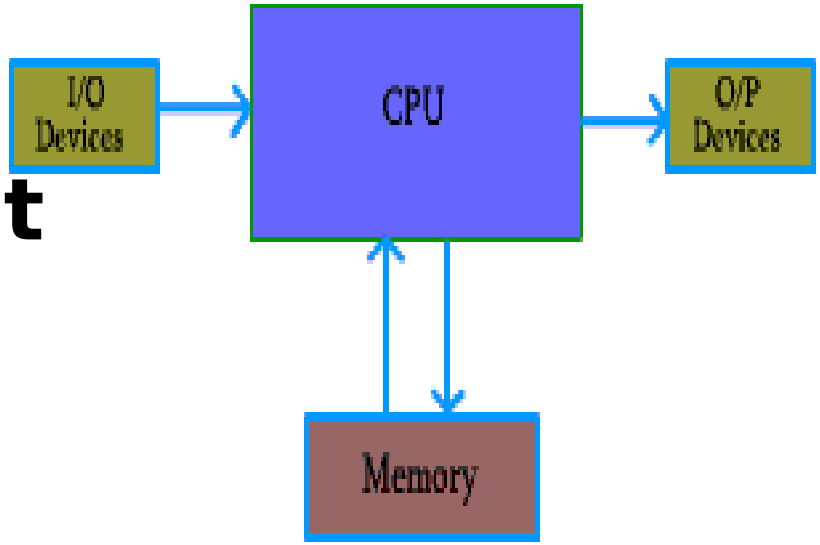
Computer organization

- Computer organization refers to the operational units and their interconnections that realize the architectural specifications.
- Examples of organizational attributes include those hardware details transparent to the programmer, such as
  - control signals,
  - interfaces between the computer and peripherals, and the memory technology used.

Figure 1.1: Basic Unit of a Computer

- **Central Processor Unit**
- **Input Unit**
- **Output Unit**
- **Memory Unit**



Central Processor Unit (CPU) :

- Central processor unit consists of two basic blocks :
  - The program control unit has a set of registers and control circuit to generate control signals.
  - The execution unit or data processing unit contains a set of registers for storing data and an Arithmetic and Logic Unit (ALU) for execution of arithmetic and logical operations.
- CPU may have some additional registers for temporary storage of data.

Input Unit

- With input unit data from outside can be supplied to the computer.
- Program or data is read into main storage from input device or secondary storage under the control of input instruction.
- Example of input devices:
- Keyboard, Mouse, Hard disk, Floppy disk, CD-ROM drive etc.

Output Unit

- With output unit computer results are provided to the user or it can be stored in storage device permanently for future use. Output data from main storage go to output device under the control of output instructions.
- Example of output devices:
- Printer, Monitor, Plotter, Hard Disk, Floppy Disk etc.

Memory Unit

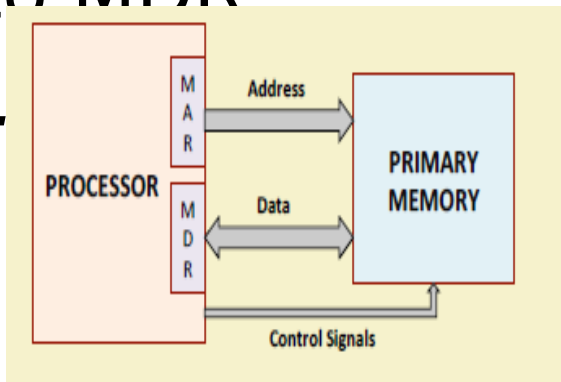
- Memory unit is used to store the data and program. CPU can work with the information stored in memory unit. This memory unit is termed as primary memory or main memory module. These are semi conductor memories.
- two types of semiconductor memories -
- Volatile Memory : RAM (Random Access Memory).
- Non-Volatile Memory :
  - ROM (Read only Memory), PROM (Programmable ROM)
    EPROM (Erasable PROM), EEPROM (Electrically Erasable PROM).

Secondary Memory

- Secondary memories are non volatile memory and it is used for permanent storage of data and program.
- Example of secondary memories
- Hard Disk, Floppy Disk, Magnetic Tape :magnetic devices
- CD-ROM: optical device
- Thumb drive (or pen drive):semiconductor memory

Interfacing with the Primary Memory

- To read data from memory
  - Load the memory address into MAR.
  - Issue the control signal *READ.*
  - The data read from the memory is stored into MDR.
- To write data into memory
  - Load the memory address into MAR.
  - Load the data to be written into MDR
  - Issue the control signal *WRITE.*



Exercise: Size of Registers

- Given : Memory size = 4096 words

  Word size = 16 bit

  One word Instruction is used

  - 1. Draw a schematic diagram of memory with above stated value
  - 2.Give word size in bytes
  - 3. How many bits required to select a word in memory?
  - 4. What is the size of MAR in bits?
  - 5. What is the size of MDR in bits and in bytes?
  - 6. What is the size of the Program Counter(PC)?
  - 7. What is the size of Instruction Register(IR)?

Registers

- The memory has 4096 words in it
- 4096 = 2^{12}, so it takes 12 bits to select a word in memory
- Each word is 16 bits long
- Size of MAR=12 bits
- Size of MDR=16bits
- Two special-purpose registers are used:
- ***Program Counter (PC):*** *Holds the memory address of the next* instruction to be executed.
- Automatically incremented to point to the next instruction when an instruction is being executed.
- ***Instruction Register (IR):*** *Temporarily holds an instruction that has* been fetched from memory.
- Need to be decoded to find out the instruction type.
- Also contains information about the location of the data.

Byte-Addressable Memory- 32 bits word

Word Address				
0	0	1	2	3
4	4	5	6	7
8	8			11
12	12			

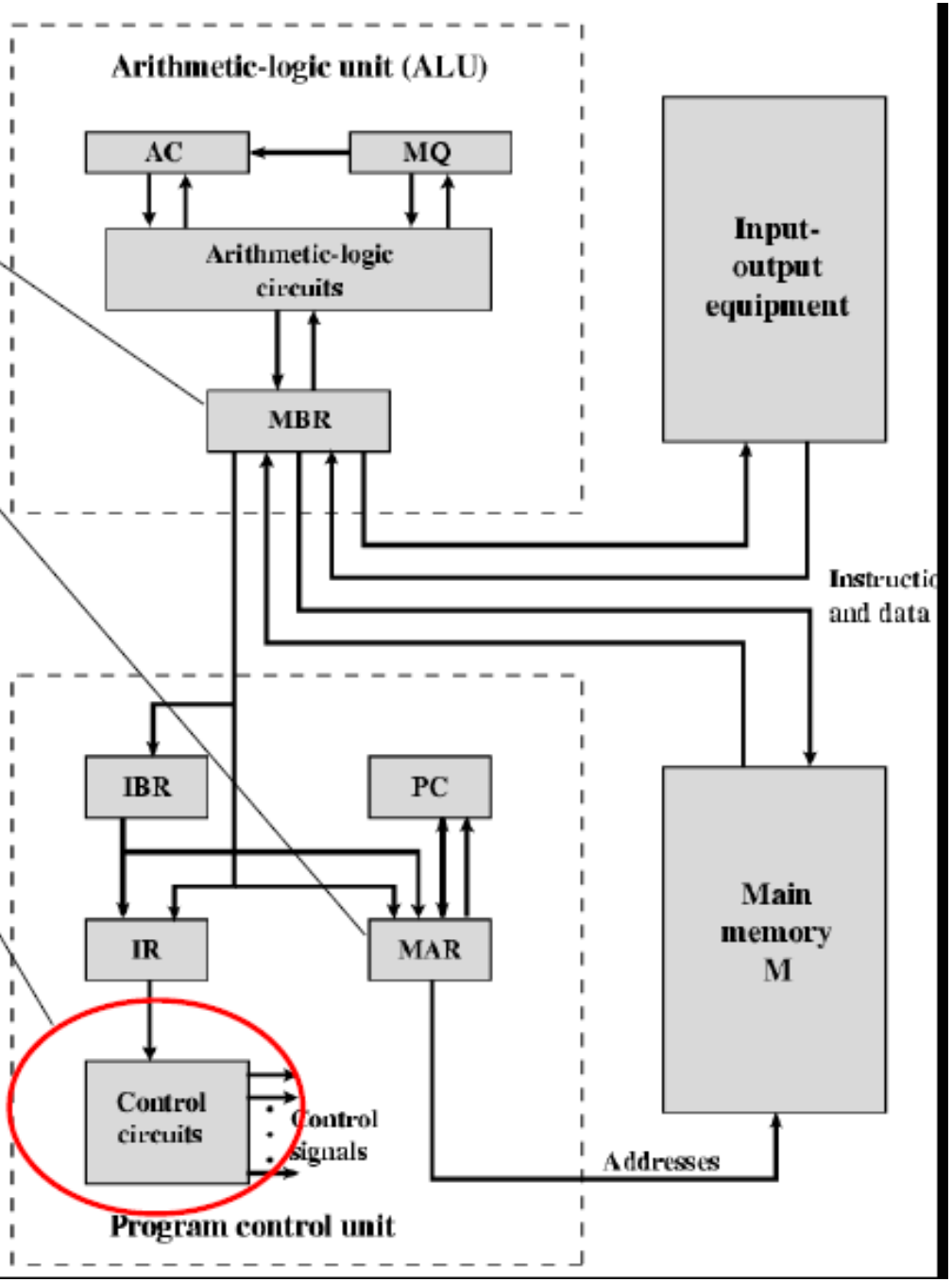
A SPECIFIC SYSTEM EXAMPLE: IAS

IAS organization

Memory Buffer Register either sends data to or receives data from Mem. or I/O

Memory Address Register specifies which Mem. location will be read or written next

