COMPUTER ORGANIZATION AND ARCHITECTURE (IT 2202)

Lecture 7

nstruction Format and Addressing Modes

Instruction comprises of two parts

- First part is Operation or Opcode,
- Next part is the Operand.

Opcode specifies the operation to be performed.

Categories of instructions:-

- data transfer,
- arithmetic, and
- logical control,
- I/O and special machine control.

Example: (ADD R1, R2) is adding two register values, and result stored back in some register. So, R1 will store R1 + R2. ADD is the operation code that specifies the operation to be performed by the instruction. This is an arithmetic instruction.

nstruction Format and Addressing Modes

(MOVE R2, R1) instruction moves the data from R2 to R1. So, R1 will have the value of R2. This instruction is called **data transfer instructions.**

JUMP 8000 is an example of the program control instruction that transfers the control to address 8000. This is **branching instruction**.

Branching to this particular location will move to location 8000, and the location will be added into PC and from this location, the next instruction will be executed because branch to this location means in this particular location, some instruction is present which will be executed.

nstruction Format and Addressing Modes

An operand specifies either a single source or two sources and a destination of the operation. Source operand can be specified by an immediate data or by a register.

Example: (ADDI 2000) is example of immediate data and it is adding 2000 with the content of the Accumulator (A) and the result will be saved in A.

(ADD R1, MEMORY) instruction adds the content of the register with the content of a memory address. Here, one operand is specified in a register, another operand is specified in a memory location. So, a register, a memory location or an immediate value will be used.

A destination operand should always be either a register or a memory location like “MEMORY”.

Instruction Format

- **What Information an Instruction should convey to the CPU?**

Opcode	Addr of OP1	Addr of OP2	Dest Addr	Addr of next intr.

- **Instruction is too long if everything is specified explicitly**
  - **More space in memory**
  - **Longer execution**
- **Can we reduce the size of an instruction?**
  - **Specifying information Implicitly**
  - **How?**
- **Using Program Counter, Accumulator, General Purpose Registers, Stack Pointer**

Addressing Modes

Number of operands varies from instruction to instruction.

Instruction may be

- zero address instruction,
- one address instruction.
- two address
- three address instruction.

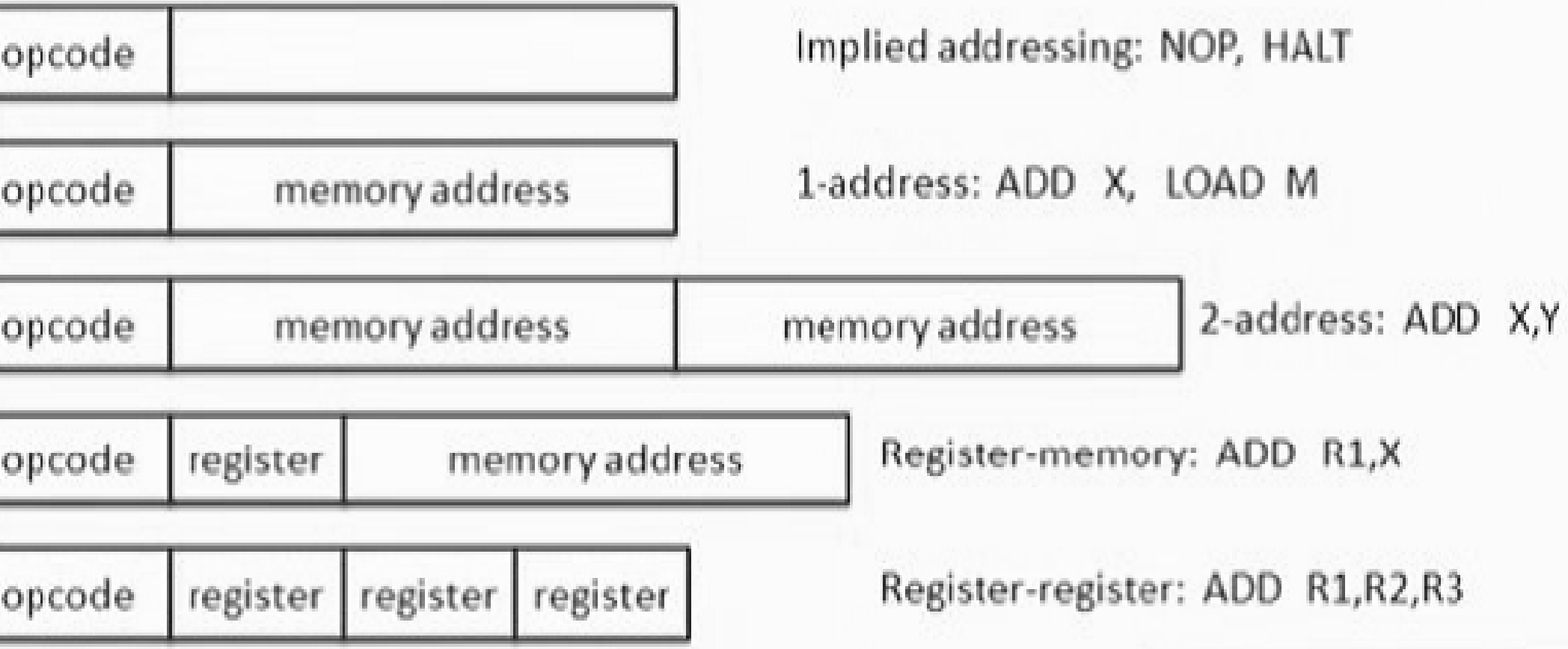
While specifying an operand, the various addressing modes need to be known.

Addressing mode actually is a way by which the location of the operand is specified in the instruction.

We will see how is the address of an operand specified, and how are the *bits* of an instruction organized to define the *operand addresses* and operation of that instruction.

ddressing Modes

nstruction Formats



ddressing Modes

Addressing Modes:

The most common addressing techniques are:

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

ddressing Modes

All computer architectures provide more than one of these *addressing modes*.

Control unit can determine which addressing mode is being used in a particul instruction. Several approaches are used.

Different *opcodes* will use different addressing modes.
One or more bits in the *instruction format* can be used as a *mode field*. The value of t mode field determines which addressing mode is to be used.

In a system without virtual memory, the effective address will be either a main memo address or a register.
In a virtual memory system, the effective address is a virtual address or a register.

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

ddressing 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

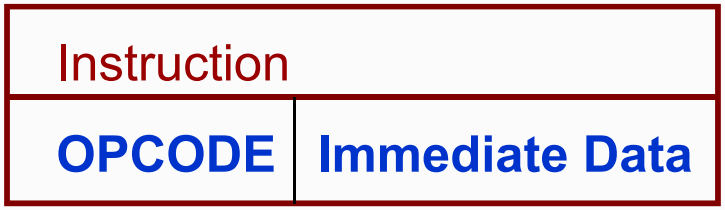
(X) = contents of location X

ddressing Modes

mmediate Addressing:

Simplest form of addressing, in which the operand is actually present in the instructio
In immediate addressing mode, the operand is part of the instruction itself.

OPERAND = Immediate data



This mode can be used to define and use constants or set initial values of variables.

Advantage of immediate addressing is that no memory reference other than t
instruction fetch is required to obtain the operand.

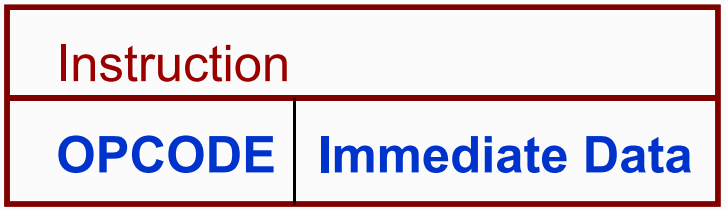
Disadvantage is that the size of the number is restricted to the size of the address fiel
which, in most instruction sets, is small compared with the world length.

ddressing Modes

mmediate Addressing:

Examples:

```
ADD #25          // ACC=ACC+25
ADDI R1,R2,42    // R1=R2+42
```



No memory access is required to get the operand and it is fast, but limited rang because we can only specify a limited number using immediate mode like ADD #25.

