CSE 215
Computer Architecture

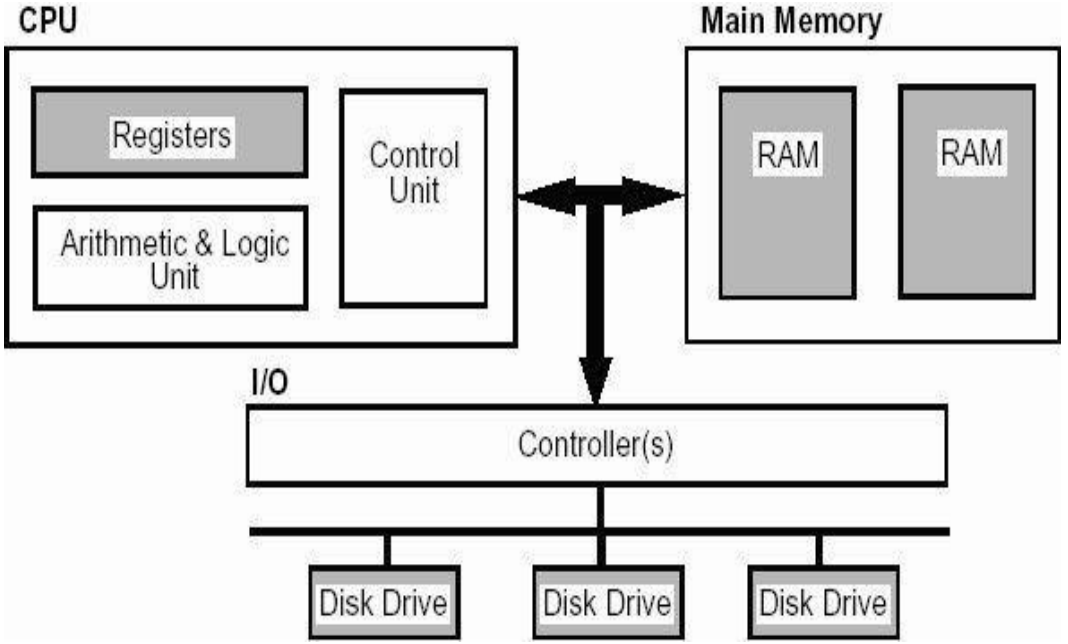
Nasrin Akter
Bangladesh University of Business & Technology

Introduction

To understand how the CPU operates, let's look at how an Instruction is executed.

Program is saved in the memory .e.g. Whatsapp is stored in the memory and whatsapp messages are also stored in the memory.

The job of the processor is to execute the program which is stored in the memory



Continue

- **Machine Language:** A CPU can only execute machine language instructions. As we've seen, they are bit strings.

- *Machine instruction* *Operation*
- 10100001 00000000 00000000 Fetch the contents of memory word 0 and put it in register AX.
- ***Assembly Language***
- In assembly language, we use symbolic names to represent operations, registers, and memory locations. If location 0 is symbolized by A,
- *Assembly language instruction* *Comment*
- MOV .AX,A ;fetch the contents of location A and put it *in register AX*

Instruction Set

Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Four principles of IS architecture
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

MIPS

- MIPS 32 ISA defines the following CPU registers that are visible to the assembly language programmer.
- 32,32 bit general purpose register(R0-R31)
- A special purpose 32 bit program counter
  - Points to the next instruction in memory to be fetched and executed.
  - Not directly visible to the programmer
  - Affected only indirectly by certain instructions
- A pair of 32 bit special purpose registers HI and LO which are used to hold the results of multiply, divide and multiply-accumulate instructions.
- Instructions are all 32 bits
- byte(8 bits), halfword (2 bytes), word (4 bytes)
- a character requires 1 byte of storage
- an integer requires 1 word (4 bytes) of storage

Registers—MIPS*

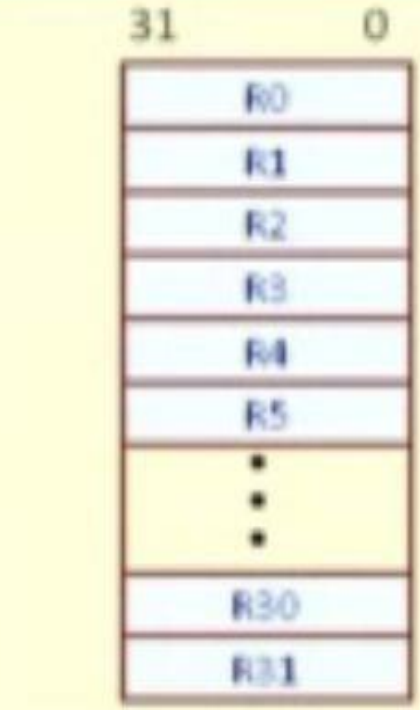
Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions - not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

