consider the eight bit number 01011100,
- 2's complements of this number is 10100100.
- If we perform the following addition:

0 1 0 1 1 1 0 0

1 0 1 0 0 0 1 1

---------- ----------

1 0 0 0 0 0 0 0 0

Representation of Signed integer in 2's complement form

- Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored.
- Therefore, the final result is 00000000.
- Since the addition of two number is 0, so one can be treated as the negative of the other number.
- So, 2's complement can be used to represent negative number.

4-bit numbers in 1's, 2's and signed magnitude form

Decimal	2's Complement	1's complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	-----	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	------	-------

Overflow in Integer arithmetic

- Occurs when the result of an arithmetic operation is outside the representable range

- Range for n bit 2's complement number

$$-2^{n-1} \text{ to } +2^{n-1} - 1$$

if n=4 then

range of values = -8 to +7

Overflow - examples

a) (+7) + (+4)

= +11

2's complement add:

0 1 1 1

0 1 0 0

= 1 0 1 1 = -5

No carry out

b) (-4) +(-6)

= -10

2's complement add:

1 1 0 0

1 0 1 0

=1 ← 0 1 1 0 = +6

Carry out

Overflow - examples

c) (+7) + (-3)

= +4

2's complement add:

0 1 1 1

1 1 0 1

= 1 ← 0 1 0 0 = +4

Carry out

Overflow - Contd

- Overflow occurs when adding two nos. that have **SAME SIGN**

- Carry out signal from Sign bit position is **not a Sufficient indicator** of overflow

To Detect Overflow

- Examine the sign bit of 2 nos, X and Y.
- Examine the sign bit of result, S.
- Overflow occurs if X and Y has same sign and S has different sign.

Representation of Real Number

- Binary representation of 41.6875 is 101001.1011

- There are **two schemes to represent real number :**
  - Fixed-point representation
  - Floating-point representation

Fixed-point representation

- Binary representation of 41.6875 is 101001.1011
- To store this number, we have to store **two information,**
  - the part before decimal point and
  - the part after decimal point.
- This is known as fixed-point representation where the position of decimal point is fixed and number of bits before and after decimal point are also predefined.
- If we use 16 bits before decimal point and 8 bits after decimal point,
  - in signed magnitude form, the range is $-2^{16} - 1$ to $+2^{16} - 1$
  - and the precision is 2^{-8}

Fixed-point representation

- One bit is required for sign information, so the total size of the number is 25 bits

(1(sign) + 16(before decimal point) + 8(after decimal point)).

Floating-point representation:

- In this representation, numbers are represented by a mantissa comprising the significant digits and an exponent part of Radix R.
- The format is: $mantissa * R^{exponent}$
- Numbers are often normalized, such that the decimal point is placed to the right of the **first non zero digit.**

- For example, the decimal number, 5236 is equivalent to
  - 5.236×10^3
  - 5236 is mantissa part and 3 is exponent part.

IEEE standard floating point format

- IEEE has proposed two standard for representing floating-point number:

1. Single precision
2. Double precision

IEEE floating point format

- **Single Precision:**
- S: sign bit: 0 denotes + and 1 denotes –
- E: 8-bit excess -27 exponent
- M: 23-bit mantissa

- **Double precision:**
- S: sign bit: 0 denotes + and 1 denotes –
- E: 11-bit excess -1023 exponent
- M: 52-bit mantissa

Exercise

- Represent the following decimal numbers in IEEE single precision notation
  - -5 =1100 0000 1010 0000 0000 0000 0000 0000
    =C0A00000
  - +8.1 =0100 0001 0000 0001 1001 1001 1001 1001
    =41019999
  - +121.2 = 0100 0010 1111 0010 0110 0110 0110 0110
    =42F26666
  - +2.125 = 0100 0000 1000 1000 0000 0000 0000 0000
    = 40080000

Exercise

- 125.23
- 3.125
- $10.1_{(2)}$
- $101000.100110011001100110011001\ldots_{(2)}$

Exercise

What are the decimal equivalent values of the following IEEE single precision numbers:

- 0X4161ED29

  $=1.11000011110110100101001 \times 2^{130}$

  ≈ 14.1171875

- 0XC083460B

  = 1100 0000 1000 0011 0100 1010 0000 1011

  $\approx$ -4.10253908

- 40049ACCCCCCCCCCCCCC

Representation of Characters

- Nonnumeric Text – alphabets, digits, punctuations etc.
- Some of standard coding schemes are: **ASCII, UNICODE.**

ASCII

- American Standard Code for Information Interchange.
- It uses a 7-bit code. All together we have 128 combinations of 7 bits and we can represent 128 character.
- As for example 65 = 1000001 represents character 'A'.

UNICODE

- **It is used to capture most of the languages of the world. It uses 16-bit**

- Unicode provides a unique number for every character

- The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others.

Characteristics of RISC Vs CISC processors

No.	RISC	CISC
1	Simple instructions taking one cycle	Complex instructions taking multiple cycles
2	Instructions are executed by hardwired control unit	Instructions are executed by microprogramed control unit
3	Few instructions	Many instructions
4	Fixed format instructions	Variable format instructions
5	Few addressing mode, and most instructions have register to register addressing mode	Many addressing modes
6	Multiple register set	Single register set
7	Highly pipelined	Not pipelined or less pipelined

Acknowledgement

Internet

PPT compiled/prepared by:

Raju K., Assoc. Prof.
CSE Dept, NMAMIT