INTRODUCTION TO COMPUTER ORGANIZATION

CCIT4026

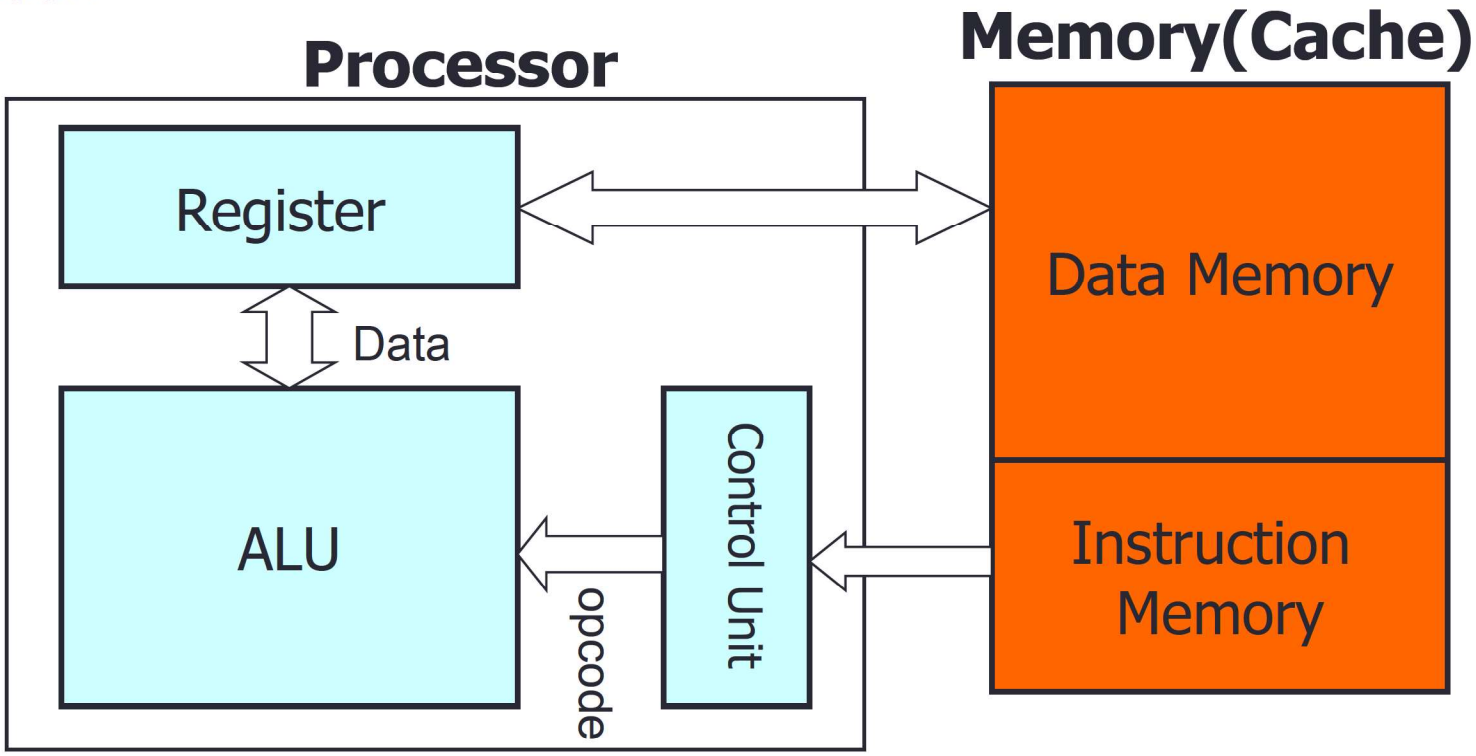
Chapter 3: Instruction Set Design (Part 2)

Outline

- Data in MIPS
- Basic MIPS instruction
- Pseudo-instructions
- Logic Operation
- Data Transfer in MIPS
- Machine Instruction

Data in MIPS

- Data and instructions are in different places during MIPS instruction execution



- All memory data should load into register before used in MIPS instructions! Data should write back to memory after instruction execution.