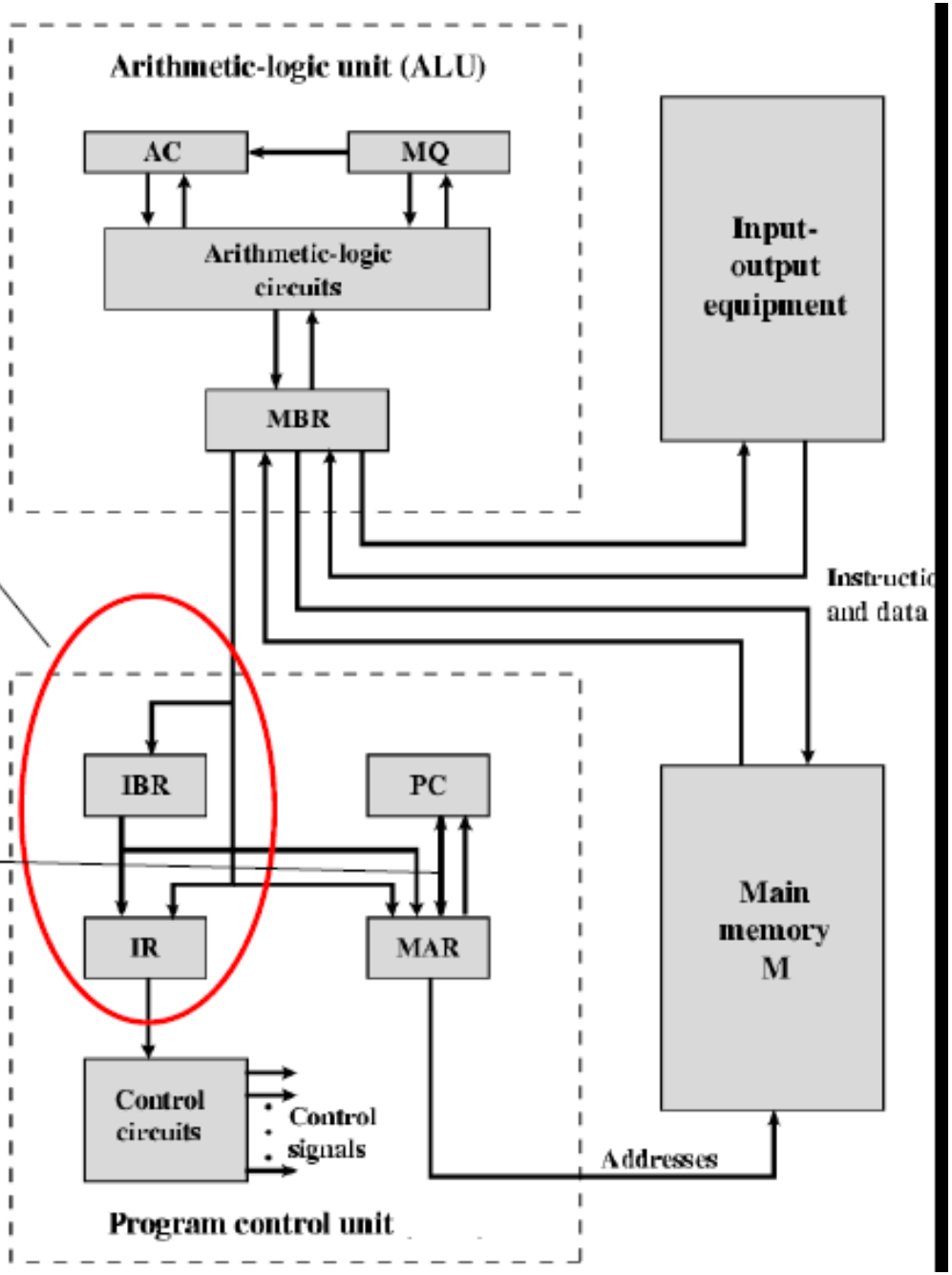
Control signals are set by the opcode part of the instruction bits.
Examples:
- Bring a new instruction from Mem. (fetch)
- Perform an addition (execute)



IAS organization

Why do we need both a register and a buffer register to hold instructions?

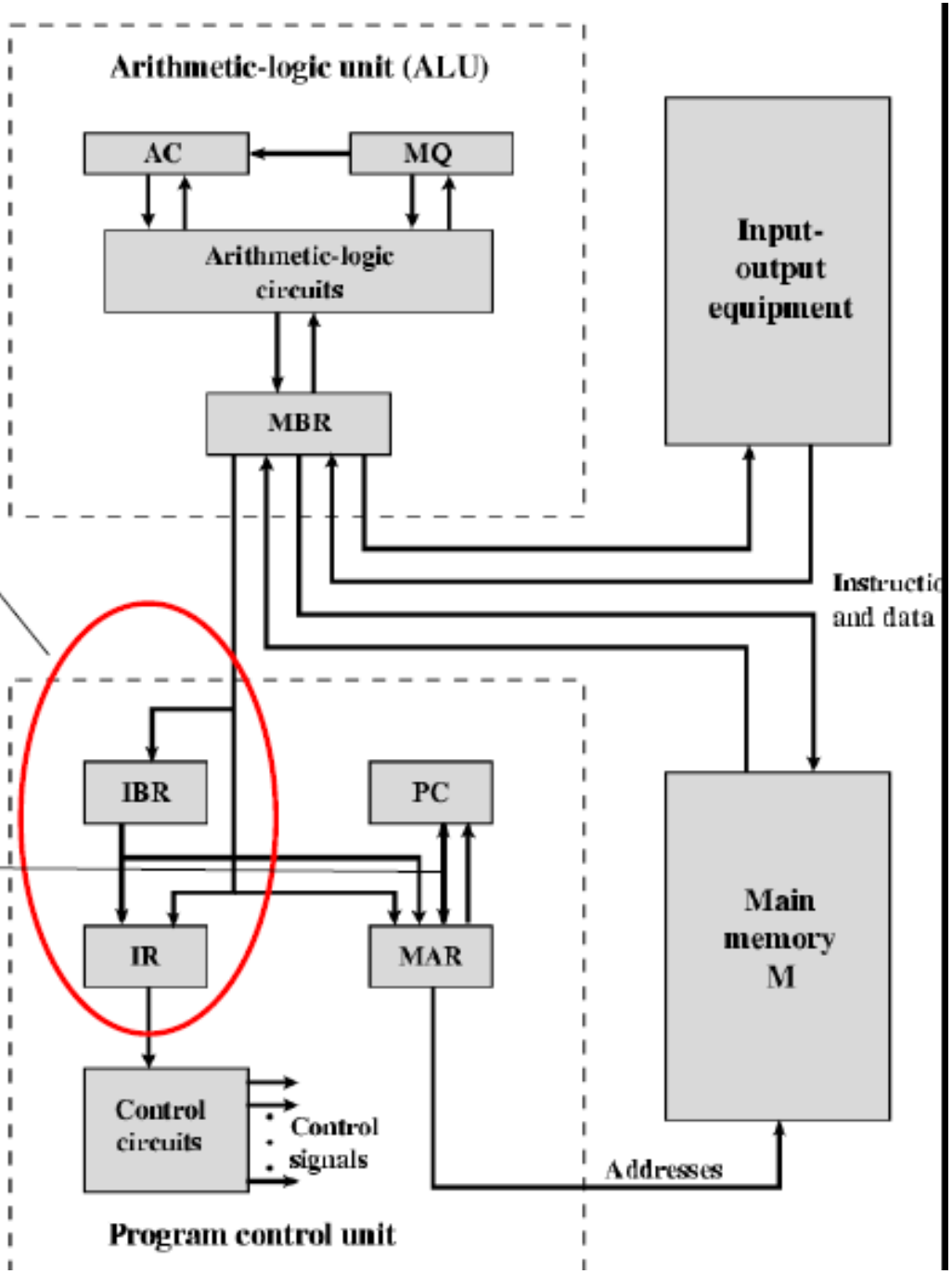
Why does the arrow between PC and MAR point both ways?