*

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```



General Purpose Registers

Special Purpose Registers

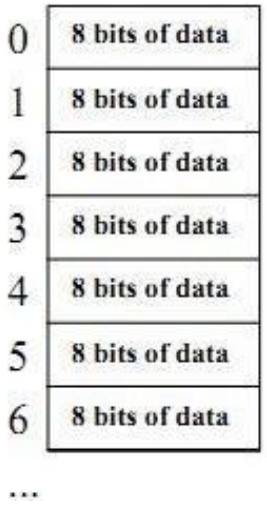
Two of the GPRs have assigned functions:

a) R0 is hard-wired to a value of zero.
   - Can be used as the target register for any instruction whose result is to be discarded.
   - Can also be used as a source when a zero value is needed.

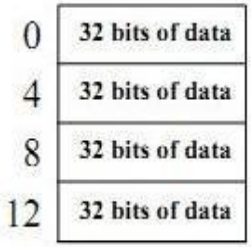
b) R31 is used to store the return address when a function call is made.
   - Used by the jump-and-link and branch-and-link instructions like JAL, BLTZAL, BGEZAL, etc.
   - Can also be used as a normal register.

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Instruction format

- An instruction format defines the layout of the bits of an instruction, in terms of its constituents parts. It contains **two** parts-opcode and, implicitly or explicitly, zero or more operands.

Opcode:

- Specifies the operation to be performed by the instruction
- Various categories of instructions-data transfer, arithmetic and logical control, I/O and special machine control

Operands

- Specifies source and destinations of the operation.
- Source can be specified by immediate, by naming a register or specifying a memory address.
- Destination can be specified by a register or memory address.

Accessing the operands

- operands are generally in one of two places:
  - registers (32 int, 32 fp)
  - memory (2^{32} locations)
- registers are
  - easy to specify
  - close to the processor (fast access)
- the idea that we want to access registers whenever possible led to *load-store architectures*.
  - normal arithmetic instructions only access registers
  - only access memory with explicit loads and stores

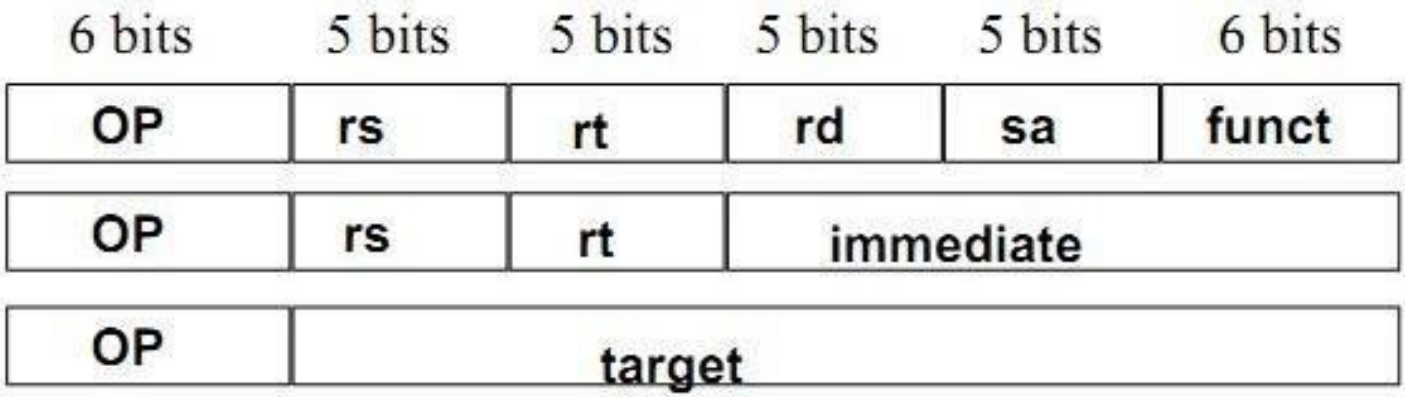
Instruction Format

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	address/immediate (16)			Load/store, branch, immediate
J-format	op	target address (26)					Jump