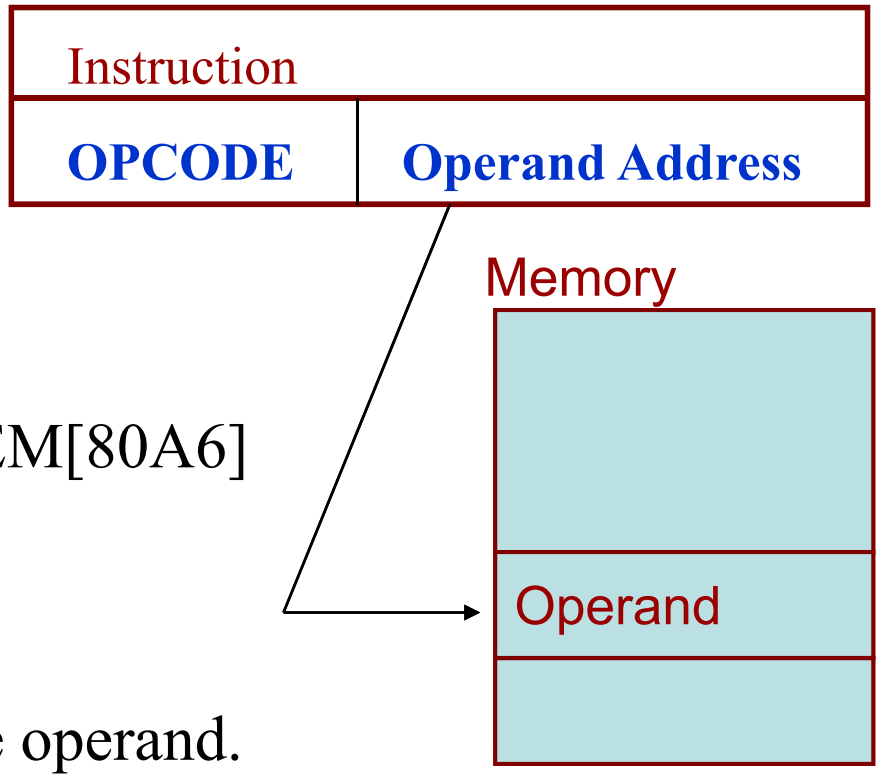
When we write #, it means is an immediate data. When we write ADD #25 that mean 25 will be added with accumulator and the result will be stored back in the accumulato Here, we have an immediate data 42, which is added to R2 and result stored in R1.

ddressing Modes

rect Addressing:

A very simple form of addressing is direct addressing, in which the address field contain
the effective address of the operand: EA = A

It requires only one memory reference and
no special calculation.



Examples:

- ADD R1, 80A6H // R1=R1+MEM[80A6]

Single memory access is required to access the operand.
No additional calculations required to determine the operand address.
Limited address space (as number of bits is limited, say,16 bits)

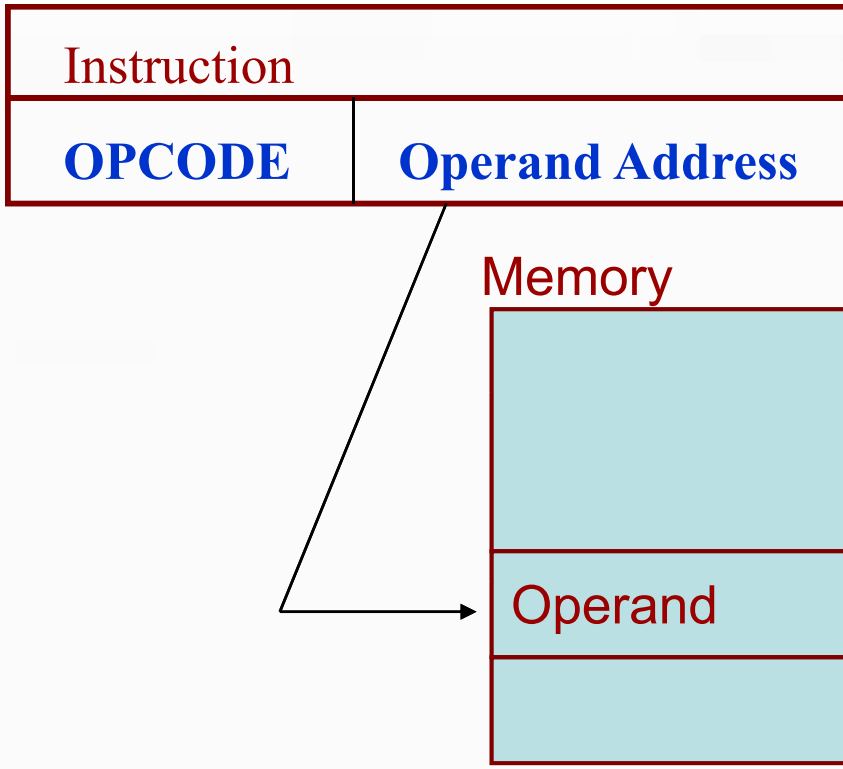
ddressing Modes

Direct Addressing:

ADD R1, 80A6H

Instruction contains a field that holds the memory address of the operand.

ADD R1,80A6 means that the content of 80A6 will be added with R1 and result will be stored in R1.



One memory access is used to access the operand and no additional calculation i required to determine the operand address.

16- bit address directly accesses the operand.

ddressing Modes

Register Addressing

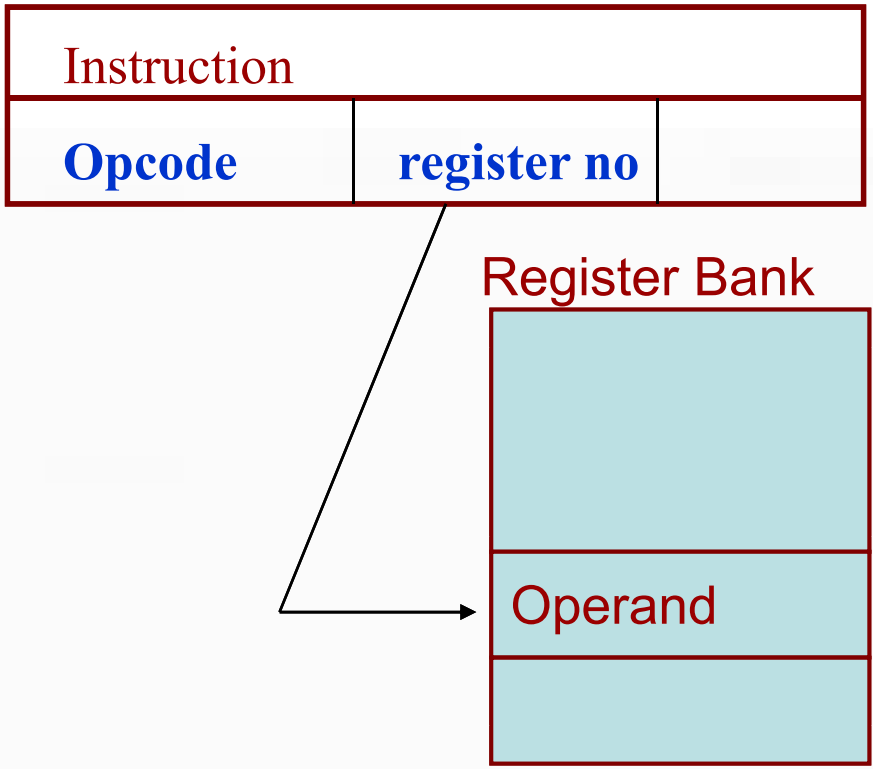
is similar to direct addressing.

Difference is that the address field refers to a register rather than a main memory address:

EA = R

Advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required.

Disadvantage of register addressing is that the address space is very limited.



ddressing Modes

egister Addressing

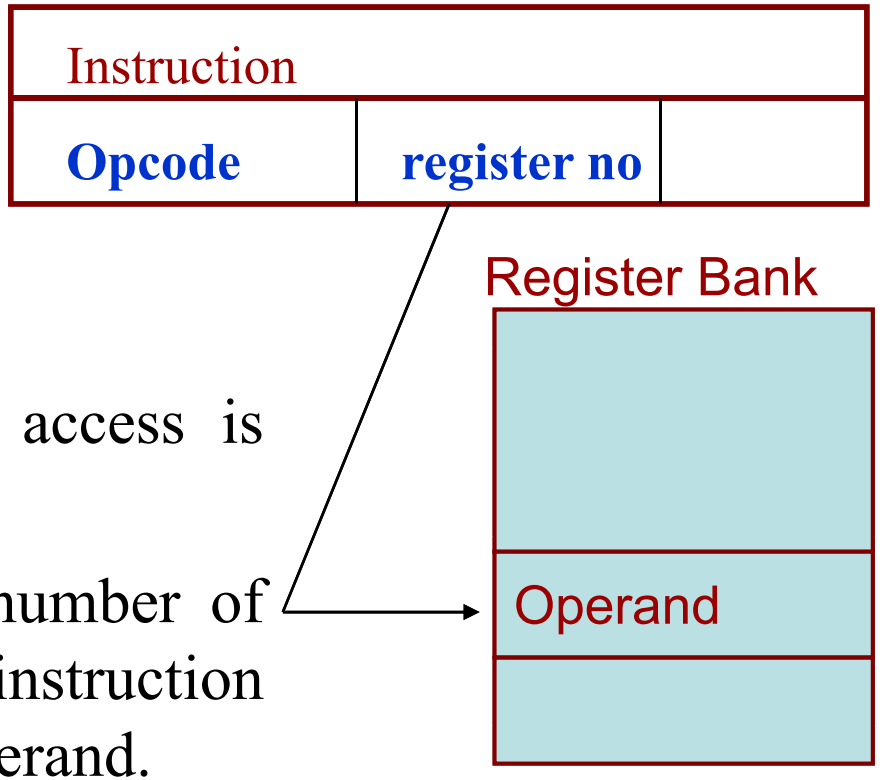
Operand is held in a register.

