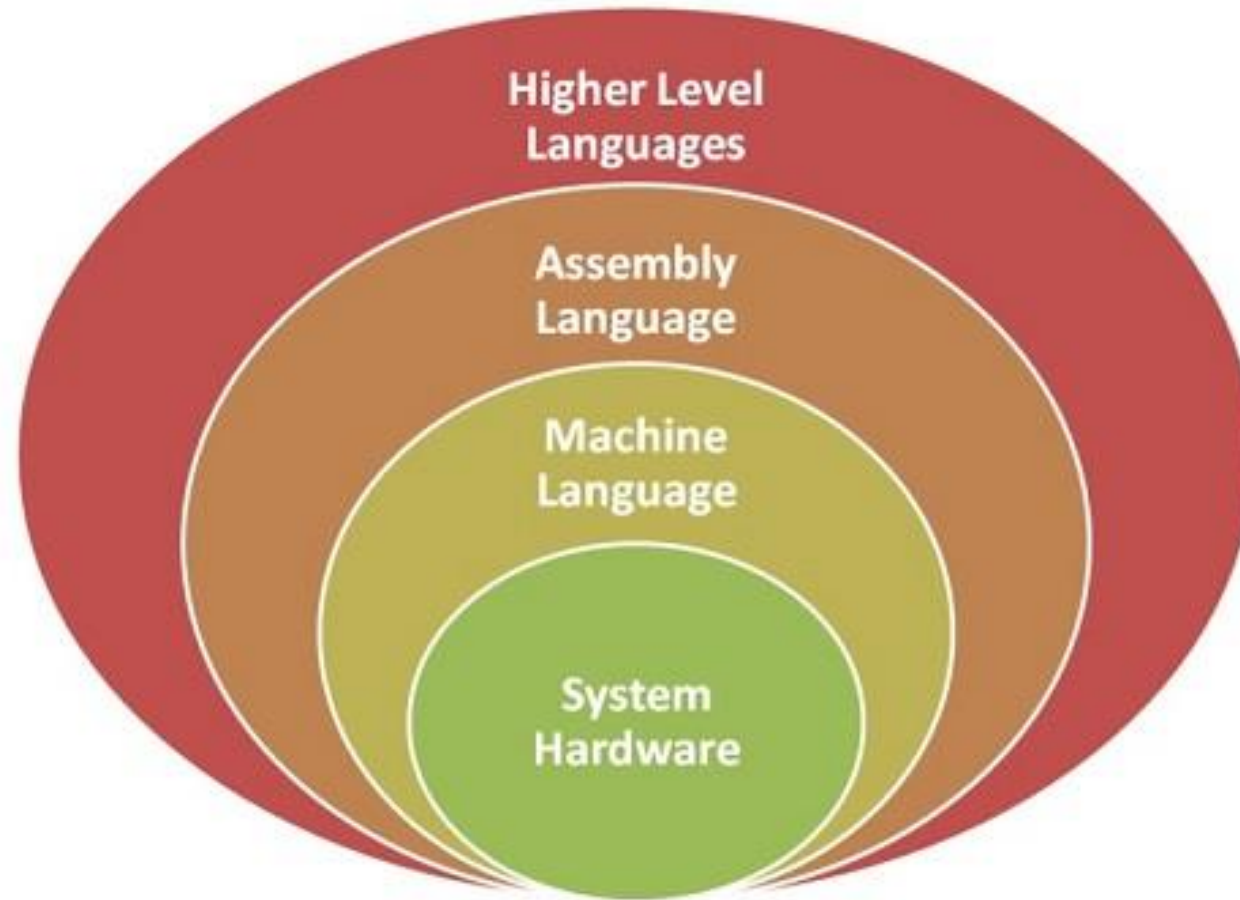


Computer Architecture

Instructions

Hierarchy of Computer Languages



Instruction Codes

▶ Program

- A program is a set of instructions that specify the operations, operands and the sequence by which processing has to occur.

▶ Computer Instruction

- A computer instruction is a binary code that specifies a sequence of micro-operations for the computer.
- The computer reads each instruction from memory and places it in a control register.
- The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of micro-operations.