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R stands for Register
I stands for Immediate
J stands for Jump

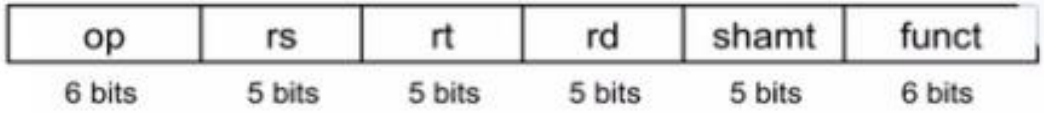
MIPS Instruction Formats



- the opcode tells the machine which format
- so add r1, r2, r3 has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - 000000 00010 00011 00001 00000 100000

- So LB R4,50(R3) has
- Opcode=32,immediate=50(110010),rt=4,rs=3
- -100000 00011 00100 110010

R-format Example



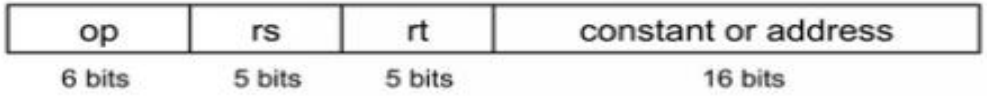
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions



op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

Addressing Modes

MIPS addressing modes

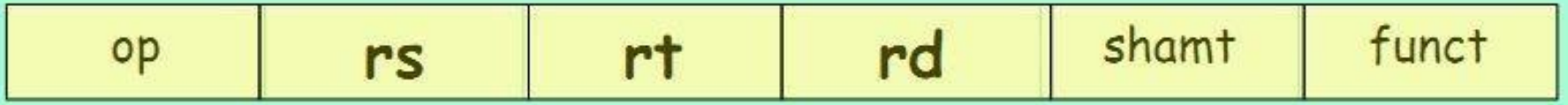
Addressing modes are the ways of specifying an operand or a memory address.

- Register addressing
- Immediate addressing
- Base addressing
- PC-relative addressing
- Indirect addressing
- Direct addressing (almost)

