Unit 2

Instruction Set Architecture

Instruction Set Architecture (ISA)

- Serves as an interface between software and hardware
- Typically consists of information regarding the programmers view of the architecture
  - Memory Organization-Address Space,Addressability
  - Register Set
  - Instruction Set – Opcodes, Data types, Addressing modes
- Many ISAs are not specific to a particular computer architecture

Instruction Set Design Issues

- Number of explicit Operands – 0,1,2,3
- Location of operands- Memory, Register, Accumulator
- Specification of operand locations –Addressing modes
- Sizes of operands supported – Bytes(8 bits),Half word(16 bits),Word(32 bits),Double(64 bits)
- Supported Operation-ADD, SUB,MUL,AND,OR

Number Representation

Number Systems-Basic

- Decimal number system
- Ten digits - Every digit position has a weight which is a power of 10
- Base or radix is 10
- Examples:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

$$250.67 = 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$$

Binary Number System

- Two digits 0 and 1
  - Every digit position has a weight that is a power of 2
  - Base or radix is 2
- Examples

$$110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

Binary to Decimal Conversion

- Each digit position of a binary number has a weight.
- Some power of 2.
- A binary number:

$B=b_{n-1}b_{n-2}\ldots\ldots\ldots b_1b_0 \,.\, b_{-1}b_{-2}\ldots\ldots b_{-m}$ where b_i are the binary digits

Corresponding value in decimal: $D=\sum_{i=-m}^{n-1} b_i 2^i$

Some Examples

1. $101011 \rightarrow 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = 43$

   $(101011)_2 = (43)_{10}$

2. $.0101 \rightarrow 0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4} = .3125$

   $(.0101)_2 = (.3125)_{10}$

3. $101.11 \rightarrow 1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2} = 5.75$

   $(101.11)_2 = (5.75)_{10}$

Decimal to Binary Conversion

- Consider the integer and fractional parts separately.
- For the integer part:
  - Repeatedly divide the given number by 2. and go on accumulating the remainders, until the number becomes Zero
  - Arrange the remainders in reverse order
- For the fractional part:
  - Repeatedly multiply the given fraction by 2
    - Accumulate the integer part (0 or 1)
    - If the integer part is 1,chop it off.
  - Arrange the integer parts the order they are obtained.

Examples

2	239	
2	119	--- 1
2	59	--- 1
2	29	--- 1
2	14	--- 1
2	7	--- 0
2	3	--- 1
2	1	--- 1
2	0	--- 1

$(239)_{10} = (11101111)_2$

2	64	
2	32	--- 0
2	16	--- 0
2	8	--- 0
2	4	--- 0
2	2	--- 0
2	1	--- 0
2	0	--- 1

$(64)_{10} = (1000000)_2$

$.634 \times 2 = 1.268$

$.268 \times 2 = 0.536$

$.536 \times 2 = 1.072$

$.072 \times 2 = 0.144$

$.144 \times 2 = 0.288$

:

$(.634)_{10} = (.10100.....)_2$

37.0625

$(37)_{10} = (100101)_2$

$(.0625)_{10} = (.0001)_2$

$\therefore (37.0625)_{10} =$

$(100101 . 0001)_2$

Octal Numbers

- Octal numbers (Radix or base=8) are made of octal digits: (0,1,2,3,4,5,6,7)
- Convert the following octal number to decimal

$$(465.27)_8 = 4x8^2 + 6x8^1 + 5x8^0 + 2x8^{-1} + 7x8^{-2}$$

$$=309.359$$

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27

Converting binary to Octal

For the integer part:

- Scan the binary number from right to left
- Translate each group of three bits into the corresponding octal digit.
  - Add leading zeros if necessary

For the fractional part:

- Scan the binary number from left to right
- Translate each group of three bits into the corresponding octal digit
  - Add trailing zeros if necessary

Converting binary to Octal

Example- − a) Convert binary number 1010111100 into octal number.