▶ Instruction Code

- An instruction code is a group of bits that instruct the computer to perform a specific operation.
- Example

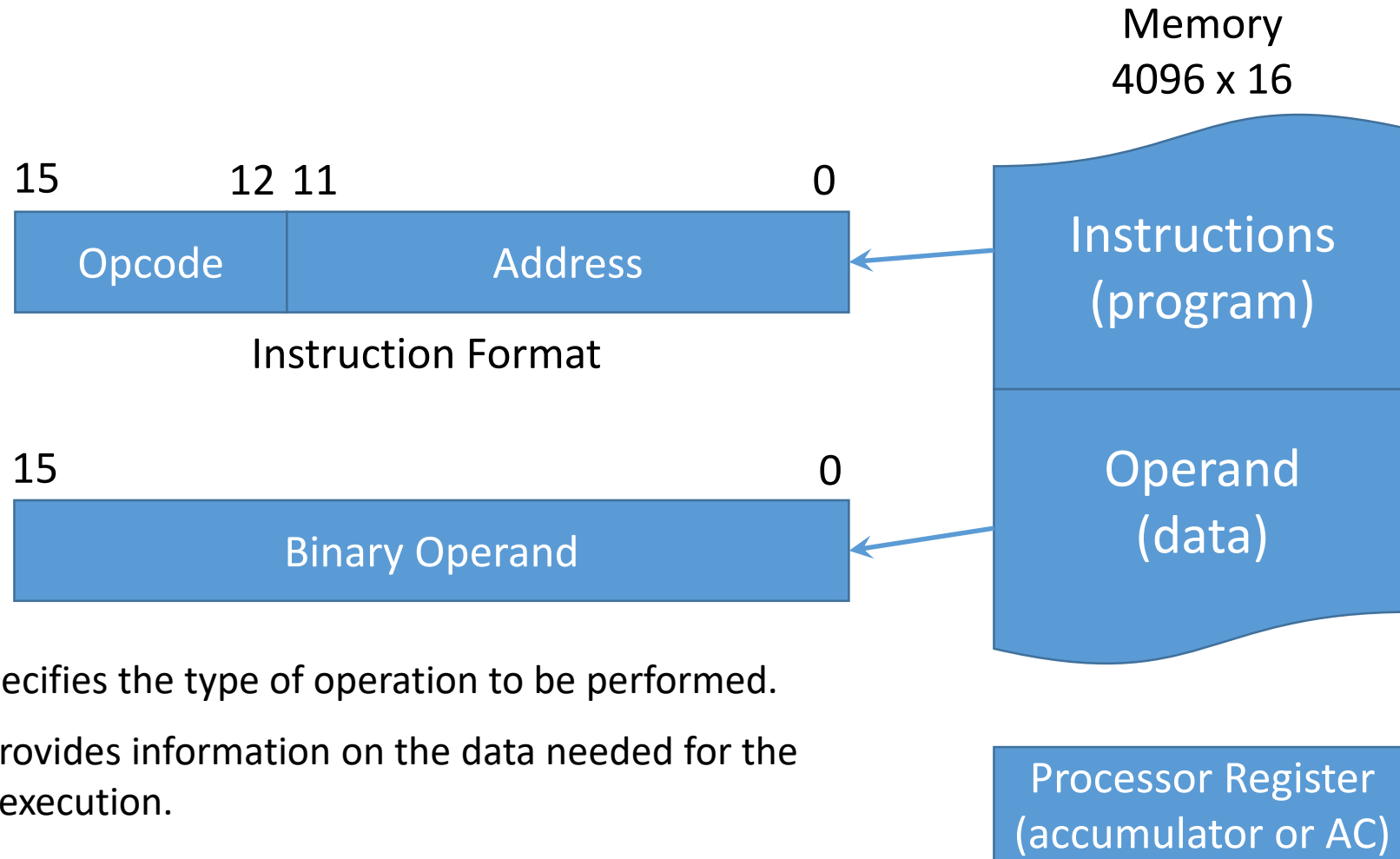
ADD
1547

Unique Binary
code is assigned
to every Opcode

▶ Operation Code (Opcode)

- The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer.
- The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.

Stored Program Organization



Opcode: Specifies the type of operation to be performed.

Operand: Provides information on the data needed for the instruction execution.

Stored Program Organization

- ▶ The simplest way to organize a computer is to have one processor register(AC) and an instruction code format with two parts.
 - ➔ The first part specifies the operation (**opcode**) to be performed and the second specifies an address (**operand**).
- ▶ The memory address tells the control where to find an operand in memory.
- ▶ This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.
- ▶ Instructions are stored in one section of memory and data in another.
- ▶ For a memory unit with 4096 words, we need 12 bits to specify an address since $2^{12} = 4096$.
- ▶ If we store each instruction code in one 16-bit memory word, we have available four bits for operation code (opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- ▶ The control reads a 16-bit instruction from the program portion of memory.
- ▶ It then executes the operation specified by the operation code.

- **Instruction:** The words of a computer language is called instruction
- **Instruction set :** The vocabulary of computer language is called instruction set.