Memory vs. Register vs. Constant

- Major difference is the instruction's execution time
  - **(1) Memory** is outside the processor; far from the processing unit
    - Memory operand takes a long time to get the value
  - **(2) Register** is inside the processor; close to the processing unit
    - Register operand takes a short time to get the value
  - **(3) Constant** already coded in the instruction
    - Constant operand value is immediately available

A program is a mixture of these three types of operations

If you are to optimize the program running time, what should you do?

Basic MIPS arithmetic instructions

- Addition

```
add $t0, $t1, $t2          # $t0 = $t1 + $t2
addi $t0, $s1, 10          # $t0 = $s1 + 10
addi $t0, $t0, -1          # $t0 = $t0 + (–1)
```

  - addi means add immediate (constant), the constant part is always the last operand of this instruction

- Subtraction

```
sub $t0, $t1, $t2          # $t0 = $t1 – $t2
addi $t0, $t1, -1          # $t0 = $t1 – 1
```

- Multiplication

```
mul $t0, $t1, $t2          # $t1 * $t2, result in $HI:$t0
```

  Or

```
mult $t1, $t2              # $t1 * $t2 result in $HI:$LO
```

- Division

```
div $t1, $t2               # $t1 / $t2, quotient in $LO, remainder in $HI
```

Basic MIPS arithmetic instructions

- Various ways to initialize a register with zero or some constants:

```
a) sub  $t0, $t0, $t0          # $t0 = $t0 - $t0 =0
b) add  $t0, $zero, $zero      # $t0 = 0
c) add  $t0, $0, $0            # $t0 = 0
d) addi $t0, $zero, 5          # $t0 = 0 + 5 = 5
```

- Move values from HI and LO
  - `mfhi $t1` # $t1 = $HI
  - `mflo $t1` # $t1 = $LO

Basic MIPS instruction

```
.data
msg0: .asciiz "Adding numbers:\n"
msg1: .asciiz "A? "
msg2: .asciiz "B? "
result: .asciiz "Sum = "
newline: .asciiz "\n"

.text
.globl __start
__start:

# Print "Adding numbers\n"
la $a0, msg0
li $v0, 4
syscall

# Print "A? "
la $a0, msg1
li $v0, 4
syscall

# Get the value for A (store in
$v0)
li $v0, 5
syscall

# Move $v0 to $t0: $t0 = 0 +
$v0
add $t0, $zero, $v0

# Print "B? "
la $a0, msg2
li $v0, 4
syscall

# Get the value for B (store
in $v0)
li $v0, 5
syscall

# Add A and B: $t0 = $t0 +
$v0
add $t0, $t0, $v0

# Print "Sum = "
la $a0, result
li $v0, 4
syscall

# Print result ($t0 is
the result)
add $a0, $zero, $t0
li $v0, 1
syscall

# Print a new line
la $a0, newline
li $v0, 4
syscall

# Terminate the program
li $v0, 10
syscall
```

Pseudo-instructions

Pseudo-instructions

- Pseudo-instructions are simple assembly language instructions that do not have direct machine instructions equivalent (i.e., they need not be implemented directly in hardware)

- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually **translated** into one or more equivalent actual machine language instructions

- Pseudo-instructions give MIPS a richer set of assembly language instructions

- The only cost is reserving one register, `$at`, for using by the assembler

- E.g. `move, li, la, bgt, bge, blt, ble, ……..`

Pseudo-instructions

Copy $t1 into $t0 => ($t0 = $t1 + 0)

```
move $t0, $t1    →    add  $t0, $t1, $zero
```

Load immediate 2000 into $a0 => $a0 = $0 + 2000

```
li $a0, 2000     →    addiu  $a0, $0, 2000
```

Load address into $a0

```
la $a0, 0x1000056c
```

↓

```
lui $a0, 0x1000          #set the most significant 16 bits of the address
ori $a0, $a0, 0x056c     #set the least significant 16 bits of the address
```