$= (1010111100)_2$

$= (001\ 010\ 111\ 100)_2$

$= (1\ 2\ 7\ 4)_8$

$= (1274)_8$

b) 101111001 into octal

$=(101\ 111\ 001)_2 =(5\ 7\ 1)_8$

c) 11000.10 into octal

$=(011\ 000.100)_2=(30.4)_8$

Octal to Binary

- Translate every octal digit into its 3-bit binary equivalent
- Example: $(765)_8=(111\ 110\ 101)_2$
- $(1054.02)_8=(001\ 000\ 101\ 100.000\ 010)_2$
- $(.654)_8=(.110\ 101\ 100)_2$

Hexadecimal Number System

- A compact way to represent binary numbers.
  - Group of four binary digits are represented a hexadecimal digit.
  - Hexadecimal digits are 0 to 9, A to F.

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Binary to Hexadecimal Conversion

- For the integer part:
  - Scan the binary number from right to left
  - Translate each group of four bits into the corresponding hexadecimal digit.
    - Add leading zeros if necessary
- For the fractional part:
  - Scan the binary number from left to right
  - Translate each group of four bits into the corresponding hexadecimal digit
    - Add trailing zeros if necessary

Examples

1. $(\underline{1011}\ \underline{0100}\ \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10}\ \underline{1010}\ \underline{0001})_2 = (2A1)_{16}$ *Two leading 0s are added*
3. $(.\underline{1000}\ \underline{010})_2 = (.84)_{16}$ *A trailing 0 is added*
4. $(\underline{101}\ .\ \underline{0101}\ \underline{111})_2 = (5.5E)_{16}$ *A leading 0 and trailing 0 are added*

Hexadecimal to Binary Conversion

- Translate every hexadecimal digit into its 4-bit binary equivalent
- Examples:

$$(3A5)_{16} = (0011\ 1010\ 0101)_2$$

$$(12.3D)_{16} = (0001\ 0010\ .\ 0011\ 1101)_2$$

$$(1.8)_{16} = (0001\ .\ 1000)_2$$

Exercise

1. Convert binary to decimal – 1011.100
2. Convert decimal to binary – 127.625 , 41.6875
3. Convert binary to octal – 11111100110.1011
4. Convert Octal to binary- 5077.371
5. Convert binary to hexadecimal – 11101010011111.1100
6. Convert hexadecimal to binary – 57B3.EF

Answers

1. 11.50
2. 1111111.10, 101001.1011
3. 3746.54
4. 101 000 111 111.011 111 001
5. 3A9F.C
6. 0101 0111 1011 0011.1110 1111

Number Representation and Arithmetic Operations

- The most natural way to represent a number in a computer system is by a string of bits, called a binary number
- Unsigned binary number- An n-bit binary number can have 2^n distinct combinations
- For example, For n=3, the S distinct combinations are:
  000, 001, 010, 011, 100, 101, 110, 111 (o to 2^3-1 = 7 in decimal).

Number of bits (n)	Range of Numbers
8	0 to 2^8-1 (255)
16	0 to 2^{16}-1 (65535)
32	0 to 2^{32}-1 (4294967295)
64	0 to 2^{64}-1

Integers

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

- Each digit position has a weight that is some power of 2

Signed Integer Representation

- Many of the numerical data items that are used in a program are signed (positive or negative) need to be represented
- Three possible approaches:

  a) Sign-magnitude representation

  b) One's complement representation

  c) Two's complement representation

Sign –Magnitude Representation

- For an n-bit number representation:
- The most significant bit (MSB) indicates sign (0: positive, 1: negative)
- In an n-bit word, the rightmost (n-1) bits represent the magnitude of the number
- Range of numbers: — $(2^{n-1} — 1)$ to + $(2^{n-1} — 1)$

Examples –Sign Magnitude Representation

Positive Integer	Negative Integer
+7=0111	-7=1111
+18=00010010	-18=10010010
+50=00110010	-50=10110010

1's Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form
  - Negative numbers are represented in 1's complement form
- How to compute the 1's complement of a number?
  - Complement every bit of the number
  - MSB will indicate the sign of the number (0: positive, 1: negative)

Examples-1's complement

Positive Integer	Negative Integer
+7=0111	-7=1000
+18=0001 0010	-18=1110 1101
+50=0011 0010	-50=1100 1101

2's Complement Representation

- Basic idea:
  - Positive numbers are represented exactly as in sign-magnitude form
  - Negative numbers are represented in 2's complement form
- How to compute the 2 's complement of a number?
- Complement every bit of the number(1 → 0 and 0→1),and then add one to the resulting number
- MSB Will indicate the sign of the number (0: positive, 1: negative).

