NETWORK COMP ADVANTAGES:

1) Communication :- Info exchange b/w computers at high speed
2) Resource sharing :- Computer on network can share I/O devices.
3) Non-local access :- By connecting Comp. over long distance, they need not be near the comp. they are using

Ethernet : Popular type of Network
→ can be a km long
→ transfer up to 40 GB/s
→ It is: LAN

LAN → Local Area Network : designed to carry data within a geography confined area, like within a building

WAN → Wide Area Network : A network extended over 100s of km's that can span over a continent.

PERFORMANCE:

i) Execution time : Its response time - The time b/w start and completion of a task

ii) THROUGHPUT : Also called bandwidth : Another measure of performance, it is the

→ No of tasks completed per unit time.

Performance = $\frac{1}{\text{Execution time}}$

X, Y are computers:

$\frac{\text{Performance } x}{\text{Performance } y} = n$

⇒ X is n times faster than Y.

Measuring Performance:

CPU EXECUTION TIME: Also called CPU time. The actual time CPU spends computing for a specific task.

USER CPU TIME: The CPU time spent in a program itself.

SYSTEM CPU TIME: The CPU time spent in the operating system performing tasks on behalf of the program.

CLOCK CYCLE: The time for one clock period, usually the process clock, which runs at a constant rate.
Also called clock or cycle or clock tick or tick.

CPU time for a program = CPU clock cycles for a program × Clock cycle time

⇒ Clock cycle time $\propto \frac{1}{\text{Clock rate}}$

⇒ CPU time for a program = $\frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$

Eg: 2GHz clock (Clock rate) ⇒ means $2 \times 10^9 \frac{\text{cycles}}{\text{second}}$

INSRUCTION PERFORMANCE:

CPU clock cycles = Instructions for a program × Average clock cycle per instruction (CPI)

CPI : Average clock cycle per instruction

Instruction count : No. of instructions executed by the program.

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

$$\text{CPU time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPI} \times \text{Instruction Count}}{\text{Clock rate}}$$

- CPI varies with Instruction mix.
- Instruction mix → measure of dynamic freq instruction across one or many programs.

RISC (Reduced Instruction Set Computer)

- ⇒ Few Instructions
- ⇒ Few Addressing modes
- ⇒ Memory access limited to LOAD & STORE
- ⇒ All operations done within registers of CPU
- ⇒ Fixed length easily decoded
- ⇒ Single cycle ~~instruct~~ Instruction execution

CISC (Complex Instruction set computer)

- ⇒ Large no. of Instructions
- ⇒ Some instructions that perform specialised task are used infrequency
- ⇒ Large variety of addressing modes
- ⇒ Variable length of instruction formats
- ⇒ Instructions that manipulate operands in memory.

MIPS ARCHITECTURE

Instruction : words of a computer language
Instruction set : Its vocabulary

Similarity of instruction sets occurs :→

- Because all computers are constructed from hardware technologies based on similar underlying principles.
- There are few basic operation that all computers must provide
- Max. performance, minimize cost & maximize energy efficiency. These things should be kept in mind while finding a language that makes it easy to build hardware.

Stored program concept :→ The idea that instructions and data of many types can be stored in memory as numbers leading to the stored program computer.

⇒ Each mips arithmetic instruction perform only 1 operation and must always have exactly three variables.

$$a = b + c + d + e$$

```
add a, b, c
add a, a, d
add a, a, e
```

- In MIPS data must be in registers to perform arithmetic.
- register $zero always equals to zero (0)
- $at register is reserved by assembler for handling larger constants.

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

⇒
```
add  t0, g, h
add  t1, i, j
sub  f, t0, t1
```

Word → The natural unit of access in a computer, usually a group of 32 bits, corresponds to the size of bit

Registers $s0 – $s7	map to 16 – 23
Registers $t0 – $t7	map to 8 – 15

⇒ Reason for 32 Registers → Smaller is faster
→ A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

⇒ $s0 $s1 ... : correspond to the variables
$t0, $t1 ... : correspond to temporary registers

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

$f, g, h, i, j \rightarrow$ are assigned to $\$s_0$, $\$s_1$, $\$s_2$, $\$s_3$, $\$s_4$

$\Longrightarrow$ add $\$t_0, \$s_1, \$s_2$
add $\$t_1, \$s_3, \$s_4$
sub $\$s_0, \$t_0, \$t_1$

- Processor can only keep small amount of data in registers, but computer memory contains billions of data elements.
- Hence, data structures are kept in memory.

data-transfer instruction: a command that moves data between memory & register.