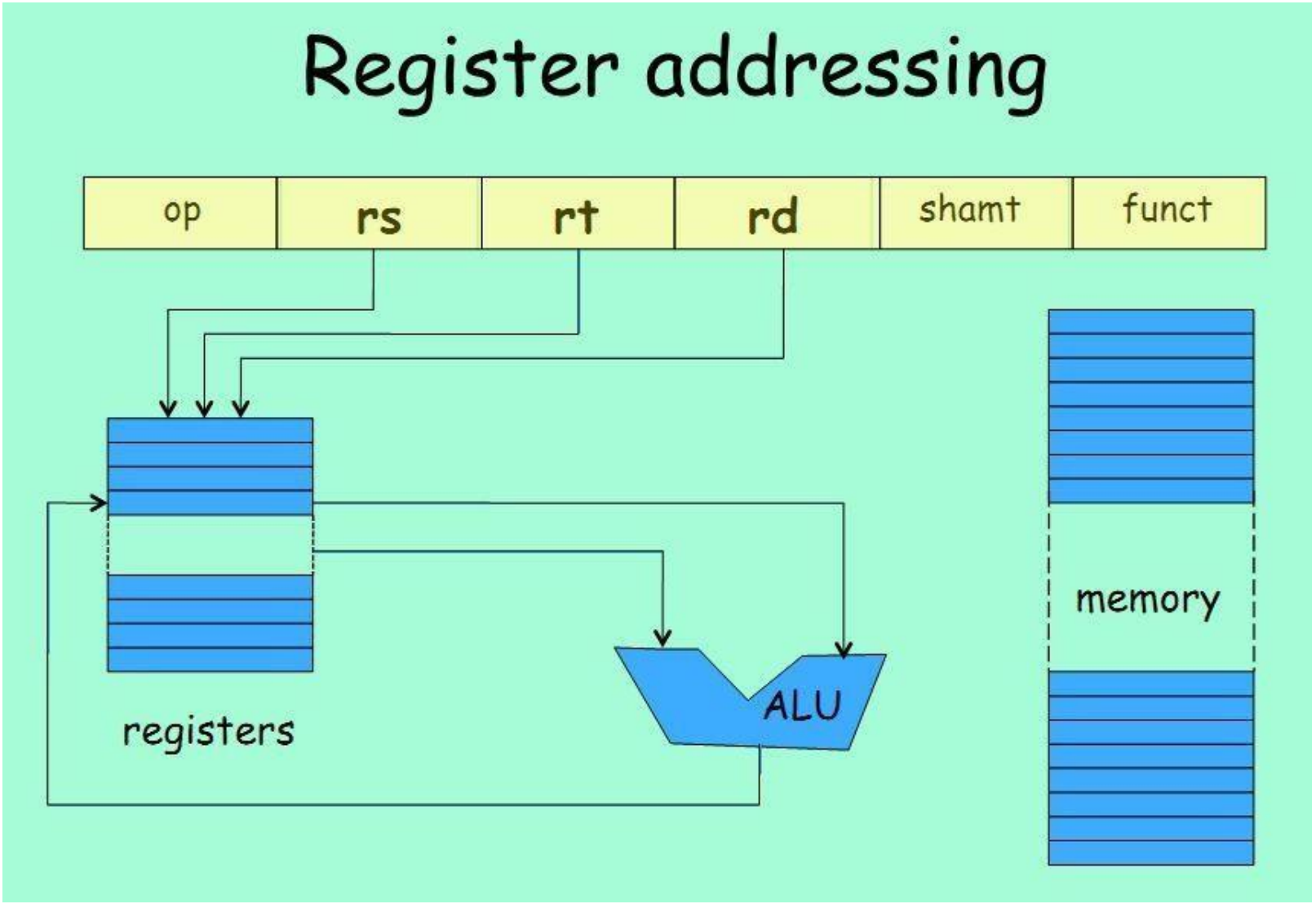
Addressing Modes

Register addressing

- Operands are in a register.
- Example: add $3,$4,$5
- Takes n bits to address 2^n registers