Examples-2's Complement

Positive Integer	Negative Integer
+7=0111	-7=1001
+18=0001 0010	-18=1110 1110
+50=0011 0010	-50=1100 1110

Calculation:

```
+18 = 0001 0010
1's  = 1110 1101
     +         1
------------------------
       1110  1110  = -18
```

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

Fig 1:Binary, signed-integer representation

Example

Integer	Positive	Sign-Magnitude	1's complement	2's complement
56				
121				
75				
107				
43				

Example

Integer	Positive	Sign-Magnitude	1's complement	2's complement
56	0011 1000	1011 1000	1100 0111	1100 1000
121	0111 1001	1111 1001	1000 0110	1000 0111
75	0100 1011	1100 1011	1011 0100	1011 0101
107	0110 1011	1110 1011	1001 0100	1001 0101
43	0010 1011	1010 1011	1101 0100	1101 0101

Addition of 1-bit unsigned numbers

- Addition of 1-bit number is shown in the figure 2

Fig 2 Addition of 1-bit Number

- The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2
- We say that the sum is 0 and the carry-out is 1

Addition of multiple-bit unsigned numbers (contd…)

- In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers.
- We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end
- The carry-out from a bit pair becomes the carry-in to the next bit pair to the left
- The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is l, which represents the value 3.

Addition of multiple bits

	1	1	1	1	←	carry
	1	1	1	0	1	
(+)	1	1	0	1	1	
1	1	1	0	0	0	

Addition in 2's complement

- Addition proceeds as if the two numbers were unsigned integers
- If the result of the operation is positive, we get a positive number in 2's complement form, which is the same as in unsigned-integer form
- If the result of the operation is negative, we get a negative number in twos complement form
- Note that, in some instances, there is a carry bit beyond the end of the word which is ignored.

Overflow in Integer Arithmetic

- On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**
- When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result
- When adding unsigned numbers, a carry-out of 1 from the most significant bit position indicates that an overflow has occurred. However, this is not always true when adding signed numbers

Overflow in Integer Arithmetic

- For example, using 2's-complement representation for 4-bit signed numbers, if we add +7 and +4, the sum vector is 1011, which is the representation for −5, an incorrect result. In this case, the carry-out bit from the MSB position is 0.
- If we add −4 and −6,we get 0110 = +6, also an incorrect result. In this case, the carry-out bit is 1.
- Clearly, overflow may occur only if both summands have the same sign
- The addition of numbers with different signs cannot cause overflow because the result is always within the representable range

Examples

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Subtraction in 2's Complement

- Subtraction is easily handled with the following rule:
- **Subtraction rule:** To subtract one number (subtrahend) from another (minuend)
  - Take the twos complement (negation) of the subtrahend
  - Add it to the minuend.
- Thus, subtraction is achieved using addition

Example

```
  0010 =  2
 +1001 = -7
  1011 = -5

(a) M = 2 = 0010
    S = 7 = 0111
   -S =     1001
```

```
  0101 =  5
 +1110 = -2
 10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   -S =     1110
```

```
  1011 = -5
 +1110 = -2
 11001 = -7

(c) M = -5 = 1011
    S =  2 = 0010
   -S =      1110
```

```
  0101 = 5
 +0010 = 2
  0111 = 7

(d) M =  5 = 0101
    S = -2 = 1110
   -S =      0010
```

```
  0111 = 7
 +0111 = 7
  1110 = Overflow

(e) M =  7 = 0111
    S = -7 = 1001
   -S =      0111
```

```
  1010 = -6
 +1100 = -4
 10110 = Overflow

(f) M = -6 = 1010
    S =  4 = 0100
   -S =      1100
```

Character Representation

- The most common encoding scheme for characters is ASCII (American Standard Code for Information Interchange)
- Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes as shown in the table 2.1(slide 43)
- It is convenient to use an 8-bit *byte* to represent and store a character
- The code occupies the low-order seven bits. The high-order bit is usually set to 0

Character Representation(Cont'd)

- Note that the codes for the alphabetic and numeric characters are in increasing sequential order when interpreted as unsigned binary numbers
- This facilitates sorting operations on alphabetic and numeric data
- The low-order four bits of the ASCII codes for the decimal digits 0 to 9 are the first ten values of the binary number system
- This 4-bit encoding is referred to as the *binary-coded decimal* (BCD) code.