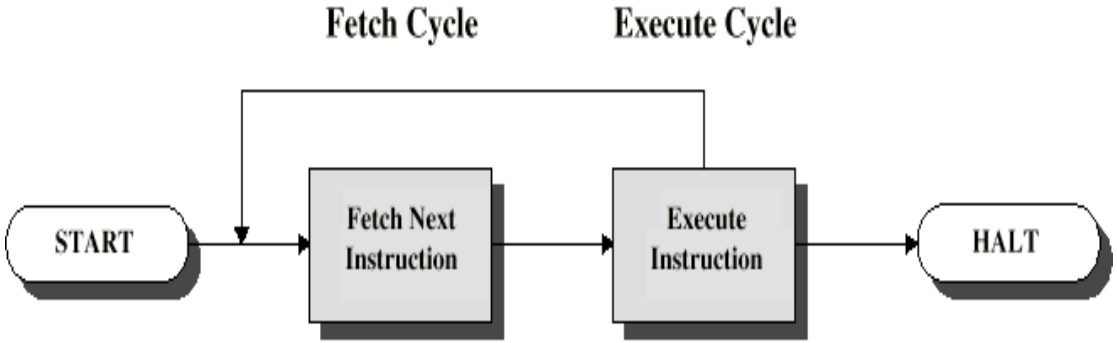
IAS organization

Hint: 2 instructions are stored in each memory word

Hint: the next instruction can be found either sequentially, or through a branch (jump)



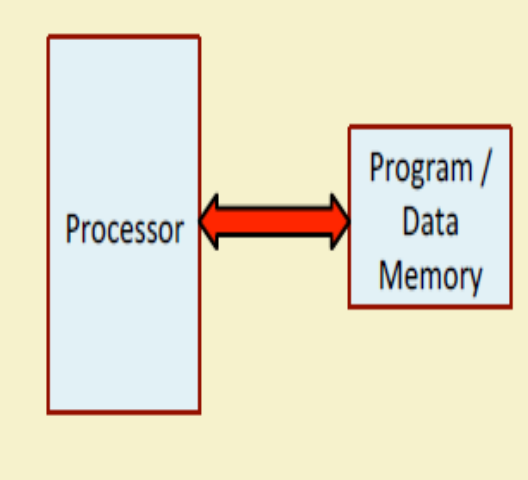
Instruction Cycle



- Two steps:
  - Fetch
  - Execute
- **Fetch Cycle**
  - Program Counter (PC) holds address of next instruction to fetch
  - Processor fetches instruction from memory location pointed to by PC
  - Increment PC
    - Unless told otherwise
  - Instruction loaded into Instruction Register (IR)
  - Processor interprets instruction and performs required actions

von-Neumann Architecture

- Stored Program concept
- Main memory storing programs and data
- ALU operating on binary data
- Control unit interpreting instructions from memory and executing
- Input and output equipment operated by control unit
- More flexible and easier to implement.
- Suitable for most of the general purpose processors.



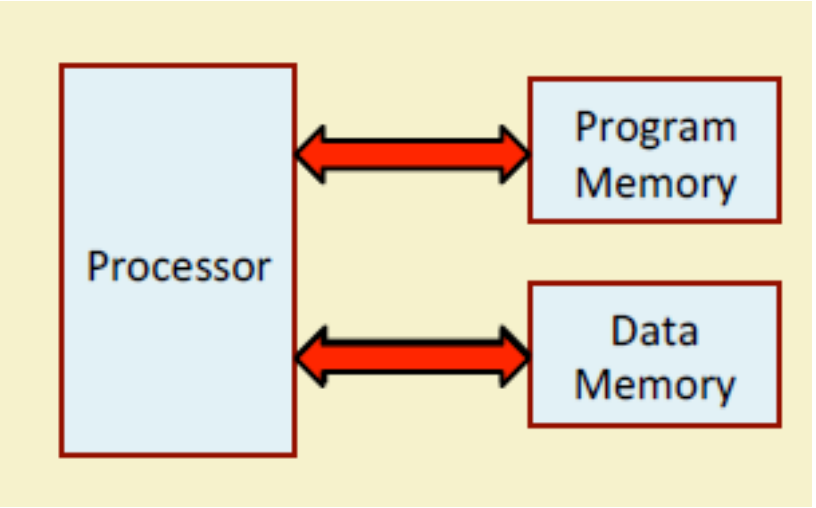
- Disadvantage:
- The processor-memory bus acts as the bottleneck.
- All instructi**ons** and data are moved back and forth through the pipe

von-Neumann Architecture

- **How can this be reduced?**
- This performance problem is reduced by using cache memory.(details in memory part)
- Using RISC architecture as it uses less number of memory reference instruction and uses large number of registers.

Harvard Architecture

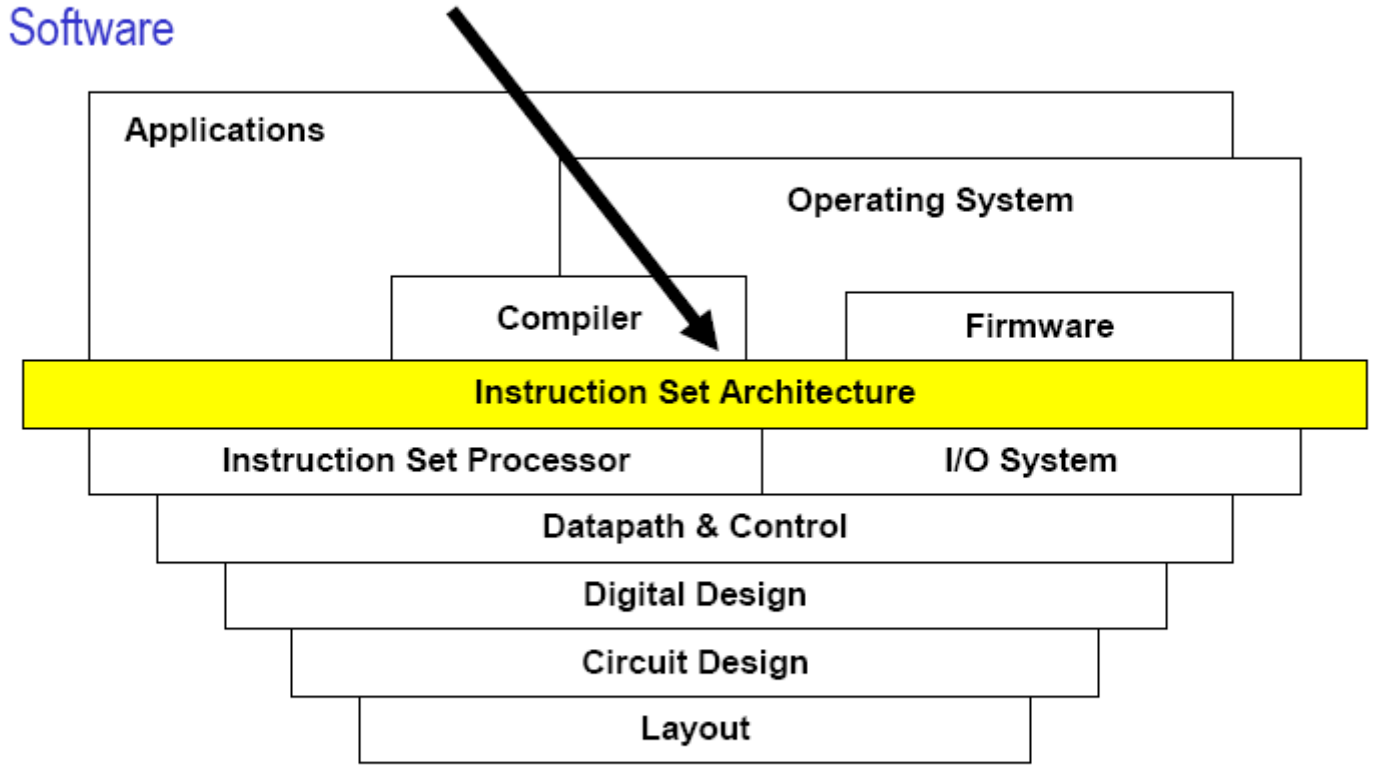
- Separate memory for program and data.
- Instructions are stored in program memory
- Data are stored in data memory.
- Instruction and data accesses can be done in parallel.
- Some microcontrollers and pipelines with separate instruction and data caches follow this concept.
- The processor-memory bottleneck remains.



Instruction Set & Addressing

- Various addressing modes
- Machine Instruction
- Instruction Format

Interfaces in Computer Systems



Instructions: Language of the Computer

To command a computer's hardware
You must speak its language

- **Instruction Set**
  - All Instructions that a computer hardware understands

Example:

1. Instruction Set from MIPS Technologies similar to ARMv7 instruction set(address size 32 bits) and ARMv8 instruction set (Address size 64 bit)
   a) ARM processor instruction set widely used in personal mobile devices
   b) Large share of embedded core market uses MIPS processor instruction set
      - Applications in consumer electronics, network/storage equipment, cameras, printers
2. Instruction set of Intel x86 dominated PC era and so far dominates cloud computing

Ref: Computer Organization and Design , Patterson and Hessessy -5th Ed

Instruction Set

- Different computers have different instruction sets
  - many aspects are common

(Note: Computer Designers' goal:

Choose instruction set

- That makes **it easy to build supporting hardware**
- For which it is **easy to write compiler** while **maximizing performance** and **minimizing cost and energy**

)

Computer

- **Must have** instructions capable of performing **4 types of instructions**
  - **Arithmetic and logical operation on data**
  - **Data transfer between the memory and processor registers**
  - **Program sequencing and control**
  - **I/O transfers**

Operations of Computer Hardware & corresponding Instruction

- Every computer must perform **arithmetic**
  - Every **computer Instruction set has arithmetic instructions**

In High Level Language (say C)

add 2 variables int A,B and assign the sum to

variable int C : - C = A + B;

Notation of Content of Variable

- High Level Language variable A,B,
  - After compilation a table keeps the address of the memory location where the value of the variable is found

  Symbol Table:

Variable	**Address**
A	00100……..10

In Register transfer language **content of variable A** given by **[A]**

Arithmetic Instruction Example contd.....