Instruction specifies the register number.

Very few bits are needed, as the number of registers is limited.

Faster execution is possible, and no memory access is required for getting the operand.

Modern load-store architecture supports large number of registers. The register number is specified in the instruction and you go to that particular register to get the operand.



xamples

```
ADD R1, R2, R3          // R1=R2+R3
MOV R2,R5               // R3=R5
```

ddressing Modes

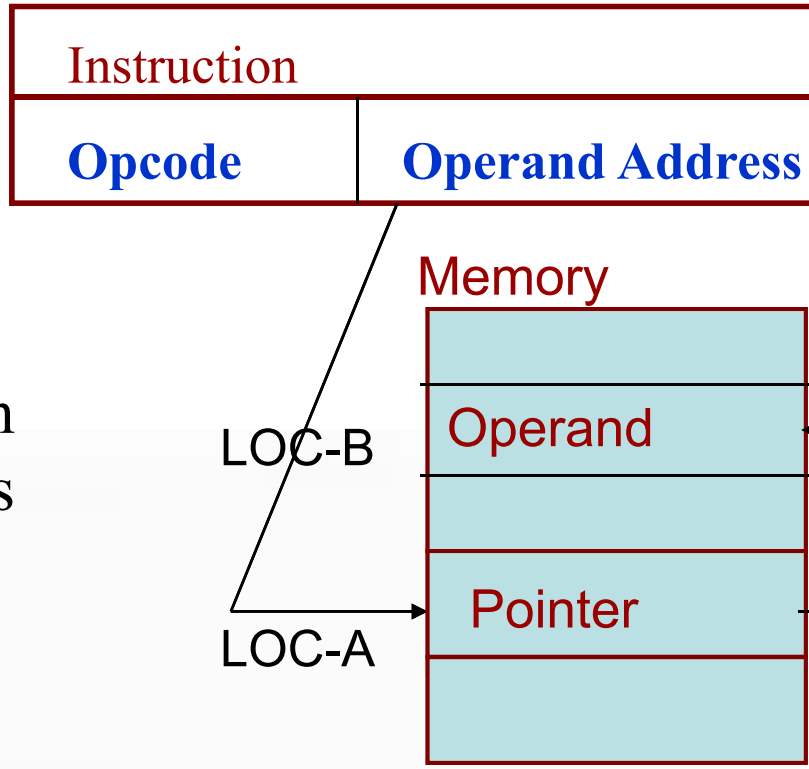
ndirect Addressing

Length of address field is usually less than the word length. It limits address range.

Address field 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)

Instruction contains a field that holds the memory address which in turn holds the memory address of the operand.



ddressing Modes

Examples:

ADD R1,(40A6H) // R1=R1+(mem[40A6])

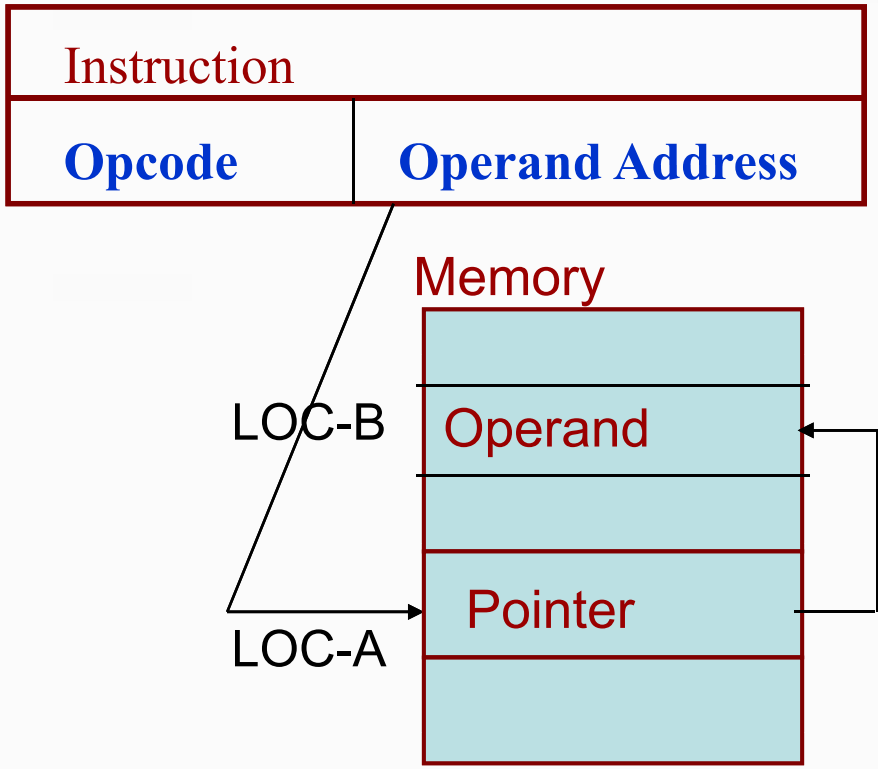
An instruction ADD R1,(LOC-A).

Location (LOC-A) contains another address LOC-B.

We will not get the operand from LOC-A, rather we will get operand from LOC-B.

Here, we have to go to the location LOC-A and LOC-A location will give another location LOC-B and this location will give the value.

Here, two memory accesses are required to get the operand value.



This is slower, but can access larger address space. It is not limited to number of bits in the operand address like direct addressing.

LOC-A points the address (LOC-B) the operand.

ddressing Modes

egister Indirect Addressing

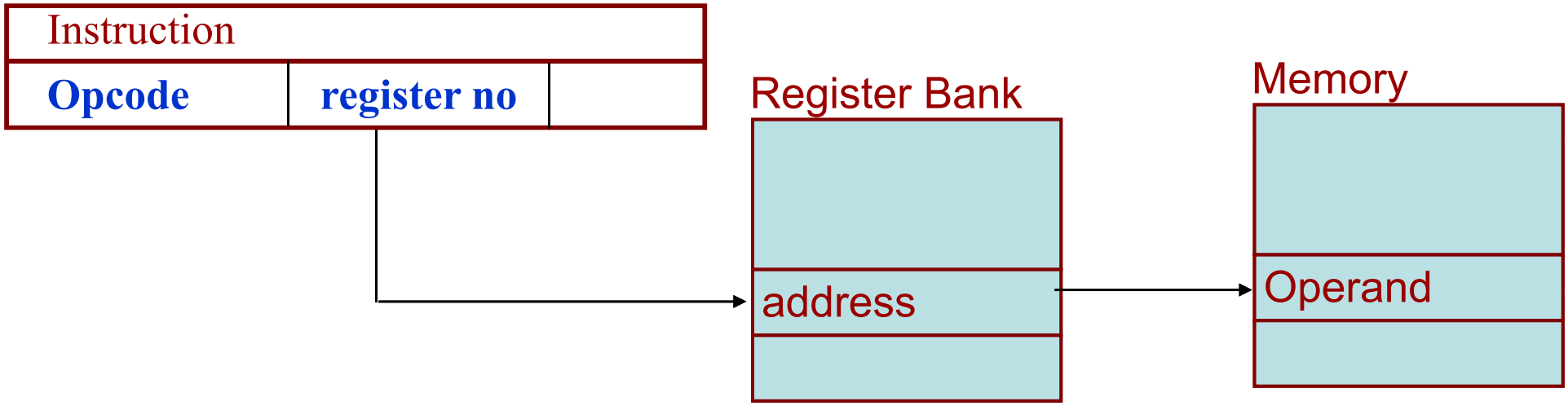
Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location.

It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.