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	´	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	1	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

Table 2.1 7-bit ASCII code

Memory Locations and Addresses

- Memory is one of the most important subsystems of a computer that determines the overall performance
- Conceptual view of Memory
  - Array of storage location, with each storage location having a unique address
  - Each storage can hold a fixed amount of information
- The memory consists of many millions of storage *cells*, each of which can store a *bit* of information having the value 0 or 1
- A memory system with M locations and N bits per location is referred to as MxN memory
- Both M and N are powers of 2 : Example- 1024x8, 65536x32

Memory Locations and Addresses(cont'd)

- The memory is organized so that a group of *n* bits can be stored or retrieved in a single, basic operation
- Each group of *n* bits is referred to as a *word* of information, and *n* is called the *word length*
- Modern computers have word lengths that typically range from 16 to 64 bits
- If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 2.2.

Memory Locations and Addresses(cont'd)

(a) A signed integer

(b) Four characters

How is memory organized?

- Memory is often byte organized
  - Every byte of the memory has unique address-byte addressable
- Multiple bytes of data can be accessed by the instruction
  - Example: Half word=2 bytes, word=4 bytes, Long word=8 bytes
- For higher data transfer rate, memory is often organized such that multiple bytes can be read or written simultaneously

How do we specify memory sizes?

Unit	Bytes	In decimal
8 bits(B)	1 or 2^0	10^0
Kilobyte(KB)	1024 or 2^{10}	10^3
Megabyte(MB)	2^{20}	10^6
Gigabyte(GB)	2^{30}	10^9
Terabyte(TB)	2^{40}	10^{12}
Petabyte(PB)	2^{50}	10^{15}
Exabyte(EB)	2^{60}	10^{18}
Zettabyte(ZB)	2^{70}	10^{21}

Memory Storage Locations

- If there are n bits in an address, the maximum number of storage locations that can be accessed is 2^n
- Example n = 3 ,we can have locations we can access; $2^3 = 8$
- So, the first location will be 0 0 0, the next location will be 0 0 1, next will be 0 1 0, 0 1 1, 1 0 0, 1 0 1, 1 1 0 and 1 1 1
- So, for n = 8, 256 locations (2^8); for n = 16, 2^{16} = 64K locations; for n = 20, 1M locations, etc. can be accessed

Memory Storage Locations

Address	Contents
0000 0000	0000 0000 0000 0001
0000 0001	0000 0100 0101 0000
0000 0010	1010 1000 0000 0000
⋮	⋮
1111 1111	1011 0000 0000 1010

Example: 2^8 x 16 memory

Byte Ordering Conventions

- Many data items require multiple bytes for storage
- Different computers use different data ordering conventions
  - **Low-order byte first**
  - **High-order byte first**
- Thus a 16 bit number 11001100 00100010 can be stored as either:

11001100 00100010 or 00100010 11001100

Data Type	Size(in bytes)
Character	1
Integer	4
Long Integer	8
Float	4
Double-Precision	8

Typical Data Sizes

Little-Endian and Big-Endian Assignments

- Little-Endian
  - The least significant byte is stored at lower address followed by most significant byte
  - Example: Intel processor, DEC alpha
- Big-Endian
  - The most significant byte is stored at lower address followed by least significant byte
  - Example :IBM's 370 mainframes, Motorola Microprocessors, TCP/IP

An Example

- Represent the following 32 bit number in both Little-Endian and Big-Endian in memory from address 2000 onwards:

01010101 00110011 01001111 11000011

Address	Data
2000	11000011
2001	01001111
2002	00110011
2003	01010101

Little Endian

Address	Data
2000	01010101
2001	00110011
2002	01001111
2003	11000011

Big Endian

An Example

- Represent the following 32 bit hexadecimal number in both Little-Endian and Big-Endian in memory from address 1160 onwards if each location can store 1 byte of data: D796234A