$g = h + A[8]$

Compiler has associated $g, h \rightarrow \$s_1, \$s_2$
Let base address of array is $\$s_3$.

lw $\$t_0, 8(\$s_3)$ [$\because$ 8 = offset added to base register]
add $\$s_1, \$s_2, \$t_0$

Big Endian & Little Endian
(We assume a 4 byte word)

POPU

Page No.:
Date: / /

Big ENDIAN: Bytes are order from LEFT TO RIGHT

Eg: MIPS

0	0	1	2	3	Word 0
4	4	5	6	7	Word 1
8	8	9	10	11	Word 2
12	12	13	14	15	Word 3

address of Word

(address of the word is the address of the starting byte)

Little ENDIAN: Bytes are ordered from RIGHT TO LEFT

0	3	2	1	0	Word 0
4	7	6	5	4	Word 1
8	11	10	9	8	Word 2
12	15	14	13	12	Word 3

↳ So to get proper byte address offset has to be added to the base register

Variable h is associated with $s2 and base address of A is in $s3

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

lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

- Registers take less time.

Throughput of Registers > Throughput of memory

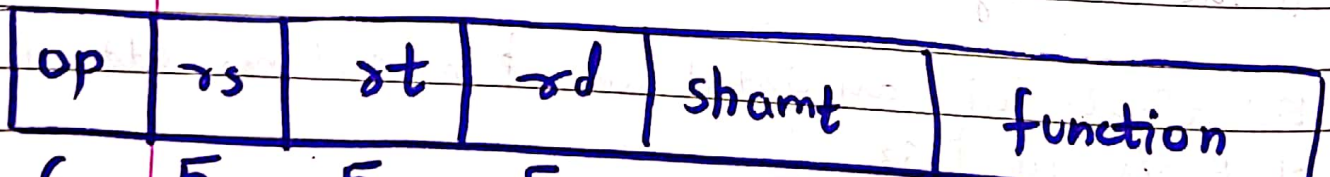
- Accessing registers uses less energy than accessing memory.

addi → add immediate
Quick add instruction with one constant operand is called addi

addi $s3, $s3, 4 # $s3 = $s3 + 4

- mov operation is add operation where one operand is zero.
- There is NO subi in MIPS because MIPS supports negative constants.

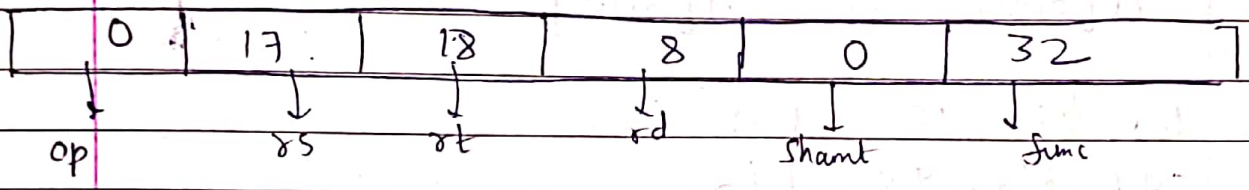
MIPS FIELDS



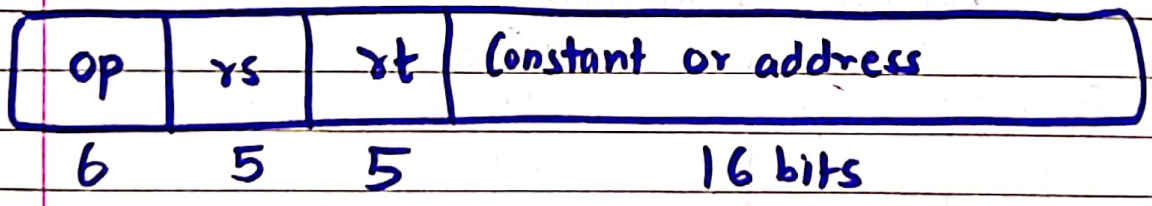
(R-TYPE format)
→ for register

op → OPCODE
rs → First register source operand
rt → Second register source operand
rd → register destination
Shamt → Shift amount
funct → function

add $t0, $s1, $s2



- If the addess were to use one of the 5-bit fields in the fomat above, the constant within the load word instruction would be limited to 25 or 32. This constant is used to select elements from arrays or data structures, it often needs to be much larger than 32. This 5 bit is too small to be useful.



(I-type format)

→ for immediate

↳ This type used for load word (lw), store word (sw), addi.

POPU

Page No.
Date : / /

→ First three fields of R-type & I-type is same size & same names, three length of 4th field of I type is equal to sum of lengths of last three fields of R-type.

Base array A → $t1
$s2 → h

A[300] = h + A[300]

lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)

lw	35	9	8	1200		
add	0	18	8	8	0	32
sw	43	9	8	1200		

LOGICAL OPERATION:

	MIPS
Shift left	SLL
Shift right	SRL
Bit by Bit AND	and, andi
" OR	or, ori
" NOT	nor

Set on less than :→

Slt → compares two registers and sets third register to 1 if first is less than second otherwise sets it to 0.

```
slt  $t0, $s3, $s4   # $t0=1 if $s3 < $s4
```

```
slti $t0, $s2, 10    # $t0=1 if $s2<10
```
(slti: immediate version)

