CSE 4305
Computer Organization and Architecture

Modes and Formats

Course Teacher: Md. Hamjajul Ashmafee

Lecturer, CSE, IUT

Email: ashmafee@iut-dhaka.edu

Introduction

- How to specify the **operands** in an instruction?
- How to specify the **operation name** in an instruction?
- How the **addresses of the operands** are mentioned in an address if addresses of those operands are referred?
- How the **bits of an instructions** are <u>organized</u> to define addresses and operation?

Addressing Modes

- Comparing with **instruction length**, address field of fields are relatively smaller in size – how to refer a large range of memory locations from main memory or virtual memory?
- **Trade-off** involved among range of addressing, the number of memory references and complexity of address calculation in an instruction
- Normally there are more than one addressing modes – how to determine which one is used?
  - **Mode field** – one or more bits in an instruction
- **Effective Address (EA)** – memory address (main or virtual), register address where the original operand is stored

Addressing Modes…

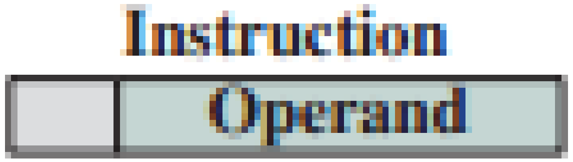
Following notations will be used:

- **A** = contents of an address field in the instruction
- **R** = contents of an address field in the instruction that refers to a register
- **EA** = actual (effective) address of the location containing the referenced operand
- **(X)** = contents of memory location X or register X

Immediate Addressing

- Simplest form
- **Operand value** is present in the instruction

Operand = A

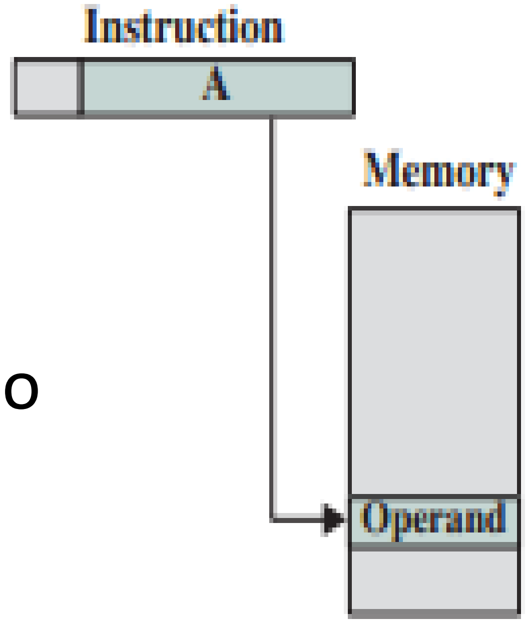


- Used to define constant or initial value
- Stored in 2's complement form (MSB is sign bit)
- **Advantage**: no memory reference is required; just one instruction fetch cycle is needed
- **Disadvantage**: the size of the value is restricted to the size of address field in the instruction

Direct Addressing

- Very simple form of addressing
- **Address field** in the instruction contains the effective address of the operand in memory

EA = A



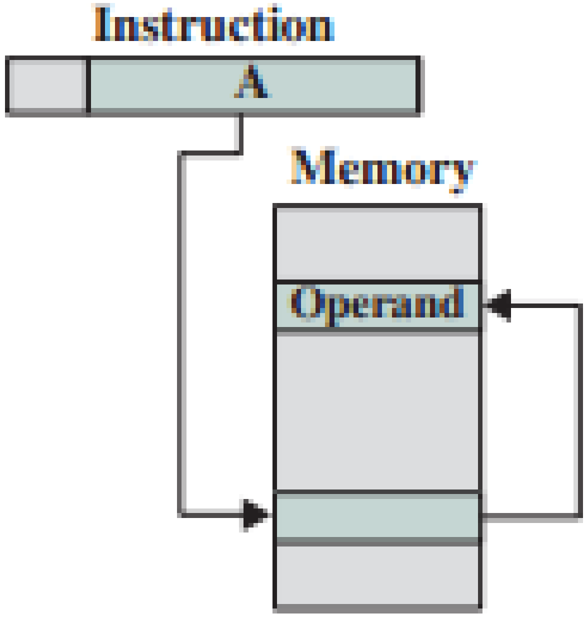
- Common in earlier generation of computers but not nowadays
- **Advantage:** it requires only one memory reference but no extra calculation
- **Disadvantage:** provides limited address space