- Add three operands
  - Two sources variables A and B , one destination variable C

```
add C, A, B  # C <- [A] + [B]
```

- **All arithmetic operations (add,sub,mult,div) have this form**

(Design Principle 1: Simplicity favours regularity

  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost)

Register Operands(General Example)

- General purpose 32 registers within processor with symbolic names
  - R0, R1, R2…..R31

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled general code :

```
add R1, g, h   # R1 = [g] + [h]
add R2, i, j   # R2 = [i] + [j]
sub f, R1, R2  # f = [R1] – [R2]
```

Instruction Type: 3 address Instruction

Register Operands using **MIPS** assembly code

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Two Address Instruction

Register Transfer Language

Operation Source, Destination

Add A, B # B <- [A] + [B]

Disadvantage:

Value of B changed after execution. Previous value of B destroyed. To ensure B does not change; transfer value of B to another variable C

Move B, C # C <- [B]....copy value in B to C variable

Add A,C # C <- [C]+ [A]

Computer

- **Must have** instructions capable of performing **4 types of instructions**
  - **Arithmetic and logical operation on data**
  - **Data transfer between the memory and processor registers**
  - **Program sequencing and control**
  - **I/O transfers**

Data transfer between the memory and processor registers

- Symbolic names stand for hardware binary address
  - LOC,A,B,
  - Processor register names R1, R2,R5…
  - I/O register names DATAIN, OUTSTATUS,DATAOUT

Mov R1, A # A <- [R1] same as :- Store R1,A

Move B, R5 # R5 <- [B] same as:- Load B,R5

Processors where **arithmetic operations** allowed

only on operands that are in processor registers

HLL :- C=A+B

Move A, Ri

Move B, Rj

Add Ri, Rj

Mov Rj , C

These processors much faster (since register access is fast) than Processors that allow arithmetic operation with memory access

One Address Instruction

- Computer with special register ACCUMULATOR (AC) use One Address Instruction

Load A # AC <- [A]

Add B # AC <- [AC] + [B]

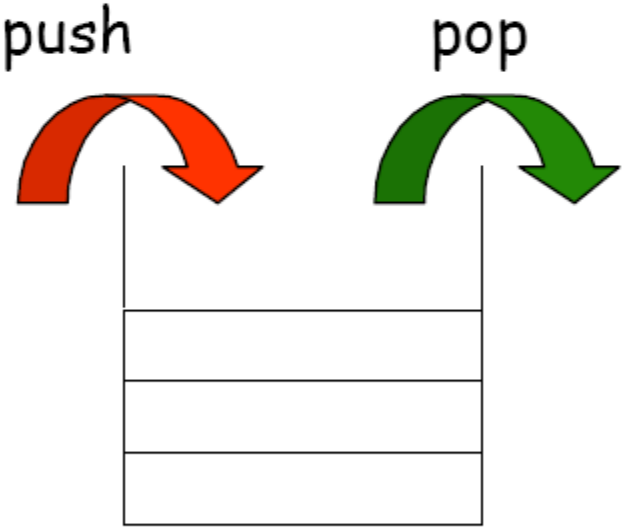
Store C # C <- [AC]

3-address instructions

```
ADD x, y, z  (x:= y+z)
```

0-address instructions (for stack machines)

```
PUSH y   (on a stack)
PUSH z   (on a stack)
ADD
POP x
```



Exercise on Various number of Address Instruction

- Evaluate the arithmetic statement

X=(A+B)/(C-D) using 3 address, two address, one address and zero address instructions

1.Three Address Instruction:

Sub Ri, C,D # Ri <- M[C] –M [D]

Add Rj, A ,B # Rj <- M[A] + M[B]

Div X, Rj,Ri # X <- [Rj] / [Ri]

Exercise on Various number of Address Instruction contd..

2.Two Address Instruction:

```
Load R1, C    #    R1 <- M[C]
Sub R1, D     #    R1  <-  [R1] - M[D]
Load R2, A    #    R2 <- M[A]
Add R2, B     #    R2 <-  [R2] + M[B]
Div  R2,R1    #    R2  <- [R2] / [R1]
Store X,R2    #    M[X]  <-  [R2]
```

Exercise on Various number of Address Instruction contd...

3. One Address Instruction(with accumulator AC):

```
Load  C    #    AC <- M[C]
Sub  D     #    AC  <-  [AC] - M[D]
Store T    #    M[T] <-  [AC]
Load  A    #    AC <- M[A]
Add  B     #    AC <-  [AC] + M[B]
Div  T     #    AC  <- [AC] / M[T]
Store X    #    M[X]  <-  [AC]
```

Exercise on Various number of Address Instruction contd…

4. Zero Address Instruction(stack based processor, stack pointer register (SP)):

```
push  C    #  [SP] <- M[C]
push D    #   [SP] <- M[D]
Sub
push  A    # [SP] <- M[A]
push  B     # [SP] <- M[B]
Add
Div
Pop X       # M[X] <- [SP]
```

Computer

- **Must have** instructions capable of performing **4 types of instructions**
  - **Arithmetic and logical operation on data**
  - **Data transfer between the memory and processor registers**

  Some idea given
  - **Program sequencing and control**
  - **I/O transfers**

  To do

Instruction Set Architecture (ISA)

- **Serves as an interface between software and hardware.**
- **Consists of information regarding the programmer's view of the architecture (i.e. the registers, address and data buses, etc.).**
- **Also consists of the instruction set.**
- **Many ISA's are not specific to a particular computer architecture.**
- **They survive across generations.**
- **Classic examples: IBM 360 series, Intel x86 series, etc.**

- Different Hardware can support same Instruction set
  - Just need a hardware to execute each instruction of the instruction set

Instruction Formats

- **Layout of bits in an instruction**
- **Includes opcode**
- **Includes (implicit or explicit) operand(s)**
- **May be more than one instruction format (different address modes, different number of) in an instruction set (complex instruction set computer CISC)**

Instruction Length

- **Affected by and affects:**
  - **Memory size**
  - **Memory organization**
  - **Bus structure**
  - **CPU complexity**
  - **CPU speed**
- Instruction may be 2 or 3 words long in Complex Instruction Set Computer

Operand Addressing

- *operands* and *operations*
  - *machine instructions*

- address of an operand specified, and b*its* of an instruction organized
  - to fit into the word size
  - Word size limit set by word size that the processor can handle.

Addressing Modes:

- how does the *control unit* determine which addressing mode is being used in a particular instruction ?
  - different *opcodes* used for different addressing modes
  - or one or more bits in the *instruction format* used as a *mode field (keeps code for addressing mode)*.

Effective Address

- What is the interpretation of *effective address*
  - In a system without virtual memory
    - effective address
      - either a main memory address or a register.
  - In a virtual memory system
    - effective address

      is a virtual address or a register

      The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

Addressing Modes:

- To explain the addressing modes, we use the following notation:
- **A**=contents of an address field in the instruction that refers to a memory
- **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 location X

Addressing Modes:

- Common addressing modes are:
  - Immediate
  - Direct
  - Indirect
  - Register
  - Register Indirect
  - Displacement
  - Stack

1. Immediate Addressing

- Operand is part of instruction
- Value of operand given explicitly
- e.g. ADD #5, R1 or ADDI 5,R1
  - Add 5 to contents of register R1
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

Immediate Addressing

- HLL constant value and address contant value translated to this assembly code
- A=B+6

  Move B, R1

  Add #6,R1 (for decimal 6) or

  Add #%00000110,R1 (binary constant) or

  Add #$06,R1 (Hexadecimal constant)

  Move R1 , A

Immediate Addressing

- used to define and use constants
  - to increment counters
  - to test for some patterns
  - or set initial values of variables

Immediate Addressing:

- The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand.
- The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

2. Direct Addressing:

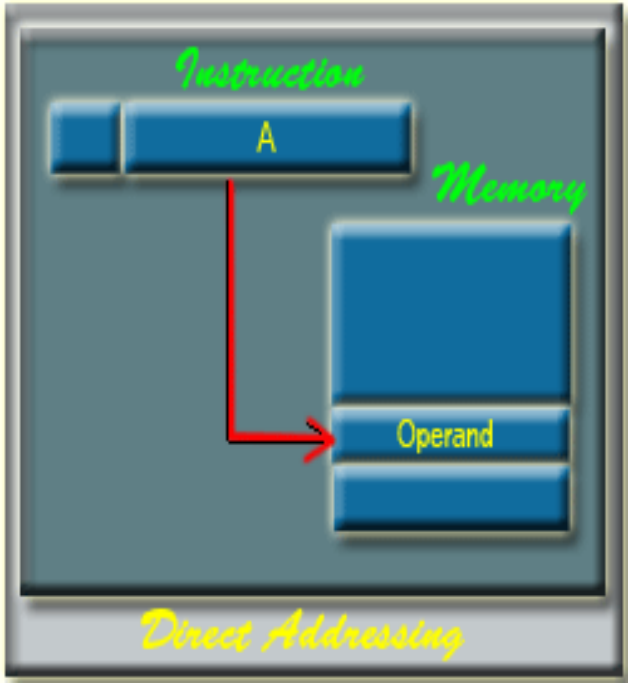
- the address field of operand contains the tive address of the operand:
- EA = A
- It requires only

one memory reference

and no special calculation.

- Here, 'A' indicates the me

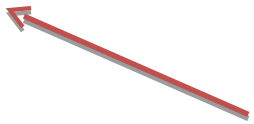
address for the operand.