11010111 10010110 00100011 01001010

Address	Data
1160	01001010
1161	00100011
1162	10010110
1163	11010111

Little Endian

Address	Data
1160	11010111
1161	10010110
1162	00100011
1163	01001010

Big Endian

Memory Operations

- Both program instructions and data operands are 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. Operands and results must also be moved between the memory and the processor
- For executing the program, two basic operations are required.
  a) Load: The contents of a specified memory location is read into a processor register
     LOAD R1, 2000
  b) Store: The contents of a processor register is written into a specified memory location.
     STORE 2020, R3

Example

Compute S = (A + B) — (C — D)

```
LOAD   R1,A
LOAD   R2,B
ADD    R3,R1,R2        // R3 = A + B
LOAD   R1,C
LOAD   R2,D
SUB    R4,R1,R2        // R4 = C — D
SUB    R3,R3,R4        // R3 = R3 — R4
STORE  S,R3
```

Instruction Format

- An instruction consists of two parts:-
- Operation or Opcode
  - Specifies the operation to be performed by the instruction.
  - Various categories of instructions: data transfer, arithmetic and logical, control, I/O and special machine control.
- Operand(s)
  - Specifies the source(S) and destination of the operation
  - Source operand can be specified by an immediate data, by naming a register, or specifying the address of memory
  - Destination can be specified by a register or address

Instruction Format

- Opcodes are represented by abbreviations, called *mnemonics,* that indicate the operation.
- Common examples include
  - ADD Add
  - SUB Subtract
  - MUL Multiply
  - DIV Divide
  - LOAD Load data from memory
  - STORE Store data to memory

Instruction Format

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Addressing Modes

- Addressing modes specify the mechanism by which the operand data can be located
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced

Opcode	**Mode**	**Address**

Figure: Instruction format with mode field

Addressing Modes

- Addressing modes specify the mechanism by which the operand data can be located
- Various addressing modes exist:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register indirect
  - Displacement
  - Stack

Immediate Addressing

- The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction
- This mode can be used to define and use constants or set initial values of variables
- The number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit
- When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer

Immediate Addressing

- The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

  Operand = A

- This mode can be used to define and use constants or set initial values of variables
- Example: MOV R1,100
- Advantage: No memory reference
- Disadvantage: Limited Operand Magnitude

Direct Addressing

- A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$EA = A$$

- Example: MOV R2, LOCA
- Advantage: Simple
- Disadvantage: Limited Address Space

Indirect Addressing

The address field here refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as indirect addressing:

EA = (A)

- Example: Add R0,(A), Add R1,(20A6H)
- Advantage: Large Address Space
- Disadvantage: Multiple memory reference

Register Addressing

- The operand is held in a register, and the instruction specifies the register number
- Examples: ADD R1,R2,R3 , MOV R2,R5
- Advantages
  - Very few number of bits needed, as the number of registers is limited
  - Faster execution. since no memory is required for getting the operand
- Disadvantages
  - Address Space is very limited

Register Indirect Addressing

- The instruction specifies a register, and the register holds the memory address where the operand is stored
  - Can access large address space.
  - One fewer compared to indirect addressing.
- Example: — ADD R1,(R5)

Displacement

- A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing.

  EA = A + (R)
- Displacement addressing requires that the instruction have two address fields, at least one of which is explicit
- The value contained in one address field (value = A) is used directly
- The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address

Displacement

Relative Addressing(PC Relative)

- The instruction specifies an offset of displacement, which is added to the program counter(PC) to get the effective address of the operand
- Since the no, of bits to specify the offset is limited, the range of relative addressing is also limited

Indexed Addressing

- Either a special-purpose register or a general-purpose ,is used as index register in this addressing mode
- The instruction specifies an offset of displacement, which is added to the index register to get the effective address of the operand
- Example:
  - LOAD R1, 1040(R3)
- Can be used to sequentially access the elements of an array
  - Offset gives the starting address of the array, and the index register value specifies the array elements to be used

Indexed Addressing

Base Addressing

- The processor has a special register called the base register or segment register
- All operand addresses generated are added to the base register to get the final memory address
- Allow easy movement of code and data in memory

Stack Addressing

- The final addressing mode that we consider is stack addressing
- A stack is a linear array of locations. It is sometimes referred to as a *pushdown list* or *last-in-first-out queue*.
- Items are appended to the top of the stack so that, at any given time, the block is partially filled
- Associated with the stack is a pointer whose value is the address of the top of the stack
- Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack.