Indirect Addressing

- To increase the address space, we **utilize** total word length of any memory location to address the original operand
- The address field of the instruction refers to the address of a word in memory, which in turn contains full length address of the operand

$$EA=(A)$$

- The **parentheses** are interpreted as meaning ***content of***



Indirect Addressing…

- k = length of address field in an instruction and N = length of a word
- 2^K addresses are available to refer memory addresses for effective address from an instruction's address field
- 2^N different effective addresses are available now from that memory reference (**advantage**)
- Requires two memory cycles to fetch the operand (**disadvantage**)
- Generally, effective addresses are referred to the words in **page 0** of <u>virtual memory</u>
- A rarely used variant is **multilevel** or **cascaded** indirect addressing using indirect flag: **EA=(…(A)…)**

Register Addressing

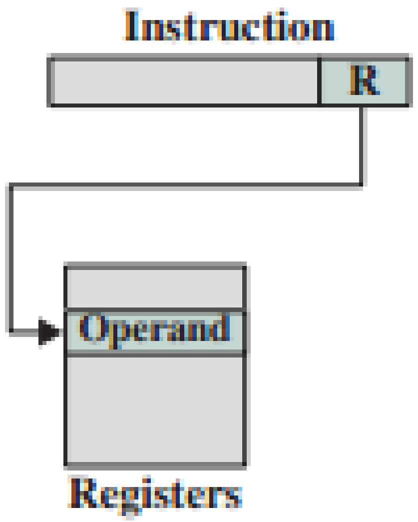
- Similar to direct addressing – only difference in **address field** referring a register address rather than a main memory address

EA = R

- **Register address field** of an instruction contains the address of the intended register where the operand is stored
- **Few bits** are required to specify the register address as their amount is less (3-5 bits to refer 8-32 registers)
- **Advantage:** small address fields in instruction and no memory reference and less execution time
- **Disadvantage**: limited address space

Register Addressing…

- But **excessive use** of **registers** needs efficient implementation of the processor architecture – which is not ideal – rather we can use registers for those operands which will be called very frequently (e.g. intermediate result of any calculation)

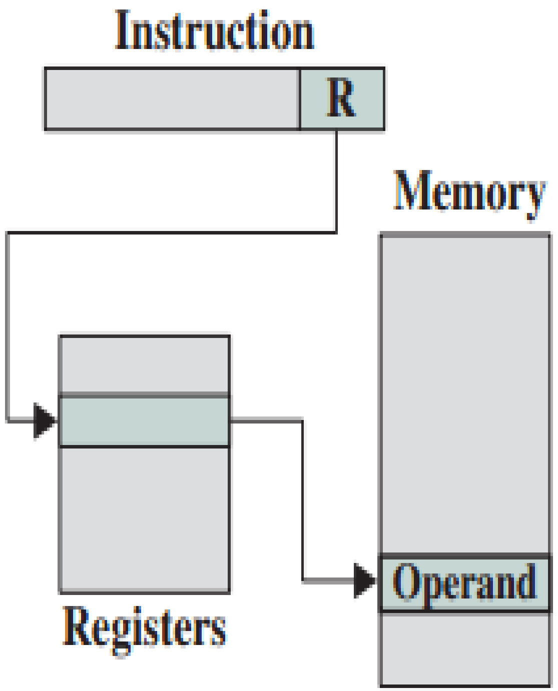


Register Indirect Addressing

- Analogous to **indirect addressing** – only difference is the register address field of an instruction refers to a memory location or register

$$EA = (R)$$

- **Advantage:** same as indirect addressing mode and also it requires one less memory access than indirect addressing mode
- **Disadvantage**: requires more time to execute than other addressing modes