Addressing Modes (Used in Assembly Language)

- 2.Absolute Mode or Direct Mode:
  - Name of operand used explicitly

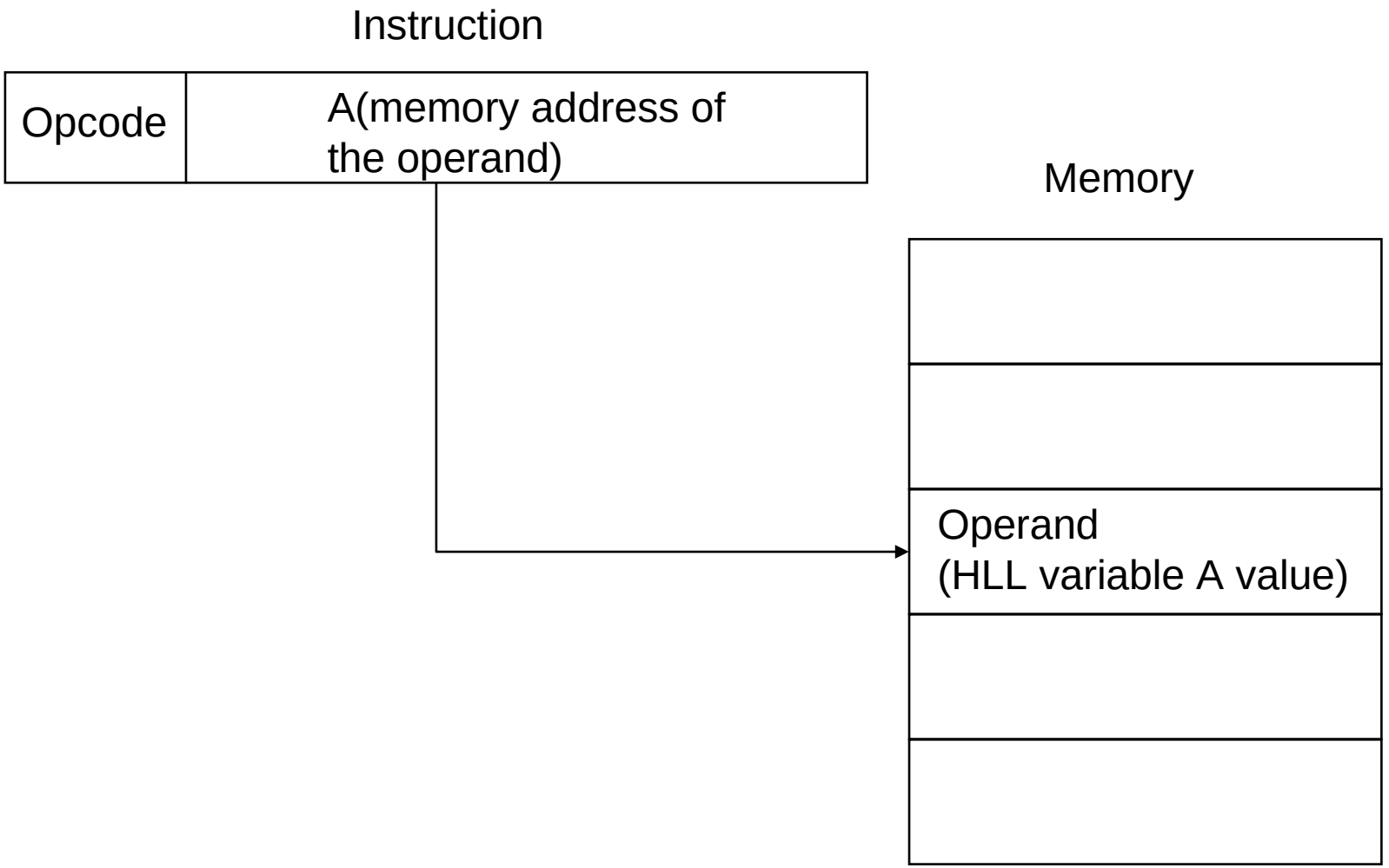
Move B , R2



HLL int A,B;

Compiler assigns memory location A and B to the variables

Absolute or Direct Addressing Diagram



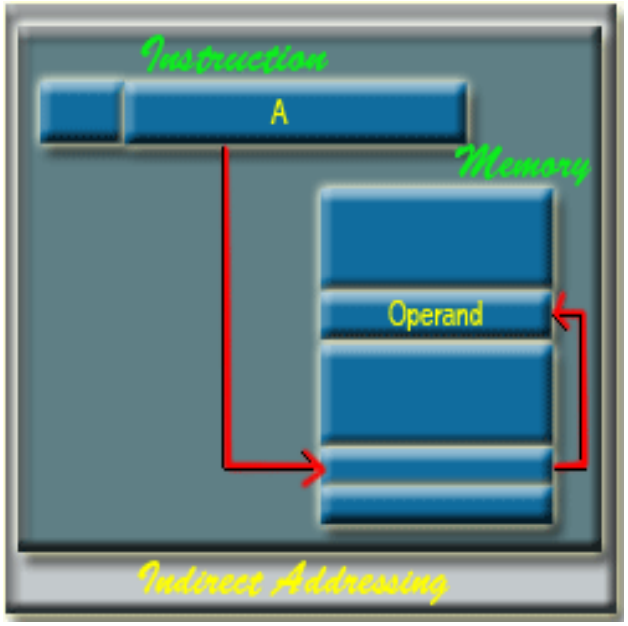
Absolute or Direct Addressing

- Disadvantage:
  - Number of bits available for memory address of operand is less than number of bits available in total word

3.Indirect Addressing:

- With direct addressing
  - length of the address field is less than the word length, thus limiting the address range
- Indirect (Pointer) addressing
  - address field refer to the address of a word in memory, which in turn contains a full-length address of the operand
- EA = (A)

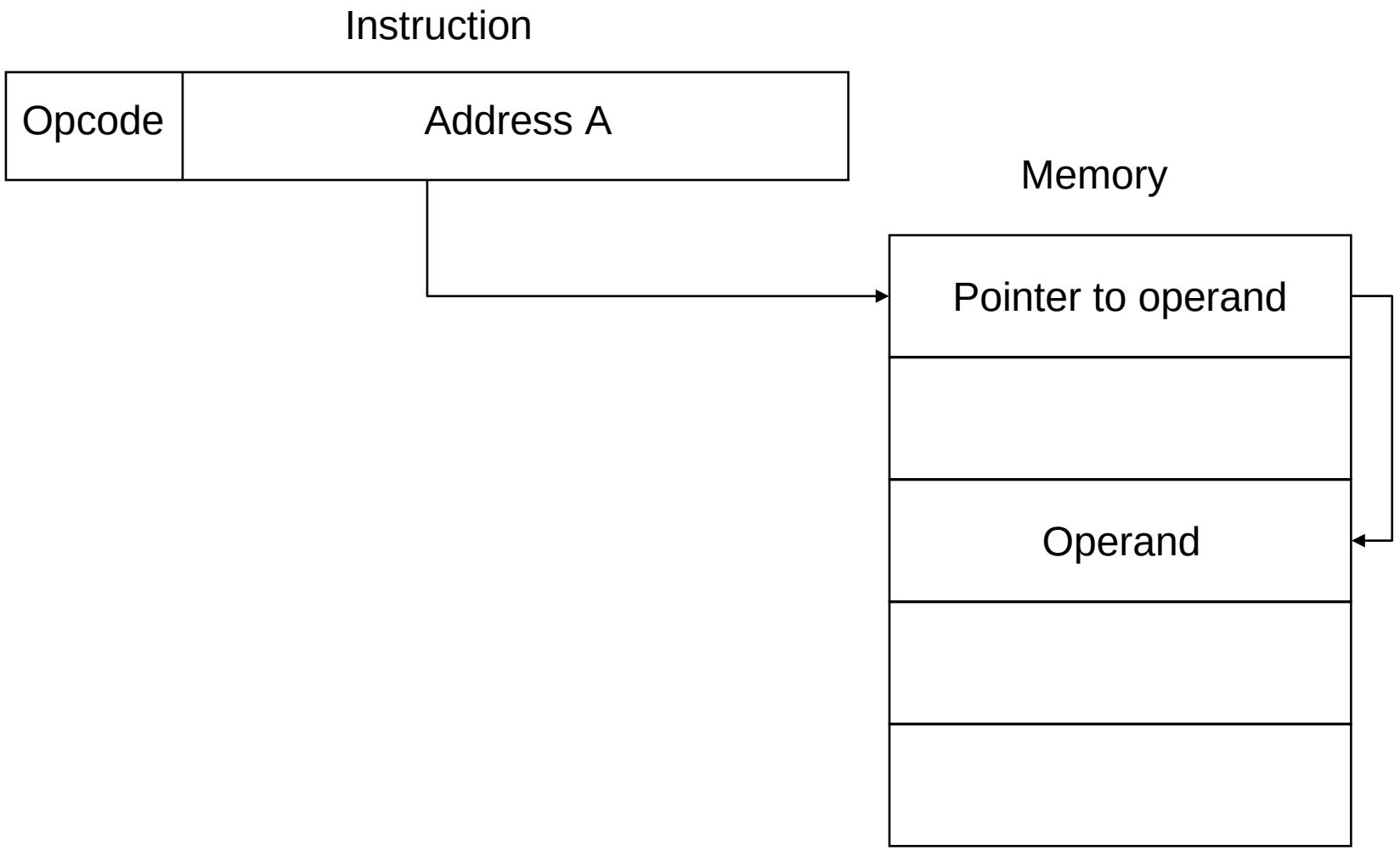
Here 'A' indicates the memory address which contains the address(one word size) of the required operands.