Goal of the computer designer:

To find a language that makes it easy to build the hardware and compiler while **maximizing performance** and **minimizing cost**.

Instruction Set Classification

Data Movement	<ul style="list-style-type: none">• Data read and write from/to memory (Load/Store)
Arithmetic and Logical	<ul style="list-style-type: none">• Add, Subtract, Multiply, Divide, Shift, Logical
Transfer or Control	<ul style="list-style-type: none">• Branch, conditional Branch(for loop or control flow of program)• System Control (Halt, Swap,Interrupt, Call, Return, etc)
Input/Output	<ul style="list-style-type: none">• IN for reading from Devices/Disk• OUT for writing/displaying on devices/Disk
Miscellaneous	<ul style="list-style-type: none">• String compare, Set Flags, Clear Flags
Floating point	<ul style="list-style-type: none">• FP arithmetic
Binary Coded Decimal	<ul style="list-style-type: none">• Decimal Arithmetic
Vector	<ul style="list-style-type: none">• For graphic operations
Emulated Instructions	<ul style="list-style-type: none">• User defined opcode and operation

Data Movement Instructions: These support movement of data between registers, registers to memory, memory to register.

Move	R1,	Total
Move	Total,	R1
Load	R1,	Total
Store	R1,	Total

Arithmetic and Logical Instructions: This category of instructions carry out calculations. The minimum in this category is ADD, SUB, AND, OR, XOR, SHIFT. Multiply and Divide can always be emulated using successive addition or subtraction.

ADD	R1,	TOTAL
ADD	R1,	R2, R3
XOR	R4,	TOTAL
MUL	R4,	MARKS

Transfer or Control Instructions: This category of instructions facilitate change in program flow described by the control structures in the high-level language. BRANCH or JMP instructions along with Condition Code flags achieve the requirement. Subroutine CALLS, RETURNS are categorized here.

JMP	LABLE1	Jump
JNZ	LABLE1	Jump on Not Zero
BZ	LABLE2	Branch on Zero
BNE	LABLE3	Branch on Not Equal

Input/Output Instructions:

There are two ways of doing Input/Output operations.

```
IN      Port#232
OUT     Port#234
Move    R1, #FFEEEE
```

Miscellaneous Instructions: NOP (No Operation) is a famous dummy instruction but a very useful one in this category. The NOP instruction does nothing, effectively consuming one clock cycle without performing any action.

```
HALT
NOP
INT
```

Floating Point Instructions:

We had seen in the previous chapter that specialised hardware is required for efficient Floating-point operations.

```
FLD      FP Load
FST      FP Store
FADD     FP ADD
FSUB     FP Subtract
FMUL     FP Multiply
```

Binary Coded Decimal Instructions: Decimal Number system hardware speeds up decimal calculations.

CPU architecture design radically differs based on the size of the Instruction set. Accordingly, there are two categories namely **Reduced Instruction Set Computer (RISC)** and **Complex Instruction Set Computer (CISC)**. As the name implies, RISC has

Instruction Formats and Classification

The components to be considered while deciding and designing instruction format is:

1. The number of instructions in the instruction set
2. The number of CPU registers to be addressed
3. Size of Main memory to be addressed and organization
4. Addressing modes to be mapped

Encoding of Instructions is called Instruction Format. There are two generic ways to encode instructions.

Fixed length Instruction Format - RISC uses fixed-length encoding.

Variable Length Instruction Format - CISC uses variable-length encoding.

Variable Length Instruction format(CISC)

- Each instruction occupies only the space required by it to convey the operands. No redundant use of memory space.
- Essentially this format provides more options to address more operands.
- Calculating the next instruction location is complex and depends on the current instruction type.
- The fetch and decode logic and Control Unit design is complex.
- Code efficiency is possible, but frequent memory reference implies relatively reduced efficiency.
- A Large number of instructions have different opcodes but same function, implying varying addressing modes.