- slt → for signed integers.
- For unsigned we have → sltu
  sltui

MIPS provides instructions to move bytes :→

→ Load byte (lb): loads a byte from memory, placing it in the rightmost 8 bits of a register

→ Store byte (sb): Takes a byte from the rightmost 8 bits of a register and write it to the memory.

```
lb $t0, 0($sp)
sb $t0, 0($gp)
```

POPU

Page No.:
Date: / /

Code:→

```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
}
```

- x → $a0 , y → $a1 , i → $s0

```
strcpy:
  addi $sp, $sp, -4      # adjust stack for 1 more item
  sw   $s0, 0($sp)
  add  $s0, $zero, $zero # i = 0+0
L1: add  $t1, $s0, $a1   # address of y[i] in $t1
  lbu  $t2, 0($t1)       # $t2 = y[i]
  add  $t3, $s0, $a0     # address of x[i] in $t3
  sb   $t2, 0($t3)       # x[i] = y[i]
  beq  $t2, $zero, L2    # if y[i] == 0, go to L2
  addi $s0, $s0, 1       # i = i+1
  j    L1                # go to L1
L2: lw   $s0, 0($sp)     # y[i] == 0
                         # Restore old $s0
  addi $sp, $sp, 4       # pop 1 word off stack
  jr   $ra               # return
```

POPU
Page No.:
Date: / /

Halfwords :-

MIPS instruction set has explicit instructions to load and store 16 bit quantities, called halfwords.

Load half (Lh): loads halfword from memory placing in rightmost 16 bits of register. It treats number as signed.

Load halfword unsigned (lhu): Works with unsigned numbers.

Store half (sh): Takes a halfword from the right-most 16 bits of register and writes it to the memory.

```
lhu  $t0, 0($sp)   # Read halfword from 16 bit source
sh   $t0, 0($gp)   # Write halfword (16 bits)
```

32-bit immediate Operand:

→ Although constants are frequently 8 bit or 16 bit fields, sometimes they are bigger.

Load upper immediate (lui): Used specifically to set the upper 16 bits of constant in a register, allowing subsequent instruction to specify the lower 16 bit of the constant.

Eg: We have a 32 bit constant:

0000 0000 0011 1101 0000 1001 0000 0000

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

(After this value of s0 is:
0000 0000 0011 1100 0000 0000 0000 0000)

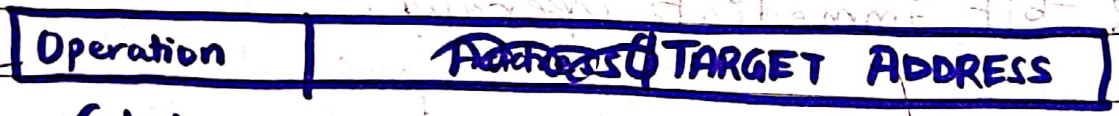
(Now inserting the lower 16 bits:

Ori $s0, $s0, 2304

★ $at → register reserved for the assembler

↳ temporary register to create long values.

ADDRESSING IN BRANCHES & JUMPS:



Opcode

(J-type format)

↳ Its a word address, meaning it represents 32-bit byte adress.

JUMP opcode = 2

Program Counter = Register + Branch Address

Addressing using program counter
= ~~PROGRAM COUNTER RELATIVE ADRESSING~~
= PC-RELATIVE ADRESSING

Register $zero: It has the value 0 &
programmer cannot change the value of register $zero

→ Register $zero is used to create assembly language instruction that copies contents of one register to another.

move $t0, $t1 # register $t0 gets register $t1

↓ equivalent instruction

add $t0, $zero, $t1

MIPS converts blt → Slt + bne

These are PSEUDOINSTRUCTIONS

MIPS assemblers use: HEXADECIMAL

POPU

Page No.:
Date: / /

ARRAYS

```
clear1 (int array[], int size)
{
    int i;
    for (i=0; i<size; i+=1)
    {
        array[i] = 0;
    }
}
```

→ array and size → $a0 & $a1
i → $t0

```
move  $t0, $zero   # i=0
loop1: sll  $t1, $t0, 2    # $t1 = i x 4
       add  $t2, $a0, $t1  # $t2 = address of array[i]
       sw   $zero, 0($t2)  # array[i] = 0
       slt  $t3, $t0, $a1  # $t3 = (i < size)
       bne  $t3, $zero, loop1  # if (i < size) go to loop1
       addi $t0, $t0, 1    # i = i+1
```

```
clear2 (int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p+1)
        *p = 0;
}
```

```
        move $t0, $a0   # p = address of array[0]
loop2:  sw   $zero, 0($t0)   # Memory[p] = 0
        addi $t0, $t0, 4     # p = p+4
        ~~add  $t~~
        sll  $t1, $a1, 2     # $t1 = size * 4
        add  $t2, $a0, $t1   # $t2 = address of array[size]
        slt  $t3, $t0, $t2   # $t3 = (p < &array[size])
        bne  $t3, $zero, loop2  # if (p < &array[size]) goto
                                  loop2
```