Indirect Addressing

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- EA = (A)
  - Look in A, find address and contents of the address in A is the operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to accumulator

Indirect Addressing Diagram



Indirect Addressing Mode

HLL A=*B;

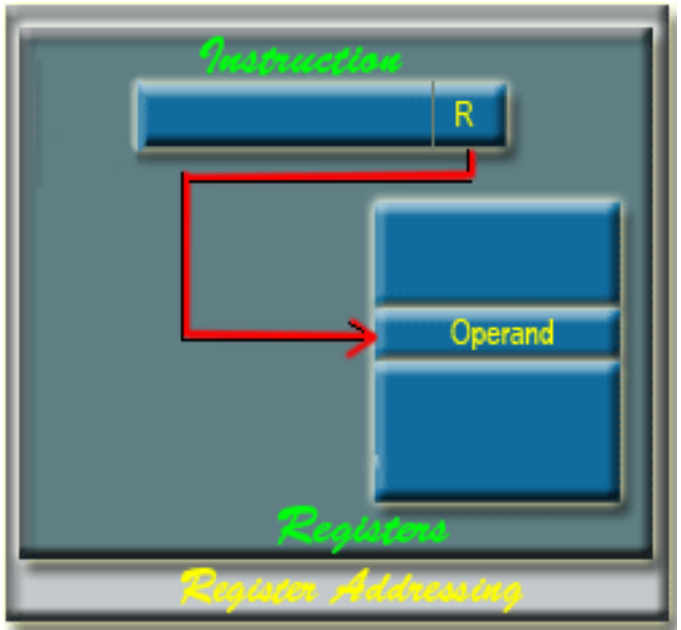
- May compile into :

Move B,R1

Move (R1),A

4. Register Addressing:

- address field refers to a register rather than a main memory address:
- EA = R
- The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required.
- The disadvantage of register addressing is that the address space is very limited.
- 'R' indicates the address of the register where the operand is present.



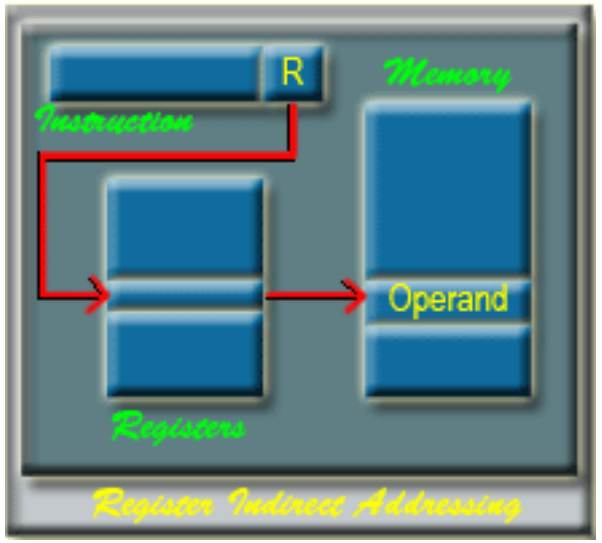
Register Mode

- Operand is the content of processor register
  - In this example R2,gives the address of the register, (if 32 registers then address needs 5 bits only)
  - Name of the processor register keeps the address of the register where the value of the operand found

Move B , R2

5. Register Indirect Addressing:

- address field refers to a register instead of a memory location, therefore 5-7 bits required(for 32 or 64or 128 registers)
- EA = (R)
- Register indirect addressing uses one less memory reference than indirect addressing. Because, the address of the memory location is available in a register (Memory not accessed to get address of the operand in memory)
- register access is much more faster than the memory access.



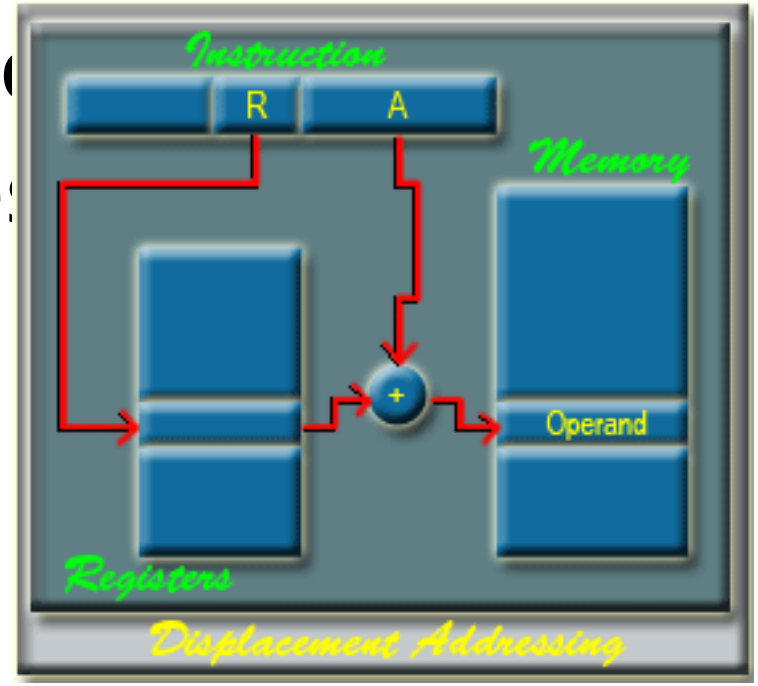
Displacement Addressing:
Relative addressing(PC) , Index Addressing/ Base-register addressing(Arrays)

- A(R)
  - Displace by offset A the address kept in register R
- EA = A + (R)

Displacement Addressing:

- Three of the most common use of displacement addressing are:
- Relative addressing
- Index Addressing M
- Base-register addre

EA= A +(R)



5. **Relative Addressing:** (Displacement Addressing)

- X(PC)
  - X (bytes offset) added to current instruction address (PC register value) to produce Effective address

EA= X + (PC)..address of the next instruction

- X(PC)…X bytes offset can be

–tive (going back in instruction sequence)

or +tive (going forward in instruction sequence)

Relative Addressing

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e. get operand from A cells away from current location pointed to by PC
- c.f locality of reference & cache usage