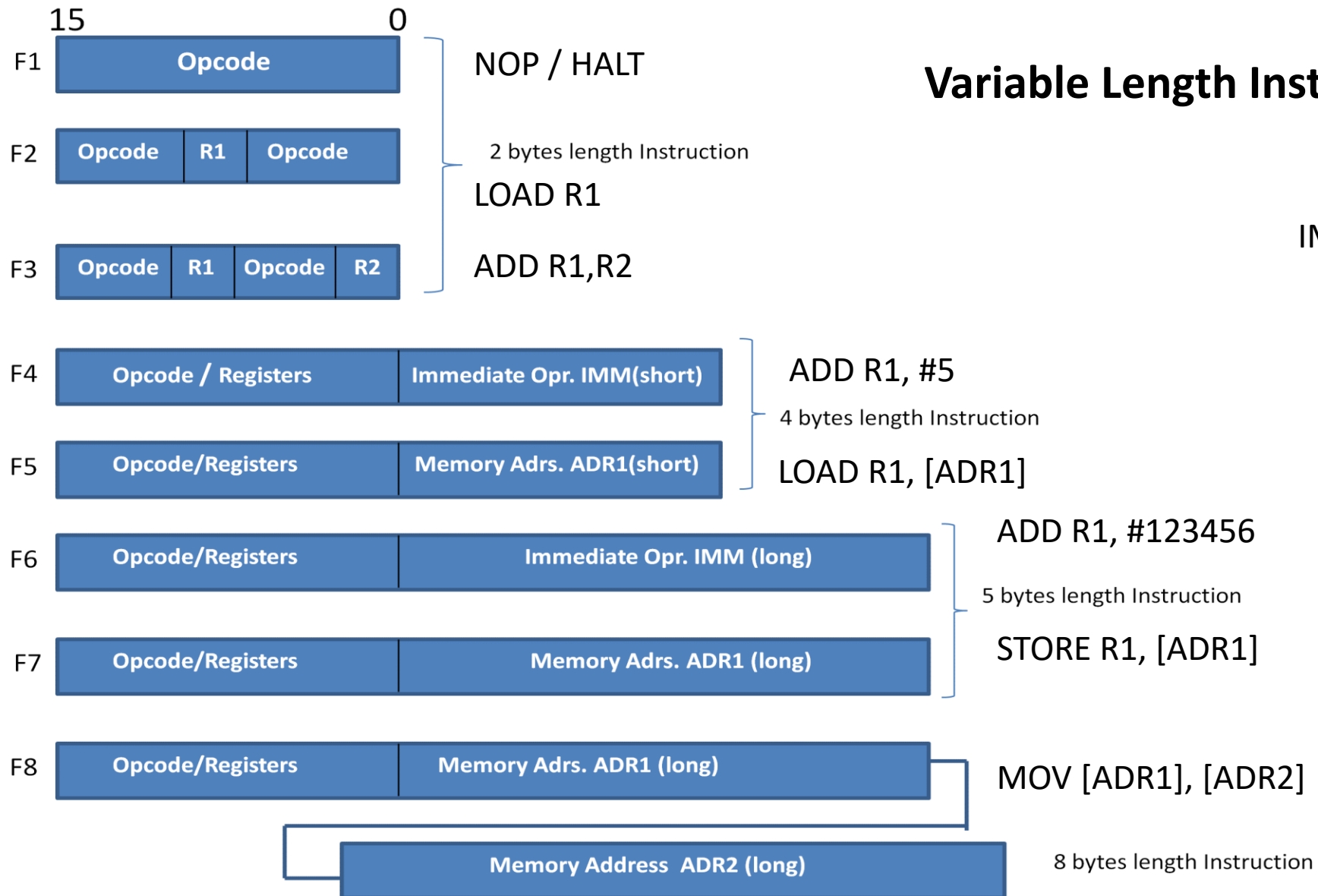
Fixed Length Instruction format(RISC)

- Each Instruction occupies the same amount of space. The length of the instruction is fixed irrespective of the opcode.
- The instruction fetch process is simpler. The next instruction address is obtained by incrementing the Program Counter by a fixed count.
- The instruction decoding process is also simpler and hence the Control Unit design is less complex comparing variable-length format.
- Faster and better performance, as the design, makes use of CPU Registers more effectively.
- The program code (machine code) for these machines is longer and proportional to the number of instructions.
- Few simple instructions equivalent to HALT, NOP which do not require any operand also occupies the same length and size of memory. A little waste of memory space at the cost of CPU efficiency.

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and microprogrammed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word.	Instructions are larger than the size of one word
Simple and limited addressing modes.	Complex and more addressing modes.
RISC is Reduced Instruction Cycle.	CISC is Complex Instruction Cycle.
The number of instructions are less as compared to CISC.	The number of instructions are more as compared to RISC.
It consumes the low power.	It consumes more/high power.
RISC is highly pipelined.	CISC is less pipelined.
RISC required more RAM .	CISC required less RAM.
Here, Addressing modes are less. Smartphones, IoT devices, basic embedded tasks	Here, Addressing modes are more. Laptops, PCs, advanced embedded systems

Variable Length Instruction format

IMM = Immediate Operand



Fixed Length Instruction format



J-type J LABEL: Jump to a specific memory address (LABEL).



I-type ADDI R2, R1, 5



R-type ADD R3, R1, R2

Rs (Source Register): Specifies the first source register.