Pseudo-instructions – Example 1

```
        .data
age:    .word 1
        ……..
abc:    .word 10

        .text
        li $a1, 2
        la $s0, abc
```

Labels

Label ▲	Address
a.asm	
abc	0x10010038
age	0x10010000
End	0x00400058
gpa	0x10010028
kb	0x1001002a

```
0x00400068  0x24050002  addiu $5,$0,0x00000002   44:   li   $a1, 2
0x0040006c  0x3c011001  lui $1,0x00001001        45:   la   $s0, abc
0x00400070  0x34300038  ori $16,$1,0x00000038
```

Pseudo-Instructions	Actual Instructions
li $a1, 2	addiu $a1, $zero, 2
la $s0, abc	lui $at, 0x00001001 ori $s0, $at, 0x00000038

Pseudo-instructions – Example 2

```
            # initialize counter value
            li          $t0,0                       # this is the loop index
__loop:
            bge         $t0,10,__continue           # exit loop if value > 10
            li          $v0,1
            move        $a0,$t0                     # put value into $a0 for printing
            syscall
            addi        $t0,$t0,1                   # increment $t0 by 1
            j           __loop
__continue:
```

0x00400010	0x24080000	addiu $8,$0,0x00000000	12:	li	$t0,0
0x00400014	0x2901000a	slti $1,$8,0x0000000a	14:	bge	$t0,10,__continue
0x00400018	0x10200005	beq $1,$0,0x00000005			
0x0040001c	0x24020001	addiu $2,$0,0x00000001	15:	li	$v0,1
0x00400020	0x00082021	addu $4,$0,$8	16:	move	$a0,$t0
0x00400024	0x0000000c	syscall	17:	syscall	
0x00400028	0x21080001	addi $8,$8,0x00000001	18:	addi	$t0,$t0,1
0x0040002c	0x08100005	j 0x00400014	19:	j	__loop
0x00400030	0x24020004	addiu $2,$0,0x00000004	21:	li	$v0,4

Notes:

- $at is $1 and $t0 is $8
- slti -> if $t0 < 10, then set $at = 1, else set $at = 0.
- beq -> if $at==$0, branch to "__continue" else continue.
- Address of "__continue" is PC+4*5 = 0x0040001c + 0x14 = 0x00400030

Logic Operation

- Bit-wise logical operations: **`and, or, not, nor, xor`**

A	B	and	or	not A	nor	xor
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	0	0	0

- **Example**
  - given register $t1 and $t2 ,

If word length is 16 bits,

$t1 = 0011 1100 0000 0000$_2$
$t2 = 0000 1101 0000 0000$_2$

- `and  $t0, $t1, $t2` → $t0 = 0000 1100 0000 0000$_2$
- `or  $t0, $t1, $t2` → $t0 = 0011 1101 0000 0000$_2$
- `nor  $t0, $t1, $zero` → $t0 = 1100 0011 1111 1111$_2$

Logic Operation

- Shift operations: `sll, srl`
- Move all the bits in a word to the left or right
- Filling the emptied bits with 0s
- Example

  `0000 0000 0000 0000 0000 0000 0000 1001` = 9_{10}

  shift left (<<) by 4

  `0000 0000 0000 0000 0000 0000 1001 0000` = 144_{10}

  shifting left by k bits gives the same result as multiplying by 2^k

MIPS shift instructions:

`sll` ('shift left logical'), `srl` ('shift right logical')

- Example

```
sll $t2, $s0, 4      # reg $s0 << 4 bits
```

Data Transfer in MIPS

What if a program manipulates a large array of elements?

- Not possible to store elements all at once in registers inside processor
- Large data structures like arrays are kept in memory
  - memory provides large storage for millions of data elements.