Stack Addressing

- The stack pointer is maintained in a register.
- Thus, references to stack locations in memory are in fact register indirect addresses.
- The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

Stack Addressing

- Operand is implicitly on top of the stack
- Used in zero-address machines earlier
  - ADD
  - PUSH X
  - POP X
- Many processors have a special register called the stack pointer (SP) that keeps track of the stack-top in memory
  - PUSH, POP, CALL, RET instructions automatically modify SP

PC=200	R1=400
XR=100	AC

Address	Memory	
200	Load to AC	Mode
201	Address=500	
202	Next Instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

Addressing Mode	Effective Address	Content of AC
Immediate		
Direct		
Indirect Address		
Relative Address		
Indexed Address		
Register		
Register-Indirect		

Addressing Mode	Effective Address	Content of AC
Immediate	201	500
Direct	500	800
Indirect Address	800	300
Relative Address	702	325
Indexed Address	600	900
Register	-------	400
Register-Indirect	400	700

RISC and CISC Instruction Sets

- One of the most important characteristics that distinguish different computers is the nature of their instructions
- Two approaches in the design of instruction sets for modern computers
  - Reduced Instruction Set Computers(RISC)
  - Complex Instruction Set Computer(CISC)

RISC

- The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer
- Such computers are called Reduced Instruction Set Computers (RISC)
- Example-CDC 6600 (1964), ARM

RISC Features

- Simple addressing modes
- All instructions fitting in a single word
- Fewer instructions in the instruction set, as a consequence of simple addressing modes
- Arithmetic and logic operations that can be performed only on operands in processor registers

RISC Features

- Load/store architecture that does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
- Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining
- Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

RISC Instructions

- Memory operands are accessed only using Load and Store instructions
- Example: To execute C=A+B, we use three registers R2,R3,R4 ,the four instructions required are:

  Load R2, A

  Load R3, B

  Add R4, R2, R3

  Store R4, C

RISC Instruction

Figure: Instruction Execution and Straight-line Sequencing

RISC-Branching

- Consider the task of adding a list of *n* numbers
- Two Approaches
- 1) The addresses of the memory locations containing the *n* numbers are symbolically given as NUM1, NUM2, . . . , NUM*n*, and separate Load and Add instructions are used to add each number to the contents of register R2
- After all the numbers have been added, the result is placed in memory location SUM.

i	Load	R2, NUM1
$i + 4$	Load	R3, NUM2
$i + 8$	Add	R2, R2, R3
$i + 12$	Load	R3, NUM3
$i + 16$	Add	R2, R2, R3
	⋮	
$i + 8n - 12$	Load	R3, NUMn
$i + 8n - 8$	Add	R2, R2, R3
$i + 8n - 4$	Store	R2, SUM
	⋮	
SUM		
NUM1		
NUM2		
	⋮	
NUMn		

Figure: Adding list of numbers without branch

RISC-Branching

- 2) It is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum.
- To add all numbers, the loop has to be executed as many times as there are numbers in the list

Figure: Adding list of numbers with branch

RISC –Addressing Mode

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = effective address
Value = a signed number
X = index value

```
        Load            R2, N        Load the size of the list.
        Clear           R3           Initialize sum to 0.
        Move            R4, #NUM1    Get address of the first number.
LOOP:   Load            R5, (R4)     Get the next number.
        Add             R3, R3, R5   Add this number to sum.
        Add             R4, R4, #4   Increment the pointer to the list.
        Subtract        R2, R2, #1   Decrement the counter.
        Branch_if_[R2]>0 LOOP        Branch back if not finished.
        Store           R3, SUM      Store the final sum.
```