Rt (Target Register): Specifies the destination register.

Rd (Destination Register): Specifies where the result is stored.

Shift Amount: Indicates how much a value should be shifted (used in shift operations).

Function: Specifies the exact operation (e.g., ADD, SUB).

- Operations of the computer hardware:

Every computer must be able to perform arithmetic. The MIPS assembly language notation

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

instructs a computer to add the two variables `b` and `c` and to put their sum in `a`.

--per line one instruction

--must always have exactly three variables

MIPS (an acronym for Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies (formerly MIPS Computer Systems)

The following sequence of instructions adds the four variables:

```
add a, b, c      # The sum of b and c is placed in a.  
add a, a, d      # The sum of b, c, and d is now in a.  
add a, a, e      # The sum of b, c, d, and e is now in a.
```


- Operands of the computer hardware:

Name	Example	Comments
32 registers	$\$s0, \$s1, \dots, \$s7$ $\$t0, \$t1, \dots, \$t7$	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $\$s0-\$s7$ map to 16–23 and $\$t0-\$t7$ map to 8–15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Memory is byte addressed – Each address identifies an 8-bit byte

Words are aligned in memory – Address must be a multiple of 4

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

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

The variables *f*, *g*, *h*, *i*, and *j* are assigned to the registers *\$s0*, *\$s1*, *\$s2*, *\$s3*, and *\$s4*, respectively. What is the compiled MIPS code?

\$t0, *\$t1*, ..., *\$t9* for
temporary data values
\$s0, *\$s1*, ..., *\$s7* for saved
variables
Word=32bit =4byte

• Answer:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1  # f gets $t0 - $t1, which is (g + h)-(i + j)
```

- How computer hardware access large memory structure?

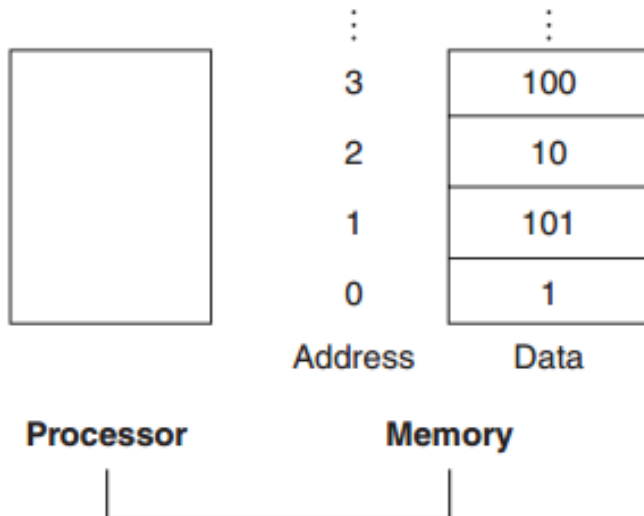
--- Data structures (array and structures) are used.

Data transfer instruction:

A command that moves data between memory and register

Memory to register = load

Register to memory = store

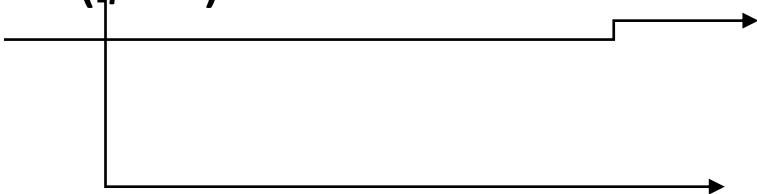


Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure 2.2, the address of the third data element is 2, and the value of `Memory[2]` is 10.

Let's assume that *A* is an array of 100 words and that the compiler has associated the variables *g* and *h* with the registers *\$s1* and *\$s2* as before. Let's also assume that the starting address, or *base address*, of the array is in *\$s3*. Compile this C assignment statement:

Answer:

```
lw $t0, 32($s3)
```



$g = h + A[8]$

offset

base register

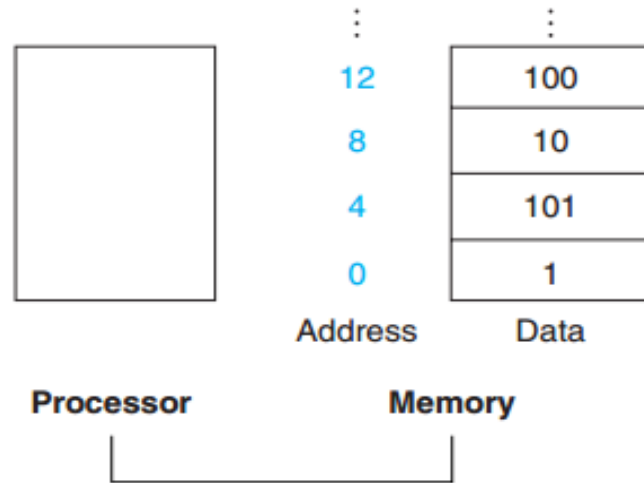
Lw = load word
Suppose, $g = \$s1$ $h = \$s2$
Base address of *A* is *\$s3*
Memory address = Base reg + 4*Offset = $\$s3 + 32 = 32(\$s3)$