Examples: ADD R1,(R5) // PC=R1+(mem[R5])



ddressing Modes

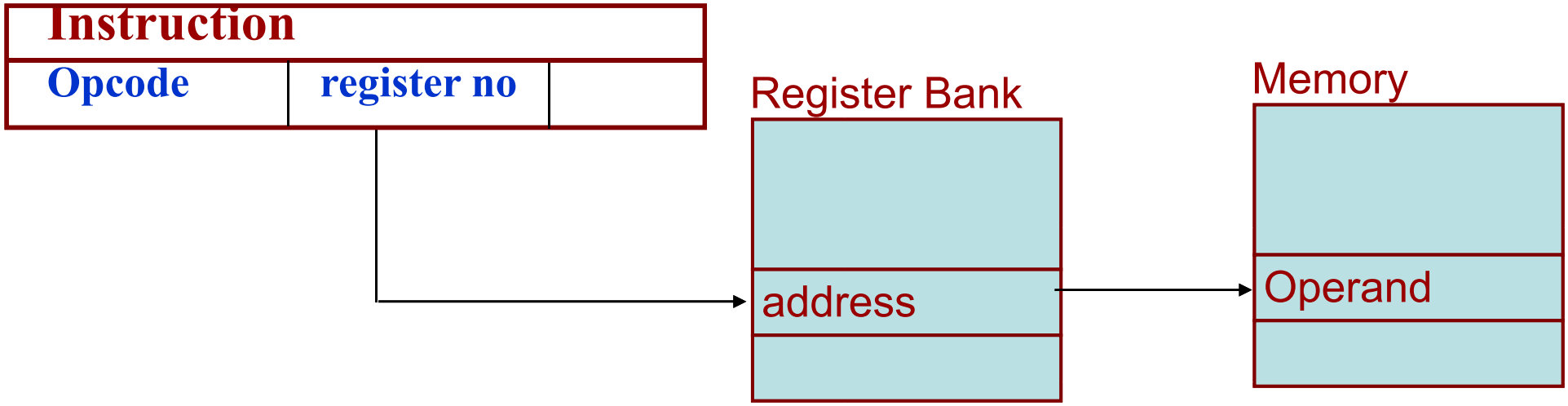
egister Indirect Addressing

Examples: ADD R1,(R5) // PC=R1+(mem[R5])

Here, memory address is kept in a register.

This register holds memory address that is used to access the operand.

Memory access is required as compared to indirect addressing mode.

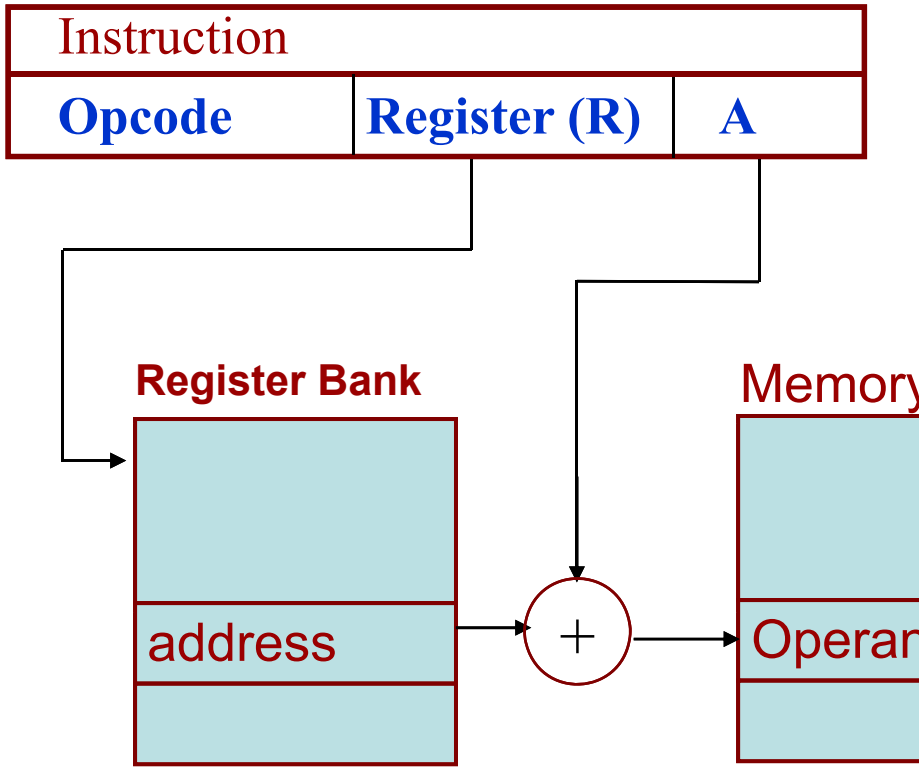


ddressing Modes

isplacement Addressing

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

- EA = A + (R)



Instruction has two address fields.

- one is explicit.
- Other address field (value = A) is directly specified.

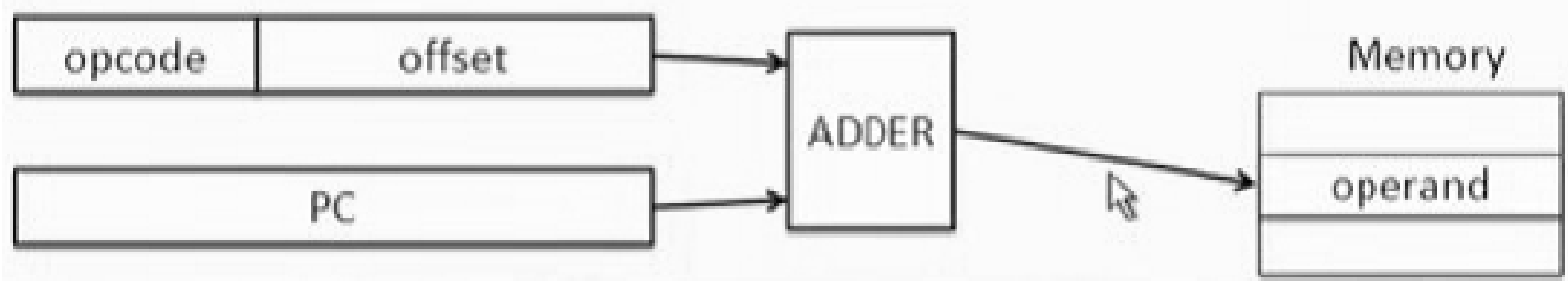
Other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

Three of the most common use of displacement addressing are:

- Relative addressing
- Base-register addressing
- Indexing

ddressing Modes

- **Relative Addressing (PC Relative)**
- **Instruction specifies an offset or displacement, which is added to the program counter to get the effective address of the operand.**
- **Since the number of bits to specify the offset is limited, the range of relative addressing is also limited. If a 12-bit offset is specified, it can have values ranging from -2048 to +2047.**
- **In branch instruction, we specify a branch address. Here, branch address will be added with the content of the PC. The offset is added with the content of PC and then we fetch the operand from EA.**

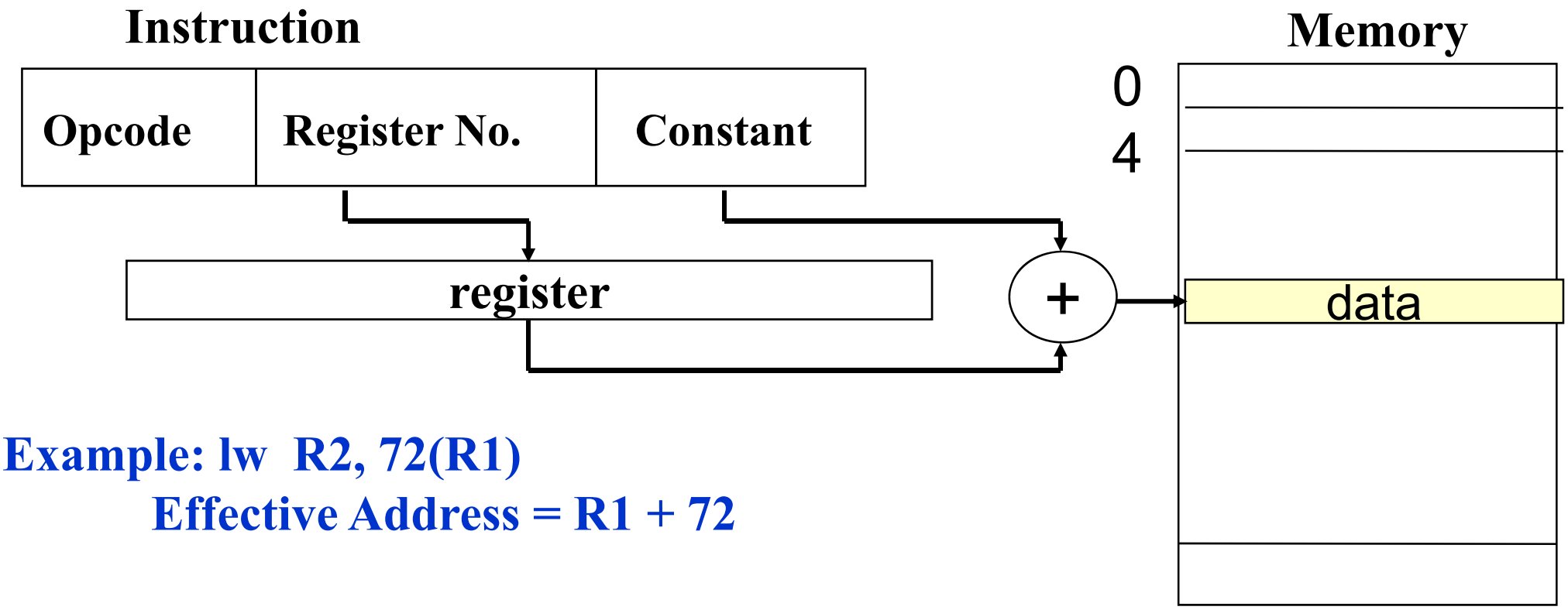


ddressing Modes

Base-Register Addressing:

Reference register contains a memory address, and the address field contains isplacement from that address. The register reference may be *explicit* or *implicit*.

n some implementation, a single segment/base register is employed and is used implicitl n others, the programmer may choose a register to hold the base address of a segmer nd the instruction must reference it explicitly.