Figure: Use of Indirect address in addling list of N numbers

CISC(Complex Instruction Set Computer)

- Traditional approach
- Main features :
- A large number of instructions—typically from 100 to 250 instructions.
- Some instructions that perform specialized tasks and are used infrequently
- A large variety of addressing modes—typically from 5 to 20 different modes(R-R,R-M,M-M,Indexed,Indirect)
- Variable-length instruction formats

CISC(Complex Instruction Set Computer)

- Ease of mapping high-level language statement to machine code is possible in CISC, but instruction decoding and control unit design are much more complex
- In the variable length instruction, pipeline implementation will also be quite complex
- Example
  - IBM 360/370 (1960-70)
  - VAX-11/780 (1970-80)
  - Intel x86/Pentium (1985-Present)

CISC(Complex Instruction Set Computer)

- CISC instruction sets are not constrained to the load/store architecture, in which arithmetic and logic operations can be performed only on operands that are in processor registers
- Instructions do not necessarily have to fit into a single word. Some instructions may occupy a single word, but others may span multiple words
- Instructions in modern CISC processors typically do not use a three-address format

CISC(Complex Instruction Set Computer)

- Most arithmetic and logic instructions use the two-address format:

  Operation destination, source

- An Add instruction of this type is Add B, A

  which performs the operation B ← [A] + [B] on memory operands

CISC(Complex Instruction Set Computer)

- Consider again the task of adding two numbers C = A + B where all three operands may be in memory locations
- Obviously, this cannot be done with a single two-address instruction. The task can be performed by using another two-address instruction that copies the contents of one memory location into another
- Such an instruction is Move C, B which performs the operation C←[B], leaving the contents of location B unchanged

CISC(Complex Instruction Set Computer)

- The operation C←[A] + [B] can now be performed by the two-instruction sequence

  Move C, B

  Add C, A

- Observe that by using this sequence of instructions the contents of neither A nor B locations are overwritten.

CISC(Complex Instruction Set Computer)

- In some CISC processors one operand may be in the memory but the other must be in a register.
- In this case, the instruction sequence for the required task would be

  Move Ri, A

  Add Ri, B

  Move C, Ri

CISC-Addressing Modes

- Most CISC processors have all of the five basic addressing modes —Immediate, Register, Absolute, Indirect, and Index
- Three additional addressing modes are often found in CISC processors
- ***Autoincrement and Autodecrement Modes*** - There are two modes that are particularly convenient for accessing data items in successive locations in the memory and for implementation of stacks.
- ***Relative mode***—The effective address is determined by the Index mode using the program counter in place of the general-purpose register R*i*.

CISC-Addressing Modes

- Autoincrement Mode:
- The effective address of the operand is the contents of a register specified in the instruction
- After accessing the operand, the contents of this register are automatically incremented to point to the next operand in memory
- Autoincrement mode is written as *(R*i*)+*

CISC-Addressing Modes

- We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed
- Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand
- Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

CISC-Addressing Modes

- Autodecrement Mode: The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.
- We denote the Autodecrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address
- Thus, we write $-(R_i)$
- In this mode, operands are accessed in descending address order

CISC-Addressing Modes

- Example: Main reason for autoincrement and autodecrement mode is to implement a stack structure.
- Instead of needing two instructions to push a new item on the stack

  Subtract SP, #4

  Move (SP), NEWITEM we can use just one instruction Move −(SP), NEWITEM
- Similarly, instead of needing two instructions to pop an item from the stack
  - Move ITEM, (SP)
  - Add SP, #4 , we can use just have Move ITEM, (SP)+

CISC-Addressing Modes

- Relative Mode:
- Some computers have a version of this mode in which the program counter, PC, is used instead of a general-purpose register
- Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter
- Since the addressed location is identified *relative* to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

CISC-Example

```
        Move        R2, N        Load the size of the list.
        Clear       R3           Initialize sum to 0.
        Move        R4, #NUM1    Load address of the first number.
LOOP:   Add         R3, (R4)+    Add the next number to sum.
        Subtract    R2, #1       Decrement the counter.
        Branch>0    LOOP         Loop back if not finished.
        Move        SUM, R3      Store the final sum.
```