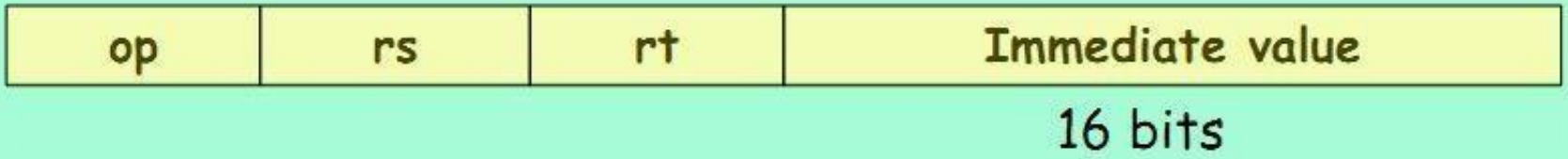
Addressing Modes



Addressing Modes

Immediate Addressing

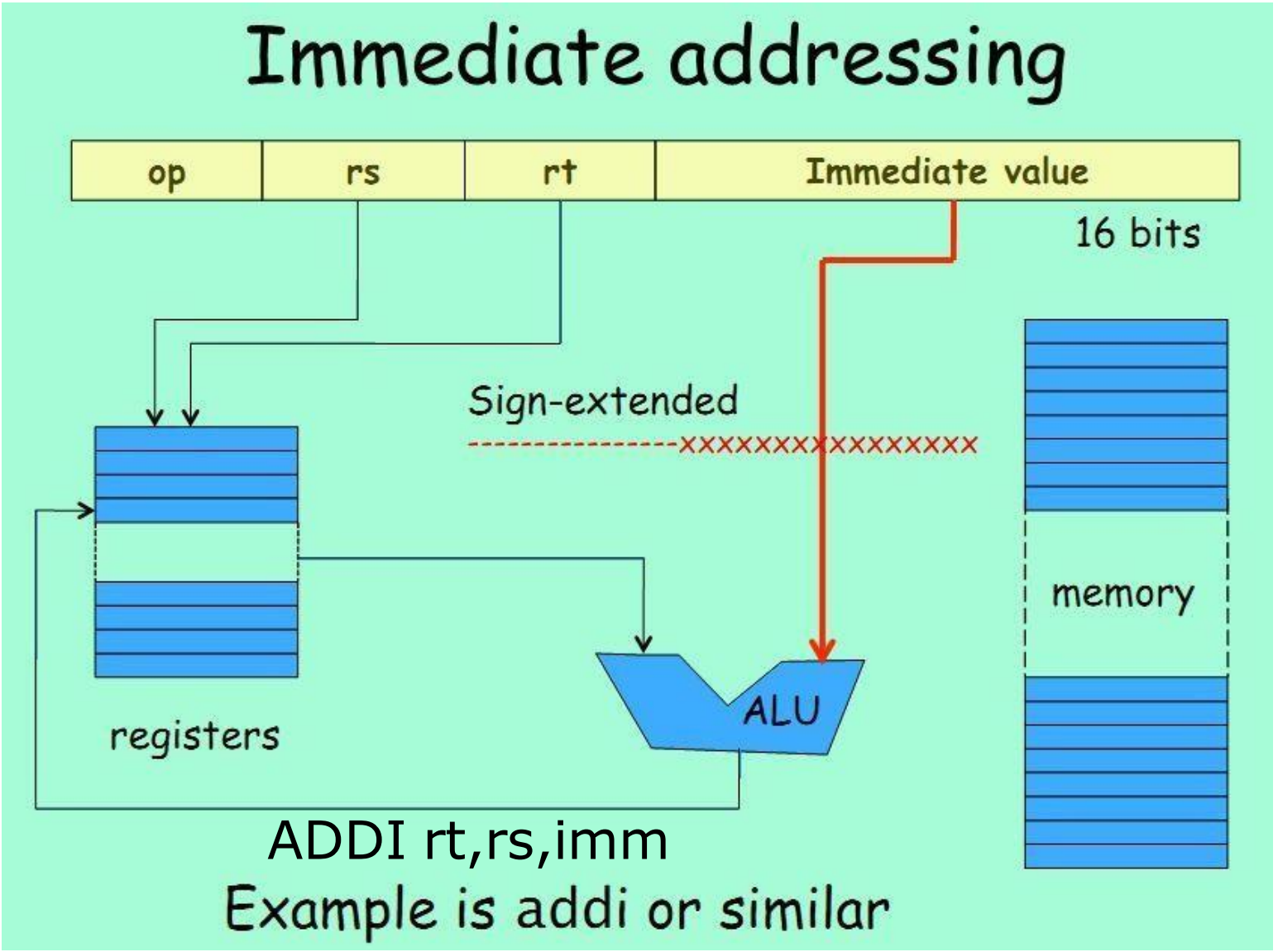
- The operand is embedded inside the encoded instruction.



16 bit two's-complement number:

$-2^{15} - 1 = -32{,}769 < \text{value} < +2^{15} = +32{,}768$

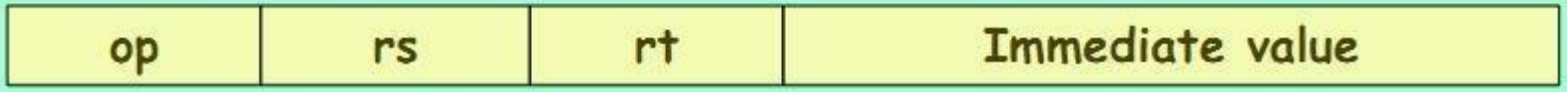
Addressing Modes



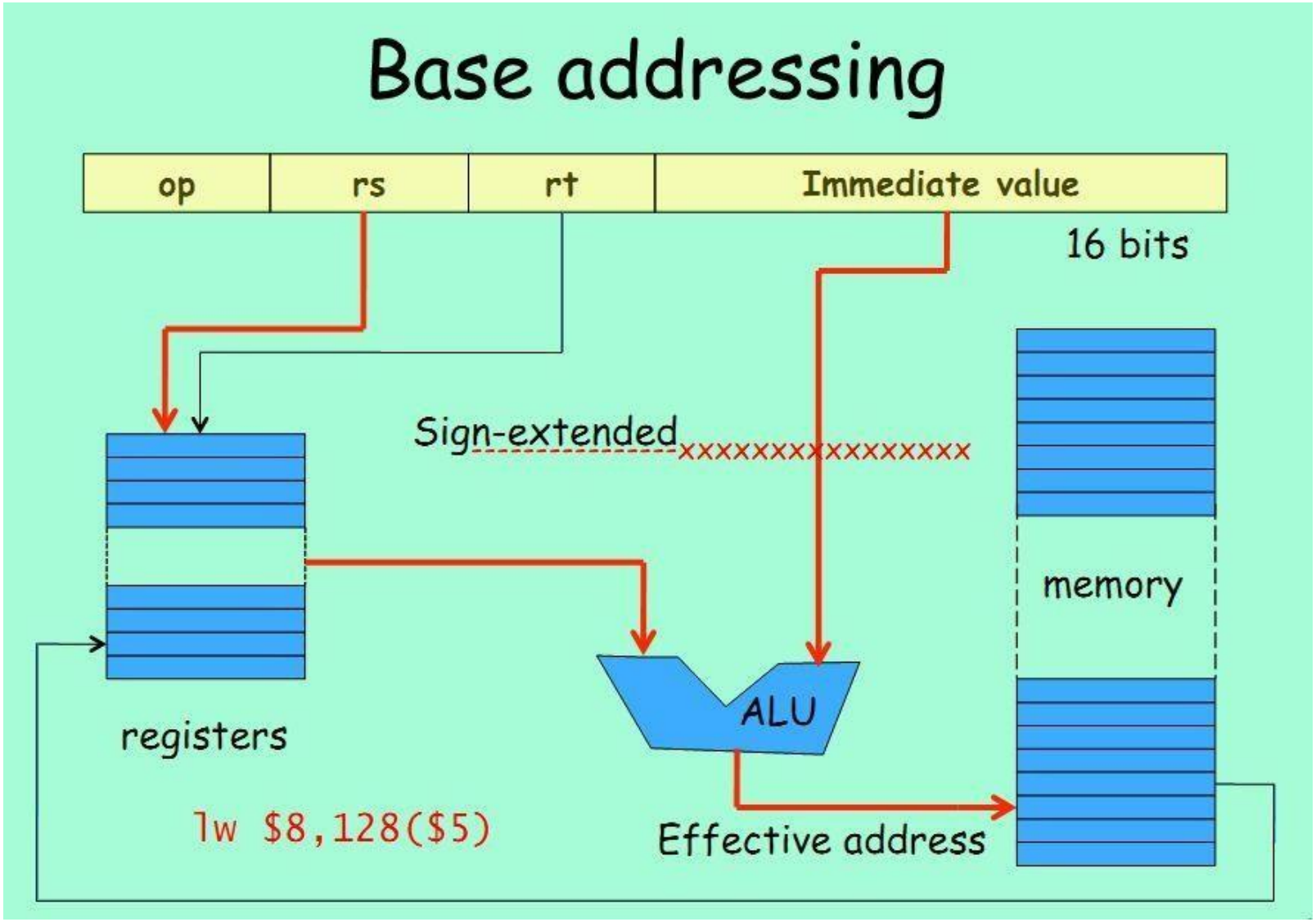
Addressing Modes

Base (or Base-offset or displacement) Addressing

- The address of the operand is the sum of the immediate and the value in a register (rs).
- 16-bit immediate is a two's complement number
- Ex: lw $15,16($12)