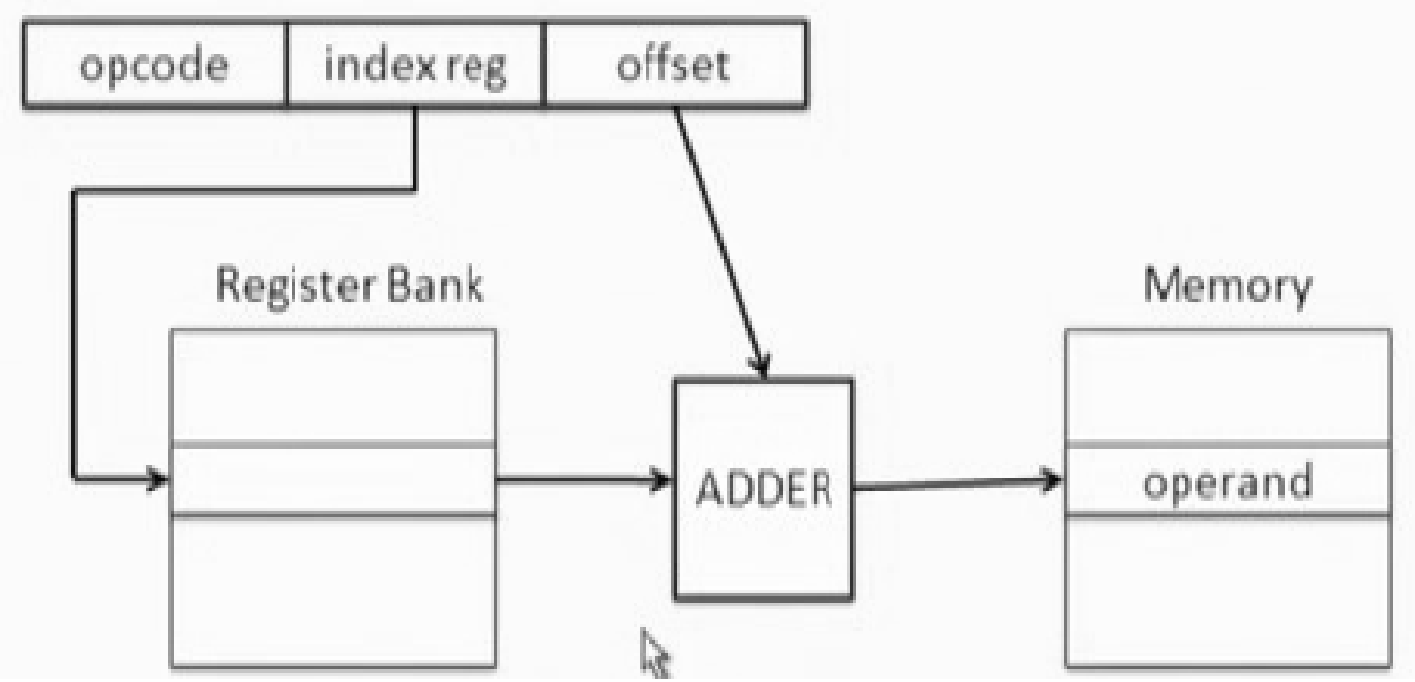
Example: lw R2, 72(R1)

Effective Address = R1 + 72

ddressing Modes

Indexed Addressing mode

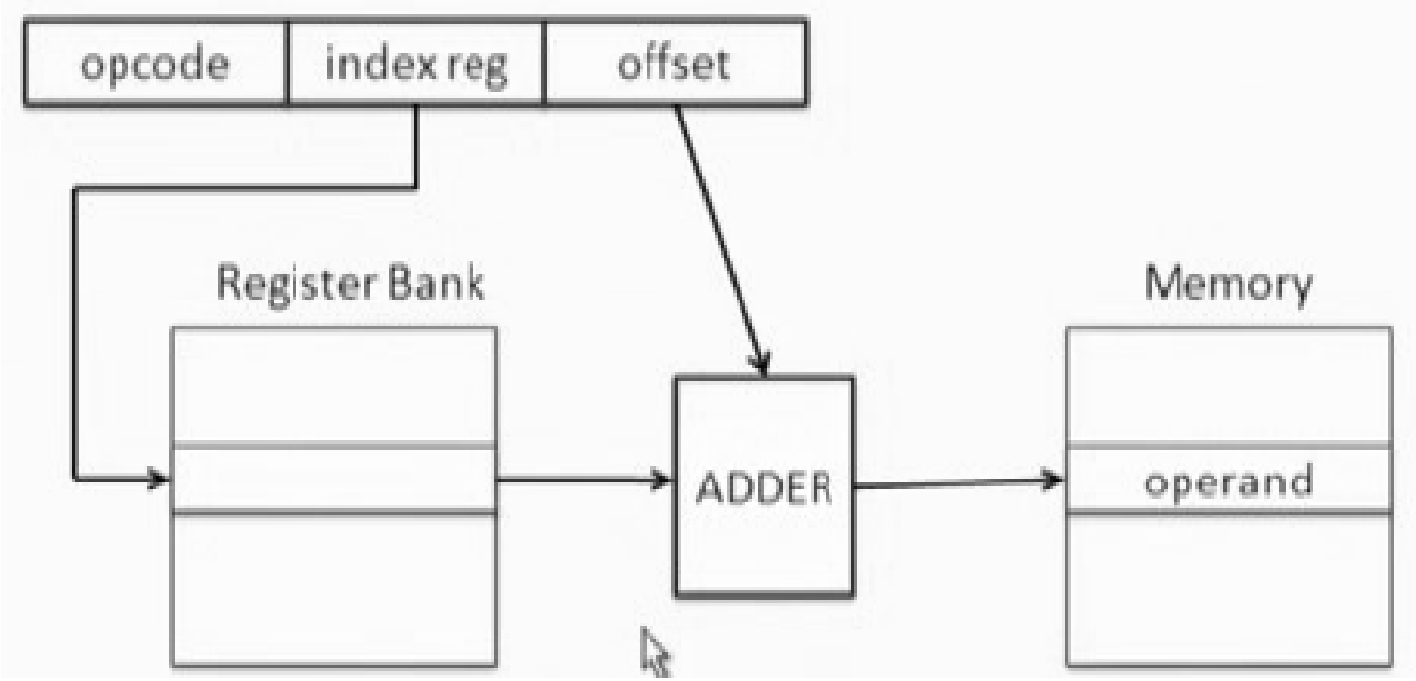
- Either a special-purpose register or a general-purpose register is used as index register. In this addressing mode, we can access an array. Array is a set of consecutive memory location.
- General-purpose register can be the used as index register and this instruction specifies an offset or displacement that is added to the index register to get the effective address of the operand.



ddressing Modes

Indexed Addressing mode

Example: LOAD R1, 2050(R2) // R1=Mem (2050+R2)



- Location 2050(R2) will give the operand. 2050 is added with R2. We get a value that value from which address in memory we get the operand, and it can be used to sequentially access the elements of an array. So, we load the first address and then we move to the next address by adding an offset to it.
- Offset gives the starting address of the array and the index register value specifies the array element to be used; The first can be 0th element, then the next, then next and so on.

ddressing Modes

tack Addressing

A stack is a linear array or list of locations.

It is also referred to as a *pushdown list* or *last-in-first out queue*.

A stack is a reserved block of locations. Items are appended to the top of the stack. At any given time, the block is partially filled.

Value of a pointer register known as stack pointer is the address of the top of the stack.

Stack pointer is associated with the stack. 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.

ddressing Modes

tack Addressing

xample

ADD, PUSH X, POP X

Operand is implicitly on the top of the stack.

It is used in zero-address machines.

First two elements on top of the stack will be taken out, will be added, and stored back there.

PUSH X will push the X value into top of the stack.

POP X will take out the top value to this location and store in X.

Many processors have a special register called a stack pointer that keeps track of the top of the stack.

Instruction Set Architecture

Instruction Set Architecture

broadly classified into i) Complex Instruction Set Computer (CISC), and Reduced Instruction Set Computer (RISC).

Main features of CISC are …

CISC has complex instruction set.

It also has a large number of addressing modes.

It also has some special-purpose registers and flags that are used to carry out various operations.

Instruction Set Architecture

CISC has variable length instructions. If computer has a fixed-length instruction, it becomes easy for encoding and decoding, but if it has variable-length instruction, then the encoding will be more complex and it also takes more time for decoding, because we have to know each of the bits and based on that, a particular action will be performed.

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. Of course, if we have variable length instruction, pipeline implementation will also be quite complex.