MIPS' design disallows values stored in memory to be manipulated directly

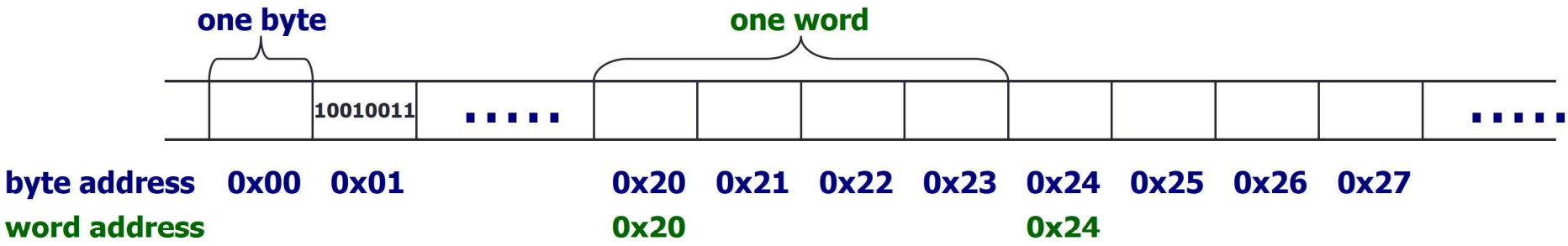
- Because running arithmetic instruction directly on memory makes pipelining the processor difficult

Then, how does MIPS read/write them?

- Data transfer instructions
  - **load** moves data from memory to a register, e.g. `lw` (load a word)
  - **store** moves data from a register to memory, e.g. `sw` (store a word)

Data Transfer in MIPS

- **Memory** is a consecutive arrangement of storage locations.



- Each memory location is indexed by an **address**.
  - Addresses of sequential 8-bit bytes differ by 1
  - Addresses of sequential 32-bit words differ by 4

- To specify the address of the memory location of any array element in assembly language, we need two parts:
  - **Base address**: starting address of an array
  - **Offset**: distance of target location from starting address. It is a constant that can be either positive or negative

Data Transfer in MIPS

`A` is an array of 100 words

- How can we perform `A[13] = h + A[8];` in MIPS?

Assume `$s0` store the starting location of array `A` (i.e., address of A[0])
Assume `$s1` store the value of `h`

- **Answer**:

```
# temp reg $t0 gets A[8]
# (i.e., address of A[8] = address of A[0] + 8*4)
lw  $t0, 32($s0)
# $t0 = h + A[8]
add $t0, $s1, $t0
# stores h + A[8] to A[13]
# (i.e., address of A[13] = address of A[0] + 13*4)
sw  $t0, 52($s0)
```

Data Transfer in MIPS

```
##############
#  We need to declare "variables" & "Arrays" used in the program in a data segment.
#  The compiler recognize .data as the beginning of data segment
.data
h:  .word 1 2 3 4  # h is an array of size 4, each element is a word (32 bit)
s:  .word 5

# The 3 lines below let the system know the program begins here
.text
.globl __start
__start:
```

Assume the address of h is 0x1000

```
# Write your program code here
la $s0, h                  # Obtain starting address of array h,  s0 = x (a constant)
lw $s1,8($s0)              # $s1 = content in address (= (x + 2*4) = address of h[2]) = 3
la $s2, s                   # load the address of the word s to $s2
lw $s3, -12($s2)           # $s3 = content in (address of (s – 3*4)) = ?

sub $s3, $s3, $s1          # Q1: Guess what is the value in $s3 ?

sw $s3, 0($s0)             # Q2: How are the values of array h changed ?
```

Common MIPS instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	3 operands
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	3 operands
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	2 operands, 1 constant
Logical	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	3 operands, bit-by-bit and
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	3 operands, bit-by-bit or
	nor	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)	3 operands, bit-by-bit or
	and immediate	andi $s1, $s2, 100	$s1 = $s2 & 100	2 operands, 1 constant, bit-by-bit
	or immediate	ori $s1, $s2, 100	$s1 = $s2 | 100	2 operands, 1 constant, bit-by-bit
	shift left logical	sll $s1, $s2, 10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1, $s2, 10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw $s1, 100($s2)	$s1 = memory[$2+100]	Word from mem to reg
	store word	sw $s1, 100($s2)	Memory[$2+100] = $1	Word from reg to mem