Relative Addressing

1000 Loop	
1004	
1008	
1012	
PC → 1016	Branch > 0 - 16(PC)

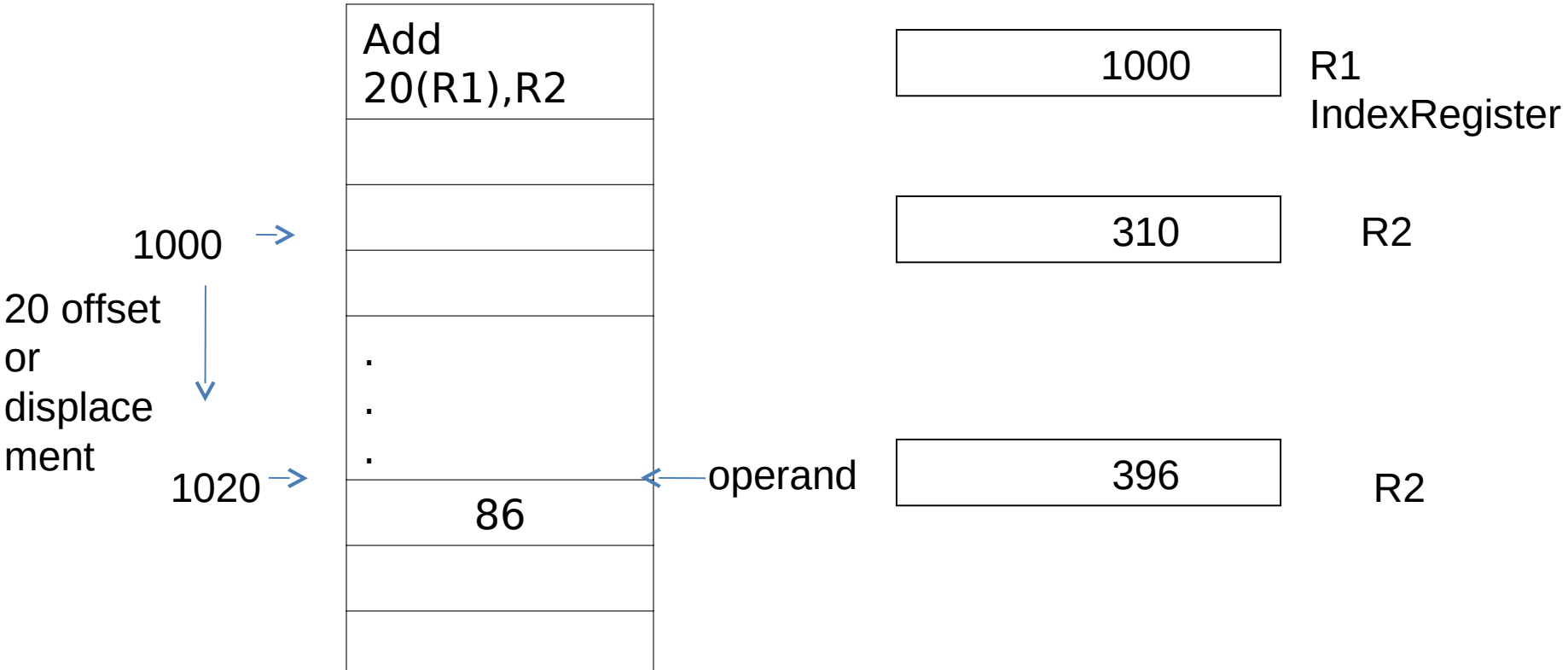
6. Index Addressing Mode

used for HLL Lists and Arrays

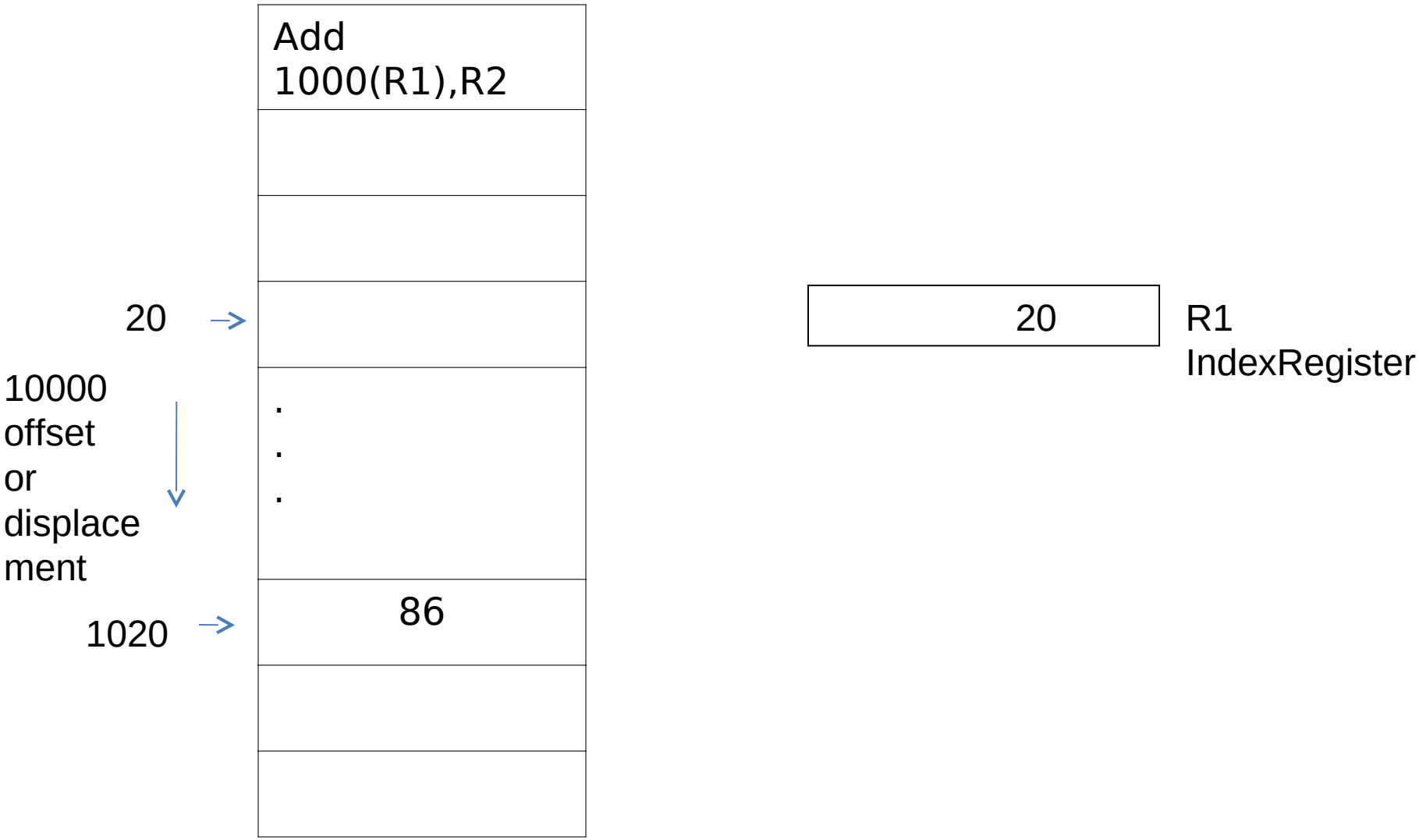
- Effective Address X(R1)

  EA = offset(or displacement)+ index register R1 contents value

  = X + [R1] = 20 + 1000 = 1020



EA = X + [R1] = 1000 + 20 = 1020



7. Base-Register Addressing (Displacement)

- Base Register Rj with Index Register Ri
  - (Ri,Rj)
- EA = (Ri) + (Rj)
- Address field hold two values
  - Rj = base address value
  - Ri = register that holds index(displacement)
  - or vice versa

Base-Register Addressing contd.…:

- **Or vice versa**
- base register (Ri)contains array base memory address
- address field contains Rj contains index/displacement from base address
- EA= (Ri) + (Rj)

Index Addressing Mode Variations

- More complex index modes
- Base register with index register and offset:
- X(Ri,Rj)

$$EA = X + Ri + Rj$$

Index Mode

	Test 1	Test2
Student 1	val1 1	val12
Student 2	val2 1	val22
.. ..	... ..	. ...
N		n

GrandTotal LIST →	STUDENT ID
LIST+4	Test1
LIST+8	Test2
LIST+12	STUDENT ID
	Test1

```
        Move N,R4
        Move #LIST,R0
LOOP    Add  4(R0),R1
        Add  8(R0),R2
        Add  #12,R0
        Decr R4
        Branch >0 LOOP
        Move R1, SUM1
        Move  R2,SUM2
```

8. Autoincrement Mode (Ri)+ Autodecrement Mode -(Ri)

Ri contains address of operand

1.(Ri)+ After accessing operand(Ri), contents of Ri automatically incremented to point to next item in list

2. –(Ri) Contents of Ri decremented by 1 and then used.

Used for stack data structure stack pointer

	Move N,R4
	Move #LIST,R0
LOOP	Add 4(R0)+,R1
	Add (R0),R2
	Add #4,R0
	Decr R4
	Branch >0 LOOP
	Move R1, SUM1
	Move R2,SUM2

Stack Addressing:

- stack is a linear array or list of locations
  - *pushdown list* or *last-in-first-out queue*
  - stack is a reserved block of locations
  - Items appended to the top of the stack so that, at any given time, the block partially filled
  - Associated with the stack is a pointer whose value is the address of the top of the stack
  - **Stack pointer maintained in a register**
    - 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.**

Table 2.1 Generic addressing modes

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*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

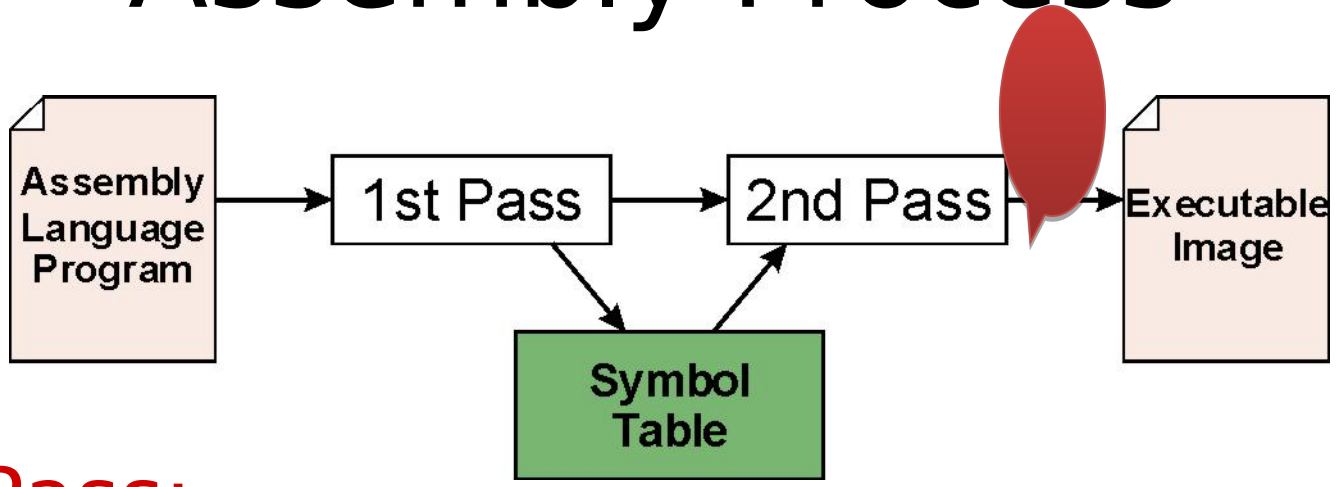
EA = effective address
Value = a signed number

Instruction Types

BASIC INSTRUCTIONS

Data Movement	LOAD, STORE, MOVE
Arithmetic & Logical	ADD, SUB, AND, XOR, SHIFT
Branch	JUMP (unconditional)
	JZ, JNZ (conditional)
Procedure Call	CALL, RETURN
Input Output	Memory-mapped I/O*
Miscellaneous	NOP, EI (enable interrupt)