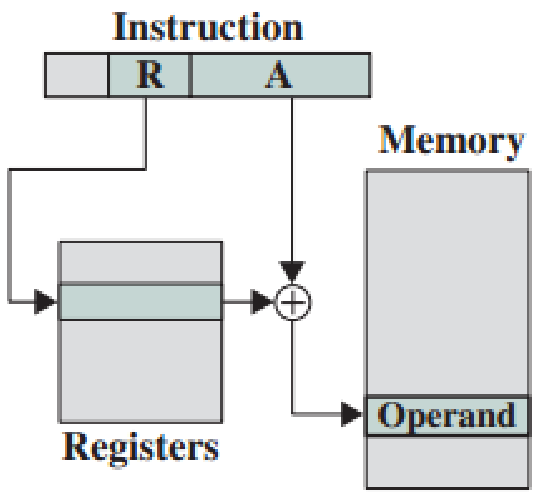


Displacement Addressing

- Very powerful addressing mode – take the advantages of direct addressing and register indirect addressing
- Have a variety of names based on their context

$$EA = A + (R)$$

- Instructions have **two address field**; one of which must be **explicit,** another can be **implicit** based on opcode (any register) – content of both fields are added to generate the effective address
- 3 most common uses:
  - Relative addressing
  - Base register addressing
  - Indexing



DA: Relative Addressing

- Also called as **PC-relative addressing**
- Implicit reference register = Program Counter

$$\mathbf{EA = (PC) + A}$$

- **A** holds 2's complement number for the operation
- Exploits the concept of **locality** – most memory references are relatively near to the instruction being executed

DA: Base-Register Addressing

- Referenced register form the instruction contains the main memory address and the address field contains the displacement (unsigned integer)

EA = (R) + A (displacement)

- **Register reference** may be <u>implicit or explicit</u>
- Based on the **concept of segmentation** – segmentation address (base register) and offset (displacement)

DA: Indexing

- Address field of the instruction contains a main memory address and the referenced register contains a positive displacement from that address – just opposite of Base-register addressing

EA = A (main memory address) + (R) (index/displacement)

- Used for iterative operations to access the operands sequentially (e.g. an array or a list)
- As the indexing is used in register it will take **less time to calculate the EA**

DA: Indexing

- **Auto-indexing:** to **increment** or **decrement** as a part of instruction automatically – array operation

EA = A +(R); (R) = (R) + 1

- **Post-indexing:** if **indirect addressing** and **indexing** is used together and the indexing is performed after the indirection – to access one of a number of block data

EA = (A) + (R)

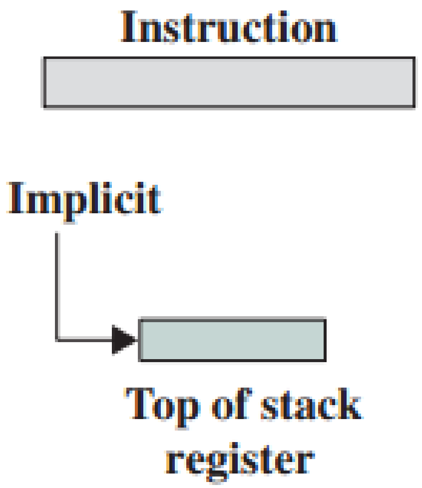
- **Pre-indexing:** if **indirect addressing** and **indexing** is used together and the indexing is performed before the indirection – multiway branch table where table entry holds different locations of different branches (A)

EA = (A +(R))

- Pre-indexing and post-indexing can't be used together

Stack Addressing

- Stack - **a linear array of locations** – last in first out queue
- **Reserved** block of locations
- To address **a stack a pointer** is used to indicate top of the stack (TOS) – may referenced by a register (register indirect)
- Sometimes **top two items** of the stack may be referenced
- **Implicit addressing** – addresses need not be included in the instruction directly



Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Instruction Formats

- An Instruction Format defines the **layout of the bits** of an instruction in terms of its elementary fields
- It includes **an opcode** (must) and **implicitly or explicitly zero or more operands** following a particular addressing mode
- Format **also** mentions the **addressing mode** for each operand
- For a particular **instruction se**t, there are **more than one instruction format**

Instruction Length

- **Basic design issue** – affected by memory size, memory organization, bus structure, processor complexity, processor speed and so on
- It defines the **richness and flexibility** of the machine in assembly language
- Obvious **trade-off** between the powerful instruction repertoire and the need to save space
    - More opcodes, more operands make the programming easier to write shorter code
    - More addressing modes gives more flexibility to implement any function
    - Increased memory size provides more physical addresses
- But they needs more bits to make instruction longer – may be wasteful