Machine Instruction

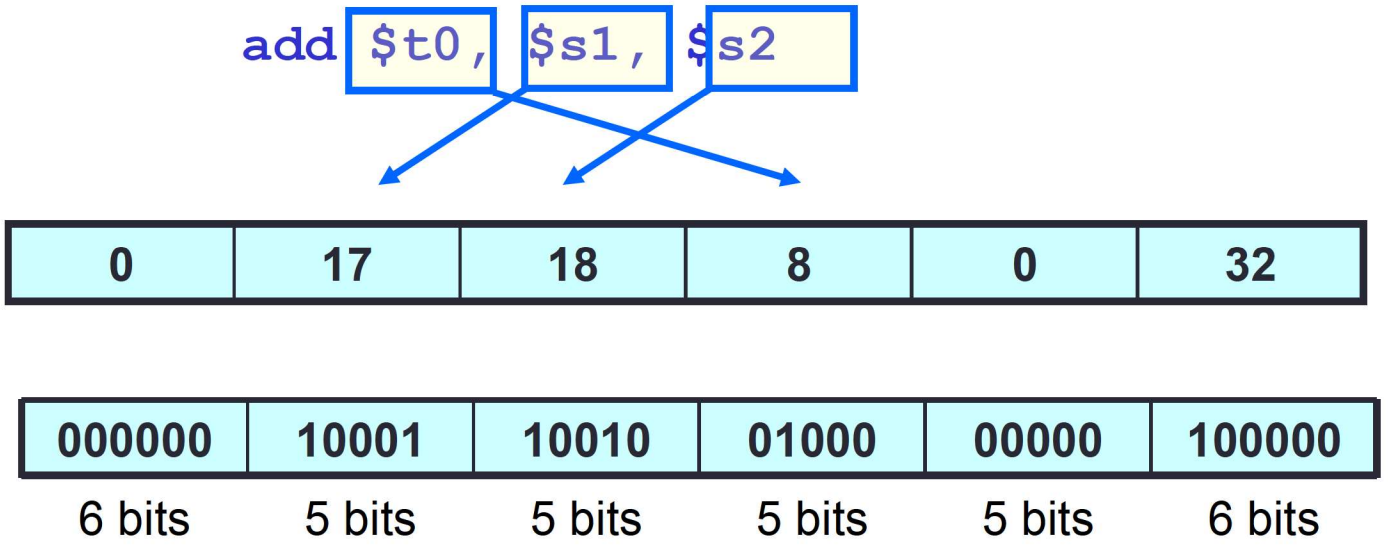
How does the computer "see" the instructions?

- Machine language or machine code; represented as binary numbers
  - Numeric data are kept in computer as a series of **high** & **low** electronic signals – binary numbers

What is the format of the machine code?

- **All MIPS instructions are 32-bit long**, broken up into a number of fields

Example:

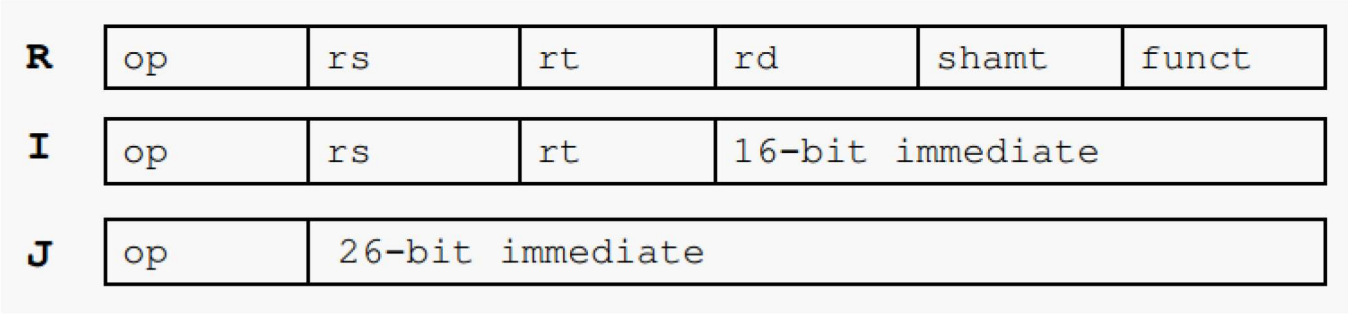


MIPS registers (recall from Ch.2)