```
add $s1, $s2, $t0
```

The constant in a data transfer instruction is called the *offset*, and the register added to form the address is called the *base register*.

- Alignment register:

Words must start at address that are multiples of 4.



word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

- Spilling register:

Process of putting less commonly used variable in a memory is called spilling register.

Assume variable `h` is associated with register `$s2` and the base address of the array `A` is in `$s3`. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select `A[8]`, and the add instruction places the sum in `$t0`:

```
lw    $t0,32($s3)    # Temporary reg $t0 gets A[8]

add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into `A[12]`, using 48 as the offset and register `$s3` as the base register.

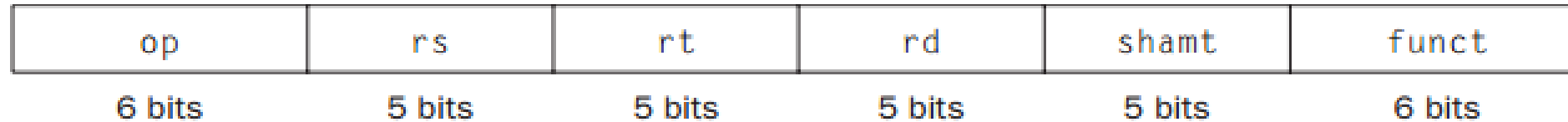
```
sw    $t0,48($s3)    # Stores h + A[8] back into A[12]
```

Representing Instructions in the computer:

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

- R-type:



Here is the meaning of each name of the fields in MIPS instructions:

- *op*: Basic operation of the instruction, traditionally called the **opcode**.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (Section 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
- *funct*: Function. This field selects the specific variant of the operation in the *op* field and is sometimes called the *function code*.

Example:

add \$t0, \$s1, \$s2

op = 0

rd= \$t0 = 8

rs= \$s1 = 17

rt= \$s2 = 18

Shmt = 0

Funct = 32

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	0100	000000	100000
--------	-------	-------	------	--------	--------

00000010001100100100000000100000₂ = 02324020₁₆

- I – format :



- Example:

lw \$t0, 32(\$s1)



Opcode (6 bits): The operation being performed is lw (load word), which has an opcode of 100011 in binary.

Rs (5 bits): (Source Reg) The base register used for the memory address is \$s1. \$s1 is register 17 in MIPS register encoding (binary: 10001).

Rt (5 bits): The destination register is \$t0, which will store the loaded word. \$t0 is register 8 in MIPS register encoding (binary: 01000).

Immediate (16 bits): The immediate value is 32, which is the offset added to the base address stored in \$s1. In binary, 32 is represented as 0000 0000 0010 0000 (16 bits).

binary representation of lw \$t0, 32(\$s1) is:

100011 10001 01000 0000 0000 0010 0000

- j- type:

- Opcode=2

- Address=given address/4;



We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
add   $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Decision making statement:

```
beq register1, register2, L1
```

This instruction means go to the statement labeled L1 if the value in `register1` equals the value in `register2`. The mnemonic `beq` stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled L1 if the value in `register1` does *not* equal the value in `register2`. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

if (i==j) go to L1;

f = g+h;

L1: f=f-i;

Convert this into MIPS assembly code.

Answer :

f= \$s0, g= \$s1, h= \$s2, i= \$s3, j= \$s4

beq \$s3, \$s4, L1

add \$s0, \$s1, \$s2

L1: sub \$s0, \$s0, \$s3

In the following code segment, *f*, *g*, *h*, *i*, and *j* are variables. If the five variables *f* through *j* correspond to the five registers \$s0 through \$s4, what is the compiled MIPS code for this C *if* statement?

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

Answer:

```
bne $s3, $s4, ELSE
```

```
add $s0, $s1, $s2
```

```
j EXIT
```

[j means jump to exit]

```
ELSE:
```

```
sub $s0, $s1, $s2
```

```
EXIT
```

N.B: use jump in case of if... else

- Loop: $g = g + A[i]$
 $i = i + j$
 If $(i \neq h)$ go to LOOP.

What will be the MIPS assemble code?