IBM 360, 370, VAX-11 were popular in seventies and eighties. From 1985 and till present, Intel architectures use CISC architecture.

It follows a CISC architecture and has survived over generations.

Computer Architecture

Reduced instruction set computer is used in most of the computers today; not only computers, also the processors for microcontrollers follow RISC architecture.

also referred as load-store architecture.

Only load and store instruction can be used to access the memory and all other instruction will be operated on register.

Only when memory data is required, we have to load it from the memory; once the data are available then you can access it.

Main feature of architecture is very simple for the sake of efficient pipeline.

Simple instruction set is used and very few addressing modes are used. It does not support variety of addressing modes; but it has a large number of general purpose registers. It does not have many special function registers.

Instruction length and encoding is uniform for easy decoding. If we encode the instruction all the instruction in a specific fashion, then the decoding will be much easier.

Compiler assisted scheduling of pipeline improves performance.

Since 60s, RISC architecture is gaining popularity.

ARM microcontroller family/MIPS are popular RISC processor. So, almost all computers today use RISC based pipeline processor for efficient implementation. RISC based computers use compilers to translate RISC instructions easily.

CISC based computer use the hardware to translate those instruction into simpler microinstructions.

omputer Architecture

C	CISC
t stands for Reduced Instruction Set Computer.	1. It stands for Complex Instruction Computer.
Relatively a few number of Instruction-ypically less than 100 instructions.	2. A large number of Instruction- typically f 100 to 250 instructions.
Relatively a few addressing modes-typically ess than or equal to 5.	3. A large number of addressing modes u typically from 5 to 15 different modes.
Easily decoded fixed length instructions.	4. Variable length instruction
Large number of general purpose registers – ypically more that 256 registers.	5. Small number of general purpose regis generally 8-24.
Three register operands are allowed per nstructions (e.g. ADD R1,R2,R3)	6. One or two operands are allowed instructions (e.g. ADD R1,MOV R1,R2)
Clock per Instruction (CPI) lies between 1-2 load and store -2)	7. Clock per Instruction (CPI) lies betwee 15 different modes.
Mostly hardwired control units.	8. Mostly micro-programmed control units
Operations are executed within registers of CPU only load and store instruction can access memory.	9. Most Instructions manipulate operanc memory.
Parameter passing through efficient on chip memory.	10. Parameter passing through inefficient chip memory.
Highly pipelined.	11. Less pipelined

Instruction Set Architecture

- MIPS architecture:
  - Developed by John Hennessy and his colleagues at Stanford University in the 1980's.
  - MIPS are popular RISC processor.
  - Used in many commercial systems, including NEC, Silicon Graphics, Nintendo, and Cisco.
- Real architecture but easy to understand

MIPS Arithmetic

- All Instructions have 3 operands
- Operand order is fixed
  - Destination appears first

Example

C code A=B+C

MIPS Code: add $s0, $s1,$s2

Instructions

- MIPS includes only simple, commonly used instructions. Same number of operands (two sources and one destination)
- Hardware to decode and execute the instruction is simple, small, and fast.
- More complex instructions (that are less common) can be performed using multiple simple instructions.
- MIPS is a *reduced instruction set computer* (RISC), with a small number of simple instructions.

Arithmetic Instructions: Addition

High-level code

```
a = b + c;
```

MIPS assembly code

```
add a, b, c
```

- add: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

Instructions: Subtraction

- Subtraction is similar to addition. Only the mnemonic changes.

High-level code	MIPS assembly code
a = b - c;	sub a, b, c

- sub: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

Instructions: More Complex Code

- More complex code is handled by multiple MIPS instructions.

High-level code

```
a = b + c - d;
```

MIPS assembly code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Operands

- **A computer needs a physical location from which it retrieves binary operands**

- **A computer retrieves operands from:**
  - **Registers**
  - **Memory**
  - **Constants (also called *immediates*)**

- **Scalars mapped to registers**

- **Structures, arrays etc in memory**

Operands: Registers

- Memory is slow.
- Most architectures have a small set of (fast) registers.
- MIPS has thirty-two 32-bit registers.
- MIPS is called a 32-bit architecture because it operates on 32-bit data.

  (A 64-bit version of MIPS also exists, but we will consider only the 32-bit version.)

MIPS Register Set

Name	Register Number	Usage
$0	0	Constant value 0
$at	1	Assembler temporary
$v0-$v1	2-3	Procedure return values
$a0-$a3	4-7	Procedure arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved variables
$t8-$t9	24-25	More temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Procedure return address

Operands: Registers

- **Registers:**
  - **Written with a dollar sign ($) before their name**
  - **For example, register 0 is written “$0”, pronounced “register zero” or “dollar zero”.**

- **Certain registers used for specific purposes:**
  - **For example,**
    - **$0 always holds the constant value 0.**
    - ***saved registers*, $s0-$s7, are used to hold variables**
    - ***temporary registers*, $t0 - $t9, are used to hold intermediate values during a larger computation.**

Instructions with registers

High-level code	MIPS assembly code
a = b + c	# $s0 = a, $s1 = b, $s2 = c add $s0, $s1, $s2

Expressions need to be broken

C code	MIPS code
A = B + C + D;	add $t0, $s1, $s2
E = F - A;	add $s0, $t0, $s3 sub $s4, $s5, $s0

Instructions with registers

Translate the following high-level code into assembly language. Assume variables a–c are held in registers $s0-$s2 and f–j are in $s3–$s7.

a =b-c;

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

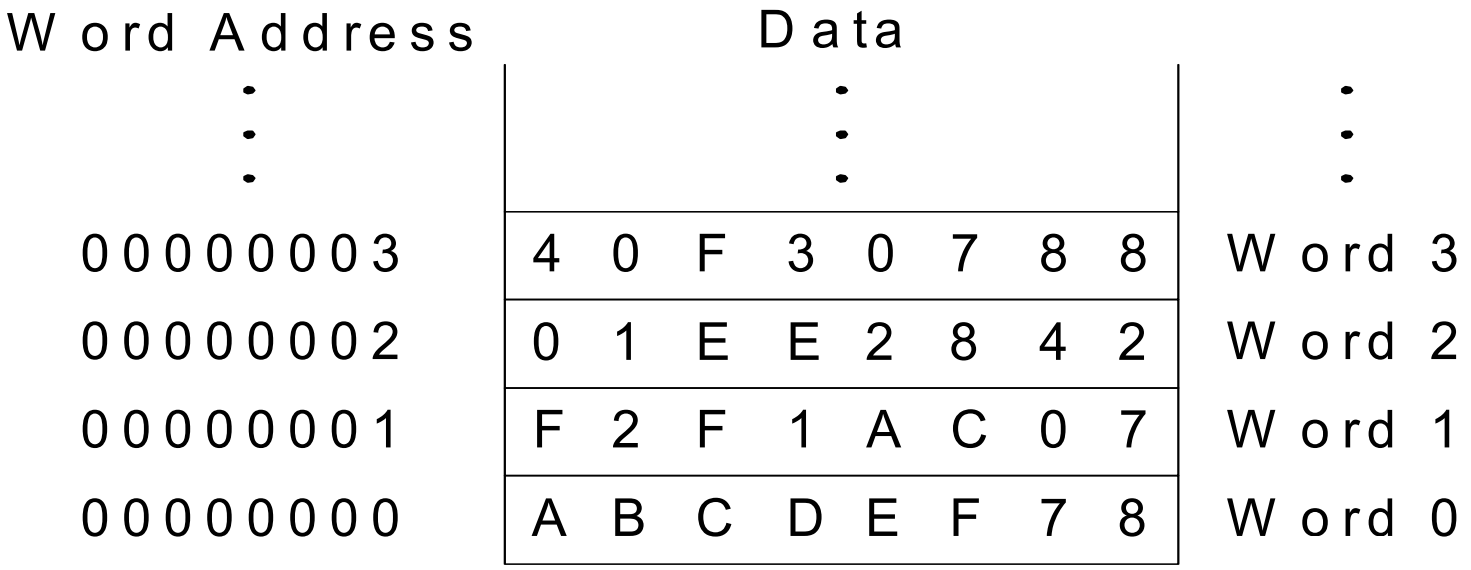
```
# MIPS assembly code
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h,
# $s6 = i, $s7 = j
  sub $s0, $s1, $s2   # a = b - c
  add $t0, $s4, $s5   # $t0 = g + h
  add $t1, $s6, $s7   # $t1 = i + j
  sub $s3, $t0, $t1   # f = (g + h) - (i + j)
```

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, so it can hold a large number of data, but it's also slow
- Commonly used variables kept in registers
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly

Word-Addressable Memory

- Each 32-bit data word has a unique address

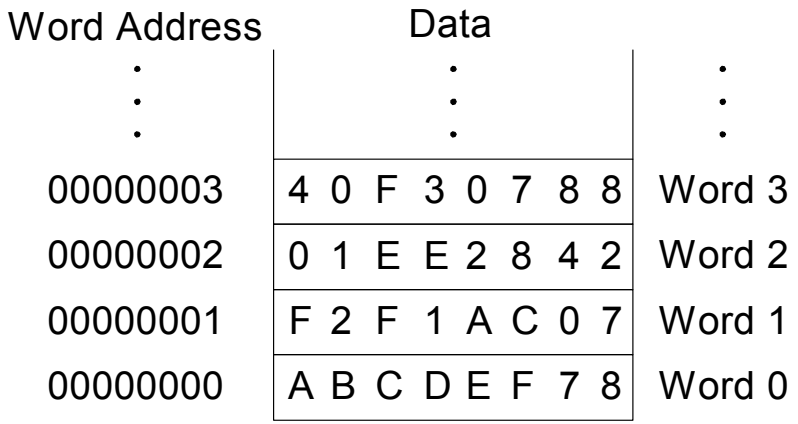


Reading Word-Addressable Memory

- Memory reads are called *loads*
- Mnemonic: *load word* (lw)
- Example: read a word of data at memory address 1 into $s3
- Memory address calculation:
  - add the *base address* ($0) to the *offset* (1)
  - address = ($0 + 1) = 1
- Any register may be used to store the base address.
- $s3 holds the value 0xF2F1AC07 after the instruction completes
- Hexadecimal constants are written with the prefix 0x

Assembly code

```
w $s3, 1($0)  # read memory
              word 1 into $s3
```



Writing Word-Addressable Memory

Memory writes are called *stores*

Mnemonic: *store word* (sw)

Example: Write (store) the value held in $t4 into memory address 3

Offset can be written in decimal (default) or hexadecimal

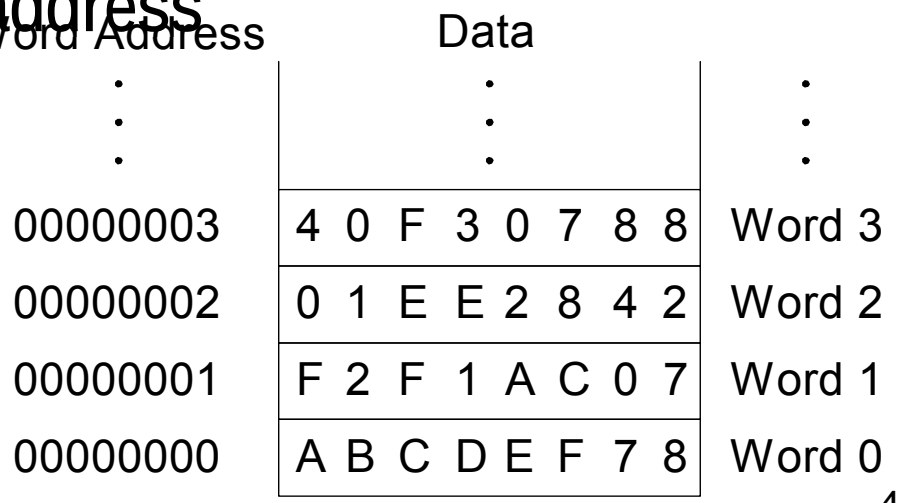
Memory address calculation:

- add the base address ($0) to the offset (0x3)
- address: ($0 + 0x3) = 3

Any register may be used to store the base address

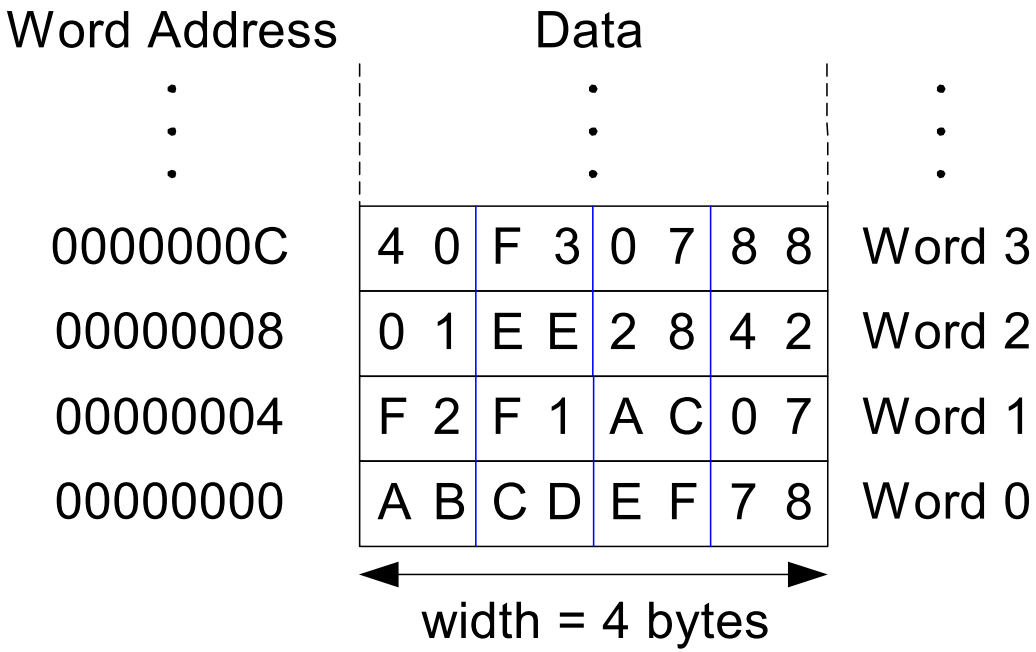
Assembly code

```
sw $t4, 0x3($0)  # write the value
                 of $t4 to memory word 3
```



Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- Each 32-bit words has 4 bytes, so the word address increments by 4



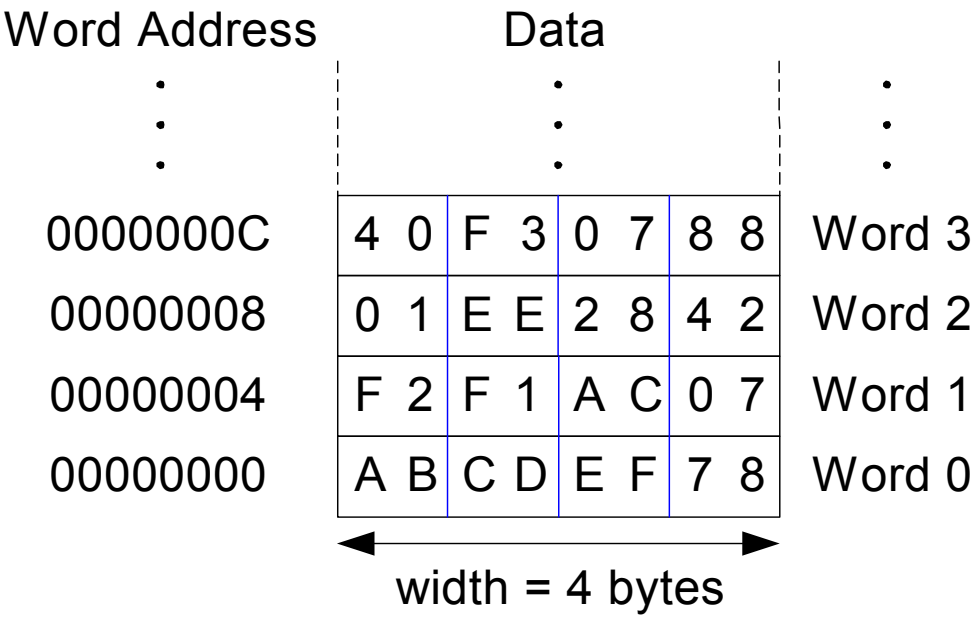
Reading Byte-Addressable Memory

- Address of a memory word must now be multiplied by 4.