Assembly Process

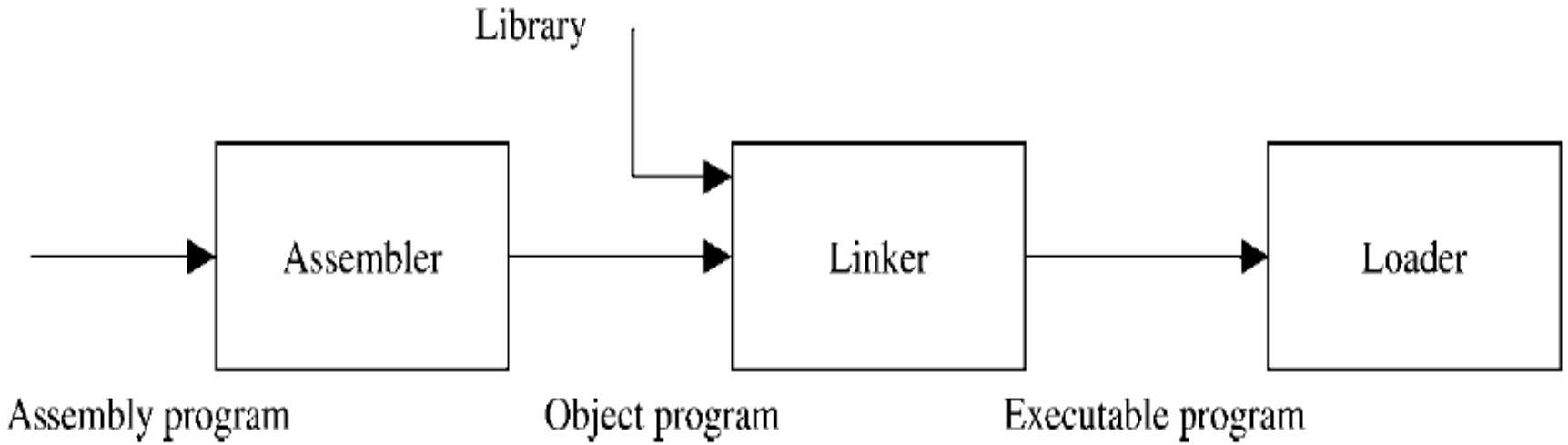


First Pass:

- scan program file
- find all labels and calculate the corresponding addresses; this is called the *symbol table*

Second Pass:

- convert instructions to machine language, using information from symbol table



Assembly and execution process

Linking and Loading

Loading is the process of copying an executable image into memory.

- more sophisticated loaders are able to *relocate* images to fit into available memory
- must readjust branch targets, load/store addresses

Linking is the process of resolving symbols between independent object files.

- suppose we define a symbol in one module, and want to use it in another
- some notation, such as `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module
- linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

Example Addressing Mode

Ref: Mano:- Computer System Architecture

Numerical Example :-Word addressable Accumulator based Computer with Two-word instruction at address 200 and 201

First word of instruction contains op code :"load to AC”instruction,

and mode

Second word of instruction contains operand field equal to 500

- PC has the value 200 for fetching this instruction
- Content of processor register R 1 is 400,
- Content of an index register XR is 100
- AC receives the operand after the instruction is executed
- Following slide figure lists other addresses of memory and their content

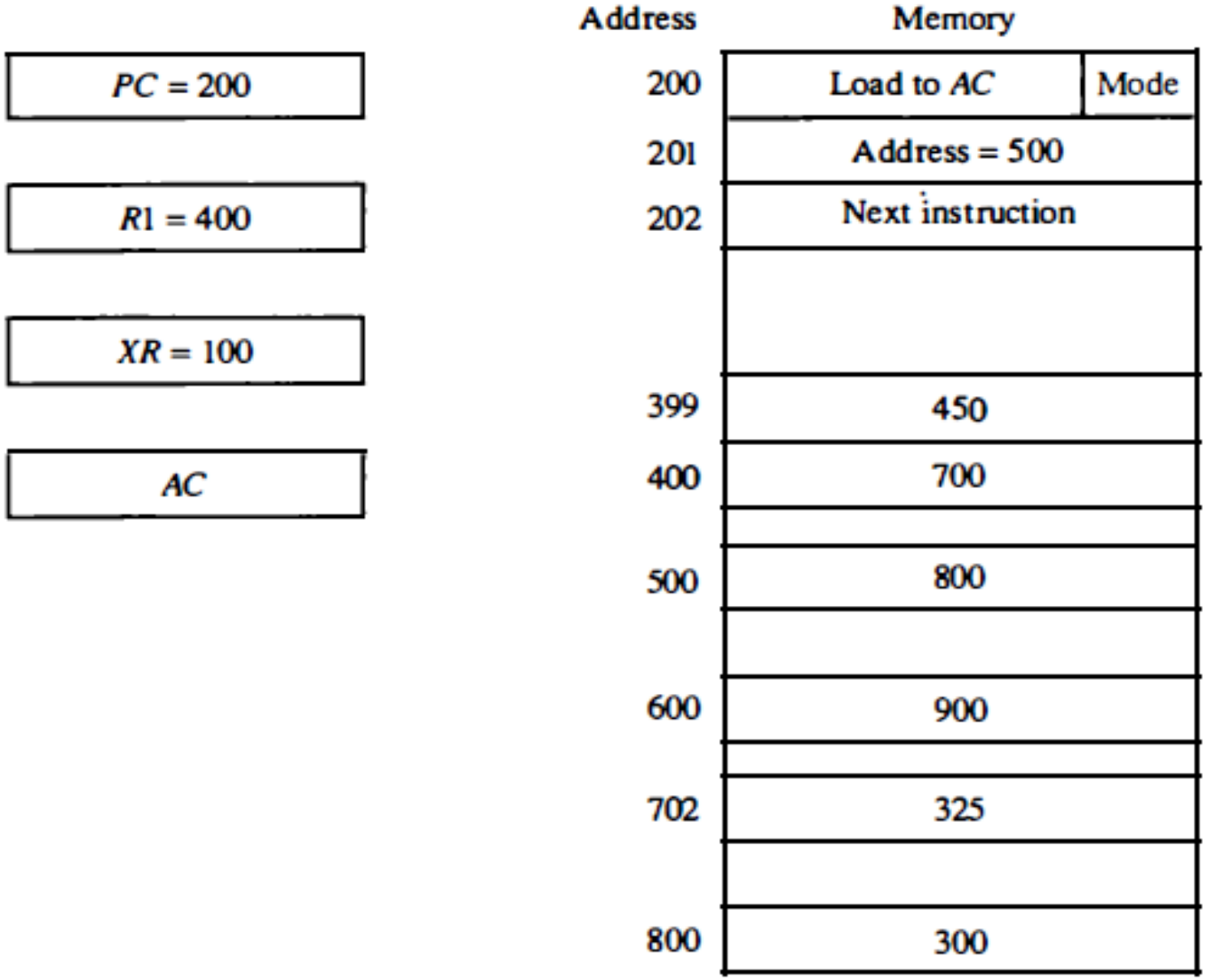


Figure 8-7 Numerical example for addressing modes.

Numerical Example

- Mode field of the instruction can specify any one of a number of modes.
- For each mode we calculate the effective address and the operand that must be loaded into AC .

- 1. In direct address mode
  - the effective address = address part of the instruction =500
  - operand to be loaded into AC is 800

- 2. In immediate mode the second word of the instruction is taken as the operand rather than an address
  - so 500 is loaded into AC (effective address in this case is 201)

- 3.In indirect mode
  - the effective address is stored in memory at address 500. Therefore, the effective address is 800
  - the operand is 300.

Numerical Example(Ref: Mano –Computer System Architecture)

- 4. In the relative mode
  - effective address is 500 + 202 = 702
  - operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202.)

- 5.In the index mode
  - effective address is XR + 500 = 100 + 500 = 600
  - operand is 900.

- 6.In the register mode the
  - operand is in R 1 and 400 is loaded into AC . (There is no effective address in this case.)

Numerical Example(Ref: Mano –Computer System Architecture)

- 7. In the register indirect mode
  - effective address is 400, equal to the content of R 1
  - operand loaded into AC is 700.

- 8. The autoincrement mode is the same as the register indirect mode
  - except that R 1 is incremented to 401 after the execution of the instruction.

- The autodecrement mode
  - decrements R1 to 399 prior to the execution of the instruction.
  - operand loaded into AC is now 450

- Following Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Example Addressing Mode

Ref: Hamacher:- Computer Organization

2.13 Registers R1 and R2 of a computer contain the decimal values 1200 and 4600. What is the effective address of the memory operand in each of the following instructions?

(*a*) Load 20(R1),R5

(*b*) Move #3000,R5

(*c*) Store R5,30(R1,R2)

(*d*) Add −(R2),R5

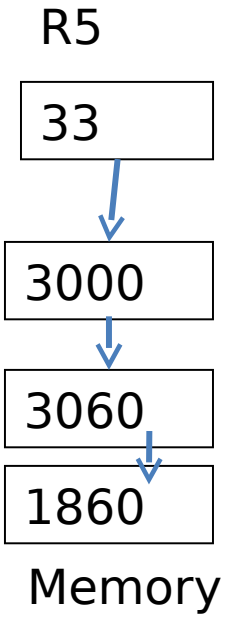
(*e*) Subtract (R1)+,R5

Assuming Word Addressable Processor

R1=1200, R2=4600

119 9

	Effective Address(EA)	Addressing Mode
load 20(R1), R5	1200+20=1220	Index Addressing Mode
move #3000, R5	3000 is the value put into R5	Immediate Addressing Mode
store R5, 30(R1,R2)	30+1200+4600=5830	Base Addressing Mode with index and offset
add –(R2) , R5	4600-1=4599	Auto-decrement mode
sub (R1)+, R5	1200(after the operation of subtracting the operand in 1200 location from R5 value is done the address 1200 incremented to 1201	Auto-increment mode



1220	33
4599	60
. .	
5830	3000