Register number	Register name	Usage
0	`zero`	Always zero
1	`$at`	Reserved for the assembler
2 - 3	`$v0 - $v1`	Expression evaluation and procedure return results
4 - 7	`$a0 - $a3`	The first four parameters passed to a procedure. Not preserved across procedure calls
8 - 15	`$t0 - $t7`	Temporary registers. Not preserved across procedure calls
16 - 23	`$s0 - $s7`	Saved values. Preserved across procedure calls
24 - 25	`$t8 - $t9`	Temporary registers. Not preserved across procedure calls
26 - 27	`$k0 - $k1`	Reserved for kernel usage.
28	`$gp`	Global pointer (pointer to global area)
29	`$sp`	Stack pointer
30	`$fp`	Frame pointer[a]
31	`$ra`	Return address

Instruction Format

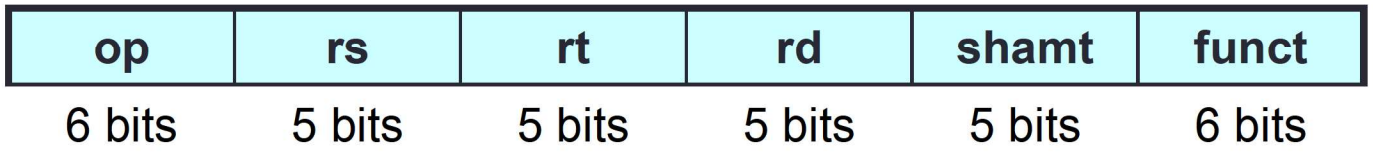
- All MIPS instructions have fixed length (32-bit), but different instructions may have different formats
- Three types of instruction formats in MIPS
  - R-type or R-format for register
  - I-type or I-format for immediate
  - J-type or J-format for immediate



- Each format is assigned a distinct set of values for the 1st field (op)
  - Hardware can interpret the instruction just by examining this field

R-type Instruction Format

R-type or R-format

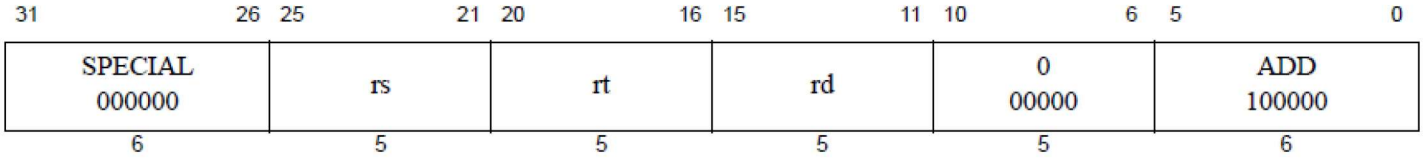


- **Instruction fields**: (6 fields)
  - **op**: basic operation of instruction, traditionally called **opcode,** which is set to 000000 in R-format
  - **rs**: first register source operand
  - **rt**: second register source operand
  - **rd**: register destination operand, which gets result of operation
  - **shamt**: shift amount (number of positions to shift)
  - **funct**: function code selecting the specific variant of the opcode