Instruction Length…

- Moreover, the instruction length should be equal to the memory transfer length or multiple of that to get **integral number** of instructions during fetch cycle
- Also **memory transfer** rate should be considered as it is **slower** than the processor execution time –
  - **Use cache memory**
  - **Shorter instructions can be designed**
- Instruction length can be multiple of the character length

Allocation of Bits

- Equally difficult issue is to **allocate bits** in **different fields**
- For a given instruction length, there is a **tradeoff** between the **number of opcodes** and the **power of the addressing capability**
- **More opcodes more bits in opcode field**
- Sometimes invoke another issues – **fixed length instruction** and **variable length instruction** (additional operations may be specified using additional bits in the instruction)

Allocation of Bits…

- Following factors are interrelated to determine the use of addressing bits:
  - **Number of addressing mode**: specified implicitly or explicitly in the instruction
  - **Number of operands**: fewer addresses in instruction makes the program longer and complex – operands also requires own mode indicator
  - **Register vs Memory**: if the register is implicit it consumes no bits in instruction – user can use 8-32 registers in contemporary architecture
  - **Number of register sets**: if another functional split in registers is there, it requires less bits as reference
  - **Address Range:** the number of bits in address field defines the range in memory to locate in maximum
  - **Address Granularity**: choice of addressing a word (16 or 32 bits) or a byte

PDP-8's Instruction Format

- **Simplest** instruction design
- **12-bits instructions** for **12-bits words**
- **Single general purpose register** – accumulator (AC)
- **Flexible addressing** – memory reference consisting of 7 bits and two 1-bit modifiers – memory is divided into fixed length pages (2^7 words each)
- **Address calculation** is based on referring **page-0 or current page** indicating by page bit (Z/C)
- **Direct or indirect addressing** based on second modifier bit (D/I)
- 3 bit opcode – three types of instructions (single address memory reference (0-5), microinstruction (7), I/O operation (6))

PDP-8's Instruction Format…

Memory reference instructions

Opcode			D/I	Z/C	Displacement	
0		2	3	4	5	11

Input/output instructions

1	1	0	Device		Opcode	
0		2	3	8	9	11

Register reference instructions

Group 1 microinstructions

1	1	1	0	CLA	CLL	CMA	CML	RAR	RAL	BSW	IAC
0	1	2	3	4	5	6	7	8	9	10	11

Group 2 microinstructions

1	1	1	0	CLA	SMA	SZA	SNL	RSS	OSR	HLT	0
0	1	2	3	4	5	6	7	8	9	10	11

Group 3 microinstructions

1	1	1	0	CLA	MQA	0	MQL	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11

D/I = Direct/Indirect address
Z/C = Page 0 or Current page
CLA = Clear Accumulator
CLL = Clear Link
CMA = CoMplement Accumulator
CML = CoMplement Link
RAR = Rotate Accumulator Right
RAL = Rotate Accumulator Left
BSW = Byte SWap

IAC = Increment ACcumulator
SMA = Skip on Minus Accumulator
SZA = Skip on Zero Accumulator
SNL = Skip on Nonzero Link
RSS = Reverse Skip Sense
OSR = Or with Switch Register
HLT = HaLT
MQA = Multiplier Quotient into Accumulator
MQL = Multiplier Quotient Load

PDP 10's Instruction Format

- Self Study

Variable Length Instruction

- Programmers **get flexibility to write instructions** of different lengths
- Provide **large repertoire of opcodes** with their different lengths
- **Addressing also become flexible** with various combinations of register and memory references plus addressing modes
- **Increase the complexity** of the processor – but it is affordable today
- But it also maintains its **integral length** related to the word length -

PDP-11 and VAX's Instruction Format

- Self Study

Assembly Language

- **A processor** understand and execute **machine instructions** written in binary numbers
- If the programmer writes code in machine instruction, it will be very **difficult** for him – rather he can use their **hexadecimal code**
- For more improvement, a programmer can use **symbolic name** of each instruction – **symbolic programming**
- To avoid absolute address, we can use **symbolic addresses** to make the program flexible to run from anywhere from memory

Assembly Language…

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program