Answer:

$g = \$s1, h = \$s2, i = \$s3, j = \$s4, A = \$s5$

Loop:

$\text{add } \$t1, \$s3, \$s3 \text{ // } \$t1 = 2 * i$

$\text{add } \$t1, \$t1, \$t1 \text{ // } \$t1 = 4 * i$ In MIPS, the size of each word (array element) is 4 bytes. Thus, multiplying the index i by 4 gives the byte offset for $A[i]$.

$\text{add } \$t1, \$t1, \$s5 \text{ // } \$t1 = \text{address of } A[i] = (4 * i + \$s5)$

$\text{lw } \$t0, 0(\$t1) \text{ // used that address to load } A[i] \text{ into a temporary register – As we already done } 4 * i + \text{ as it } i \text{ is so we use } 0$

$\text{add } \$s1, \$s1, \$t0 \text{ // } g = g + A[i]$

$\text{add } \$s3, \$s3, \$s4 \text{ // } i = i + j$

$\text{bne } \$s3, \$s4, \text{LOOP}$

Rule for getting address of an array:

1. Multiply the index i by 4

2. Add with the base of array

while (save[i] = k)

i= i+j

what will be the MIPS assembly code for this?

Answer:

i=\$s3, j=\$s4, k=\$s5, save = \$s6

LOOP:

add \$t1, \$s3, \$s3 // \$t1 = 2*i

add \$t1, \$t1, \$t1 // \$t1 = 4*i

add \$t1, \$t1, \$s6 // \$t1 = address of save[i] = (4*i + \$s6)

lw \$t0, 0(\$t1)

bne \$t0, \$s5, EXIT

add \$s3, \$s3, \$s4

j LOOP

EXIT

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
and	R	0	18	19	17	0	36	and \$s1,\$s2,\$s3
or	R	0	18	19	17	0	37	or \$s1,\$s2,\$s3
nor	R	0	18	19	17	0	39	nor \$s1,\$s2,\$s3
andi	I	12	18	17	100			andi \$s1,\$s2,100
ori	I	13	18	17	100			ori \$s1,\$s2,100
sll	R	0	0	18	17	10	0	sll \$s1,\$s2,10
srl	R	0	0	18	17	10	2	srl \$s1,\$s2,10
beq	I	4	17	18	25			beq \$s1,\$s2,100
bne	I	5	17	18	25			bne \$s1,\$s2,100
slt	R	0	18	19	17	0	42	slt \$s1,\$s2,\$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

- What is the code to test if variable a (=\$s0) is less than variable b (=\$s1) and then branch to label LESS if the condition holds?

Slt = set if less than
Blt = branch if less than

Answer:

Solution1:

```
slt $t0, $s0, $s1
```

```
bne $t0, $zero, LESS
```

Solution 2:

```
blt $s0, $s1, LESS # Branch to LESS if $s0 < $s1
```

LESS:

```
# Code to execute if $s0 < $s1
```

Solution 1 explanation: Uses a temporary variable to check if the condition is true/false

If $\$s0 < \$s1$

$\$t0=1$

If $\$s0 > \$s1$

$\$t0=0$

32-Bit Immediate Operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (`lui`) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. Figure 2.23 shows the operation of `lui`.

Ori = Bitwise or immediate

The machine language version of `lui $t0, 255` # \$t0 is register 8:

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Contents of register \$t0 after executing `lui $t0, 255`:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

FIGURE 2.23 The effect of the `lui` instruction. The instruction `lui` transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register \$s0?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61    # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register \$s0 afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to add the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register \$s0 is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

MIPS Addressing Mode Summary

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

Multiple forms of addressing are generically called **addressing modes**. The MIPS addressing modes are the following:

1. **Register addressing**, where the operand is a register
2. **Base** or **displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
3. **Immediate addressing**, where the operand is a constant within the instruction itself
4. **PC-relative addressing**, where the address is the sum of the PC and a constant in the instruction
5. **Pseudodirect addressing**, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