  For example,
  - address of memory word 2 is 2 × 4 = 8
  - address of memory word 10 is 10 × 4 = 40 (0x28)
- Load a word of data at memory address 4 into $s3.
- $s3 holds the value 0xF2F1AC07 after the instruction completes.
- MIPS is byte-addressed, not word-addressed

MIPS assembly code

w $s3, 4($0) # read word at
address 4 into $s3

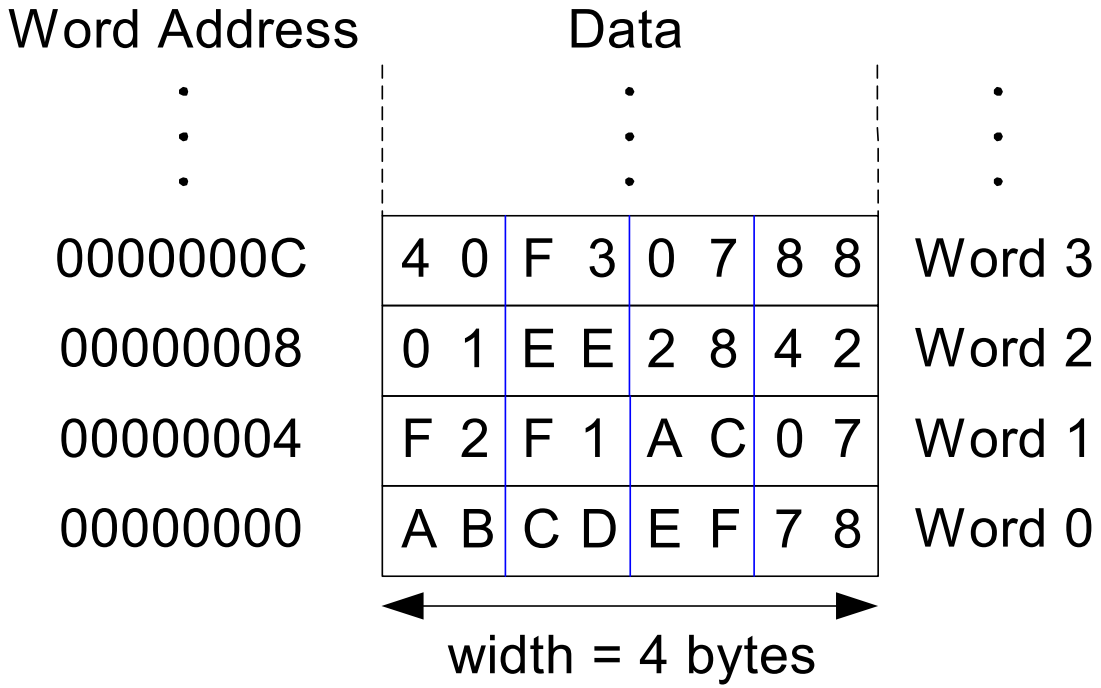


Writing Byte-Addressable Memory

Example: stores the value held in $t7 into memory address 0x2C (44)

MIPS assembly code

```
sw $t7, 44($0)  # write $t7 into address 44
```



Load / Store example

C code: A[8] = h + A[8];

MIPS code:

```
lw   $t0, 32($s3)
add $t0, $s2, $t0
sw   $t0, 32($s3)
```

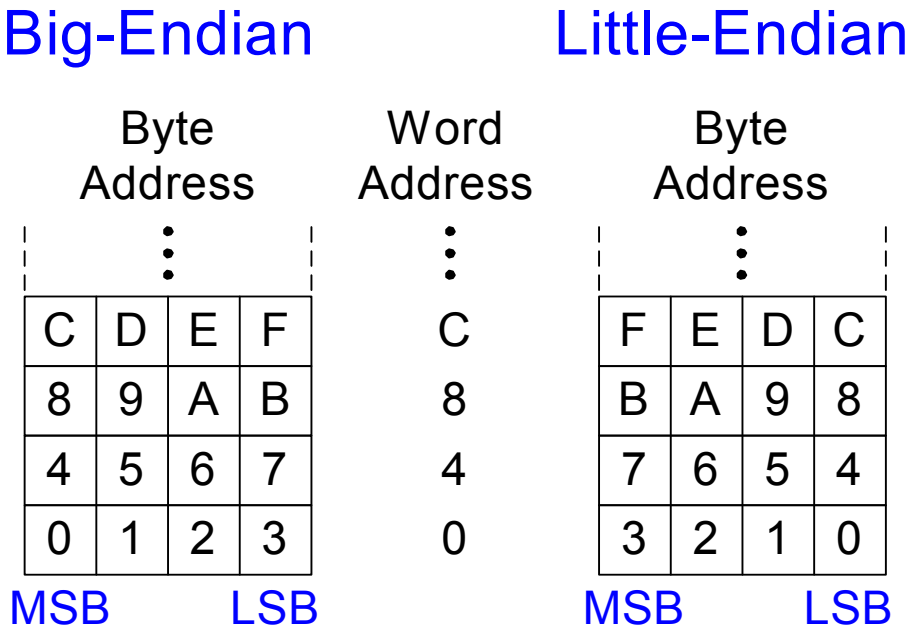
Words and Bytes

- 2^{32} bytes : byte addresses from 0 to 2^{32}-1

- 2^{30} words : byte addresses 0, 4, 8, ... 2^{32}-4

Big-Endian and Little-Endian Memory

- Word address is the same for big- or little-endian
- Little-endian: byte numbers start at the little (least significant) end
- Big-endian: byte numbers start at the big (most significant) end



Big Endian Addressing

- With Big Endian addressing, the byte binary address

  x . . . x00

  is in the most significant position (big end) of a 32 bit word (IBM, Motorolla, Sun, HP).

MSB			LSB
0	1	2	3
4	5	6	7

Little Endian Addressing

- With Little Endian addressing, the byte binary address

  x . . . x00

  is in the least significant position (little end) of a 32 bit word (DEC, Intel).

M S B			L S B
3	2	1	0
7	6	5	4

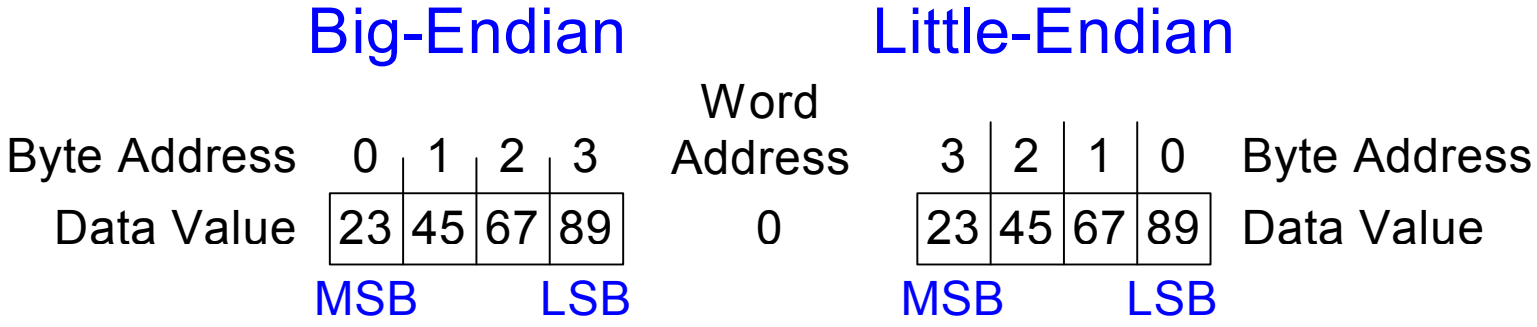
- Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

Big- and Little-Endian Example

- Suppose, $t0 initially contains 0x23456789. After the following program is ru on a big-endian system, what value does $s0 contain? In a little-endia system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- **Big-endian: 0x00000045**
- **Little-endian: 0x00000067**



Operands: Constants/Immediates

- lw and sw illustrate the use of constants or *immediates*
- Called immediates because they are *immediate*ly available from the instruction
- Immediates don't require a register or memory access.
- Add immediate (addi) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.

High-level code

```
a = a + 4;
b = a – 12;
```

MIPS assembly code

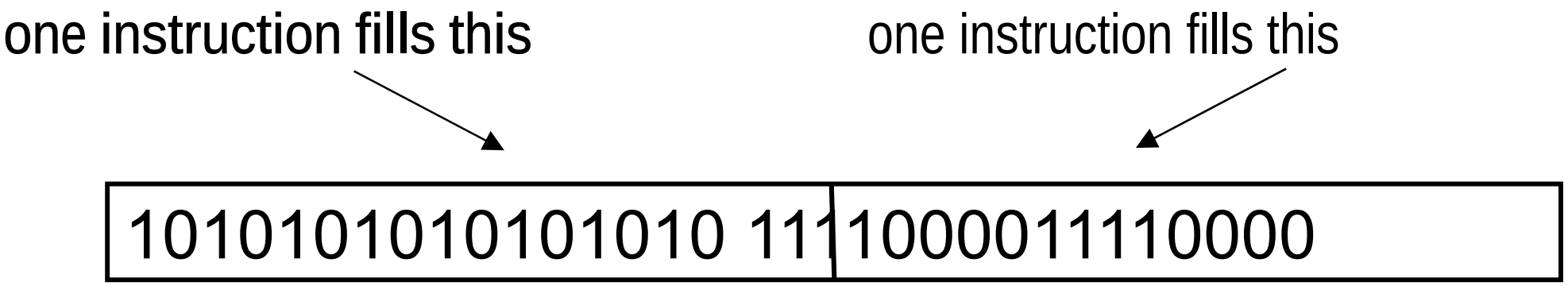
```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

Constants in MIPS instructions

- addi $29, $29, 4 ‘i’ is for ‘immediate’
- slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4

How about larger constants?

To load a 32 bit constant into a register, we use two instructions



Loading larger constants

- "load upper immediate" instruction
  lui $t0, 1010101010101010
- then get the lower order bits right, i.e.,
  ori $t0, $t0, 1111000011110000