R-type Instruction Format

- Example

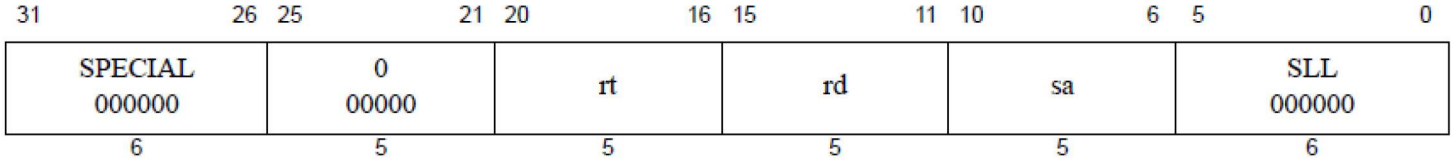
(1) add $t1, $s0, $t7



Format: `ADD rd, rs, rt` **MIPS32**

000000	10000	01111	01001	00000	100000

(2) sll $t2, $v1, 12

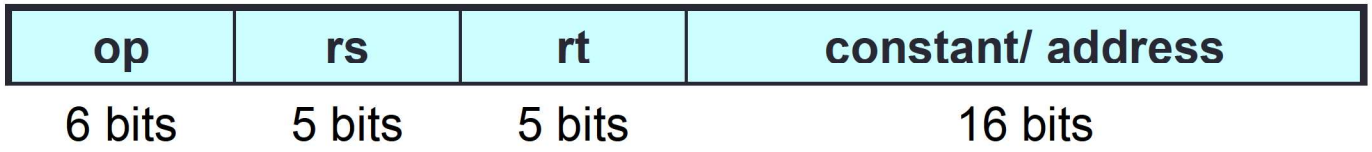


Format: `SLL rd, rt, sa` **MIPS32**

000000	00000	00011	01010	01100	000000

I-type Instruction Format

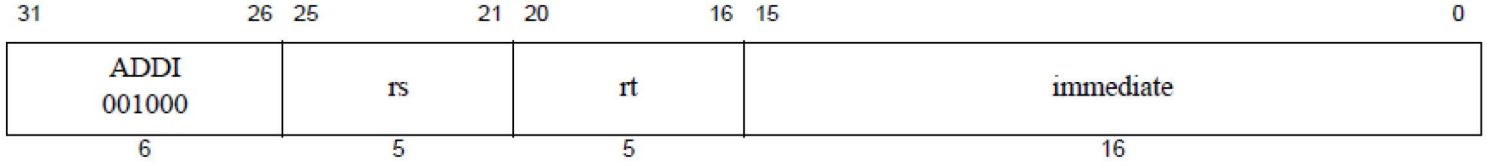
I-type or I-format



- **Instruction fields**:
  - **op**: as before
  - **rs**: base register
  - **rt**: register source operand (for `sw`)
    - *or* destination operand (for `lw`)
  - **constant/ address**: 16-bit address offset from the starting address
    - needed for data transfer instructions
    - 5-bit field in the R-format is too small for specifying the offset for reasonably sized arrays.

I-type Instruction Format

- Example
- addi $s0. $0. -10



001000	00000	10000	1111 1111 1111 0110

- lw $s3, 50($t2)

100011	01010	10011	0000 0000 0011 0010

Machine code in hexadecimal: 0x 8D 53 00 32

Machine instruction (summary)

Instruction	Type	op	rs	rt	rd	shamt	funct	const/address
`add`	R	0	reg	reg	reg	0	32_{10}	-
`sub`	R	0	reg	reg	reg	0	34_{10}	-
`and`	R	0	reg	reg	reg	0	36_{10}	-
`or`	R	0	reg	reg	reg	0	37_{10}	-
`sll`	R	0	0	reg	reg	constant	0	-
`srl`	R	0	0	reg	reg	constant	2_{10}	-
`addi`	I	8_{10}	reg	reg	-	-		constant
`lw`	I	35_{10}	reg	reg	-	-		address
`sw`	I	43_{10}	reg	reg	-	-		address

Machine instruction (Example)

- **Description**:
  - Suppose `$t1` stores the base address of word array `A` and `$s2` is associated with `h`, the following C assignment statement

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

  is compiled into

```
lw  $t0, ????(???)   # Temporary reg $t0 gets A[300]
add $t0, ???, ???    # Temporary reg $t0 gets h + A[300]
sw  $t0, ????(???)   # Store h + A[300] back into A[300]
```

Problem to solve:

- What is the MIPS machine code for these three instructions?

Machine instruction (Example)

- **Translate MIPS Assembly language into machine code**