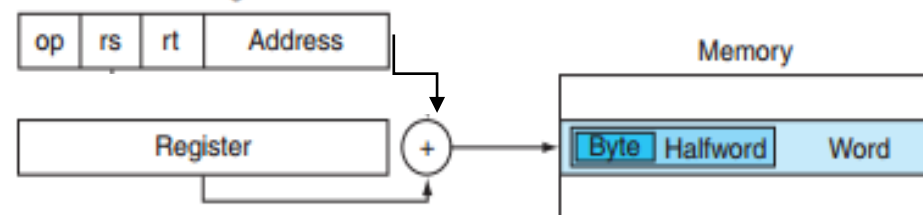
1. Immediate addressing



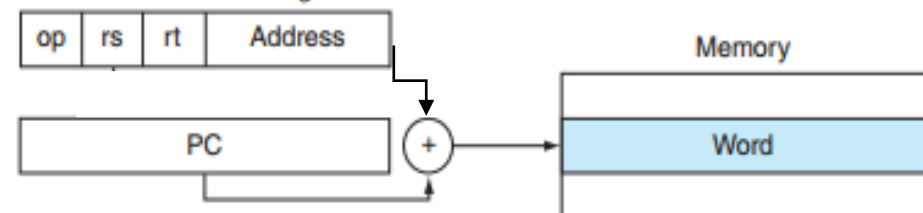
2. Register addressing



3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing

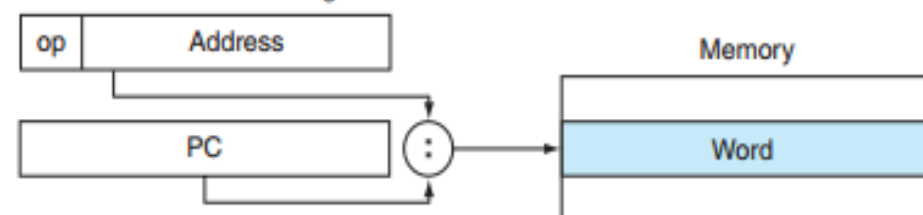


FIGURE 2.24 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

- **Register Addressing:**

- Register Addressing is a source or destination operand is specified as content of one of the registers.
- **Register Addressing** is considered the simplest addressing mode. This is because both operands are in a register.
- Which allow instructions to be executed much more faster in comparison with other addressing modes because they does not involves with memory access.
- The number of registers is limited since only a few bits are reserved to select a register.
- Register Addressing is a form of direct addressing , this is because we are only interested in the number in the register , rather than using that number as a memory address.
- Here's an example of Register Addressing :
add \$s1 , \$s2 , \$s3 also means that $\$s1 \leftarrow \$s2 + \$s3$

where ; \$s1 = rd

\$s2 = rs

\$s3 = rt

• Immediate Addressing:

- **Immediate Addressing** is a numeric value embedded in the instruction in the actual operand.
- In **immediate addressing** , the operand is a constant within the encoded instruction.
- **Immediate addressing** has the advantage of not requiring an extra memory access to fetch the operand , hence will be executed faster. However , the size of operand is limited to 16 bits.
- The jump instruction format also falls under **immediate addressing** , where the destination is held in the instruction.
- Now an example of **Immediate Addressing** :

addi \$t1 , \$zero , 1 means $\$t1 \leftarrow 0 + 1$

(add immediate , uses the I-type format)

where ; \$t1 = rd

\$zero = r1

1 = immediate value

- **Base Addressing:**

- **Base Addressing** is a data or instruction memory location is specified as a signed offset from a register.
- **Base addressing** is also known as **indirect addressing** , where a register act as a pointer to an operand located at the memory location whose address is in the register.
- The register is called base that may point to a structure or some other collection of data and immediate value is loaded at a constant offset from the beginning of the structure. The offset specifies how far the location of the operand data from the memory location pointed by the base.
- **The address of the operand is the sum of the offset value and the base value**(rsHere's an example for **Base Addressing** :

Instruction : lw \$t1 , 4 (\$t2)

where \$t1 = rs

\$t2 = base (memory address)

4 = offset value

Thus ; \$t1 = Memory [\$t2 +4]

•

- **PC-Relative Addressing:**

- **PC-Relative Addressing** also known as **Program Counter Addressing** is a **data or instruction memory location** is specified as an offset relative to the incremented PC.
- **PC-relative addressing** is usually used **in conditional branches**. PC refers to special purpose register , Program Counter that stores the address of next instruction to be fetched.
- **The effective address is the sum of the Program Counter and offset value in the instruction.** The effective address determines the branch target.
- In **PC-relative addressing** , the offset value can be an immediate value or an interpreted label value.
- Another word of saying to explain **PC-Relative Addressing** :

The operand address = PC + an offset

Example: bne \$10, \$11, ELSE

•

- **Pseudo-direct Addressing:**

- **Pseudo-direct Addressing** is the memory address which (mostly) embedded in the instructions.
- **Pseudo-Direct** addressing is specifically used for **J-type instructions**. The instruction format is 6 bits of opcode and 26 bits for the immediate value (target).
- In **Pseudo-Direct** addressing , the **effective address** is calculated by taking the **upper 4 bits of the Program Counter(PC)** , concatenated to the 26 bit immediate value , and the lower two bits are 00.