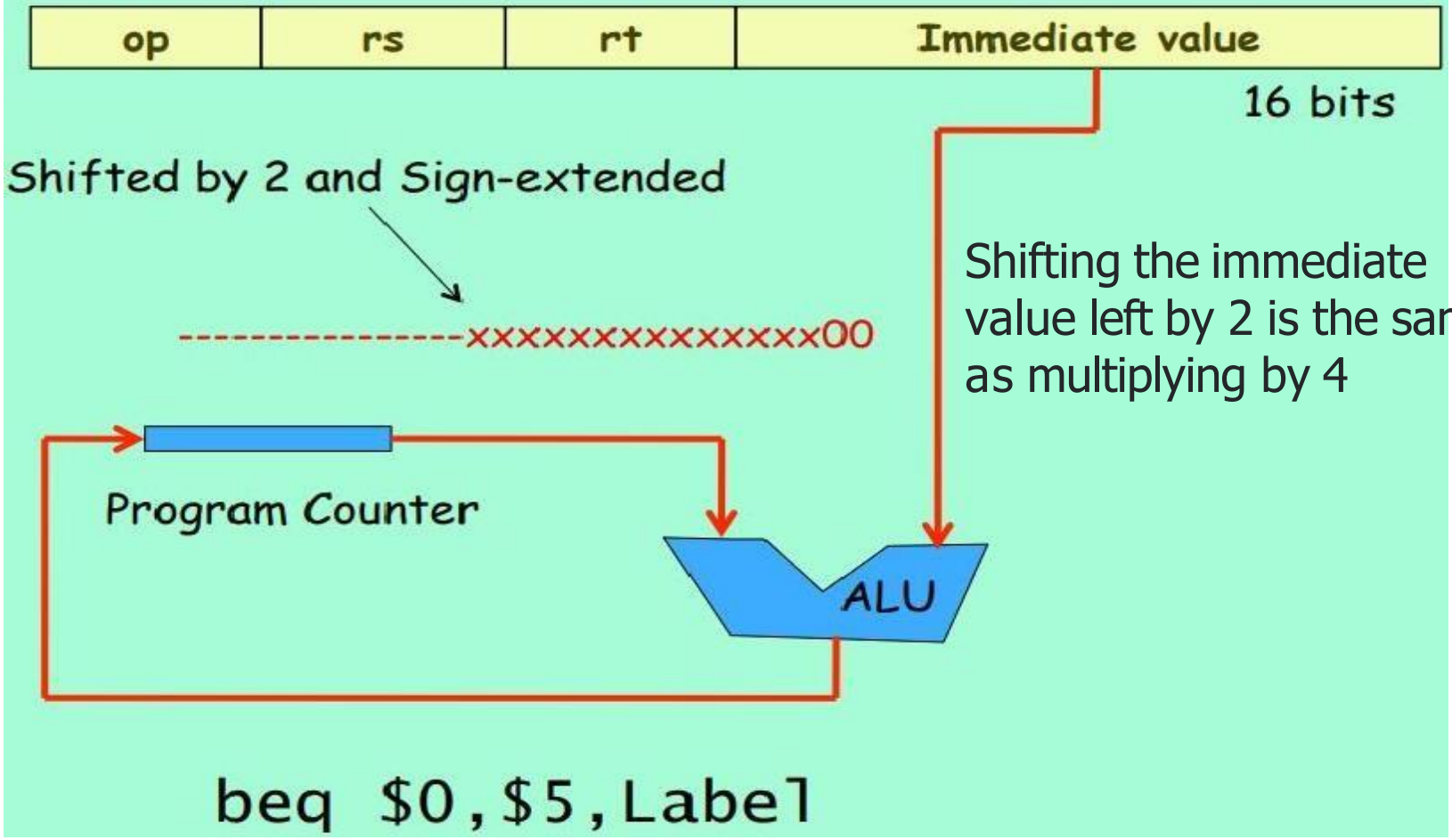


Addressing Modes



PC-relative addressing: the value in the immediate field is interpreted as an offset of the next instruction (PC+4 of current instruction)

Example: beq $0,$3,Label



beq $0,$5,Label

Detail of MIPS PC-Relative

address	instruction
40000008	addi $5, $5, 1
4000000C	beq $0, $5, label
40000010	addi $5, $5, 1
40000014	addi $5, $5, 1
40000018	label addi $5, $5, 1
4000001C	addi $5, $5, 1
40000020	etc…

Binary code to beq $0,$5, label is 0x10050002, which means 2 instructions from the next instruction.

PC =	0x4000000C
PC+4=	0x40000010
Add 4*2 =	0x00000008
Eff. Add. =	0x40000018

op	rs	rt	Immediate value
00010	00000	00101	0000000000000010

Data Transfer

Instruction	Example	Meaning	Comments
load word	`lw $1,100($2)`	$1=Memory[$2+100]	Copy from memory to register
store word	`sw $1,100($2)`	Memory[$2+100]=$1	Copy from register to memory
load upper immediate	`lui $1,100`	$1=100x2^16	Load constant into upper 16 bits. Lower 16 bits are set to zero.
load address	`la $1,label`	$1=Address of label	*Pseudo-instruction* (provided by assembler, not processor!) Loads computed address of label (not its contents) into register
load immediate	`li $1,100`	$1=100	*Pseudo-instruction* (provided by assembler, not processor!) Loads immediate value into register

Arithmetic Example

- C code:

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

Where f-j is in $s0-$s4

- Compiled MIPS code:

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

Example

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

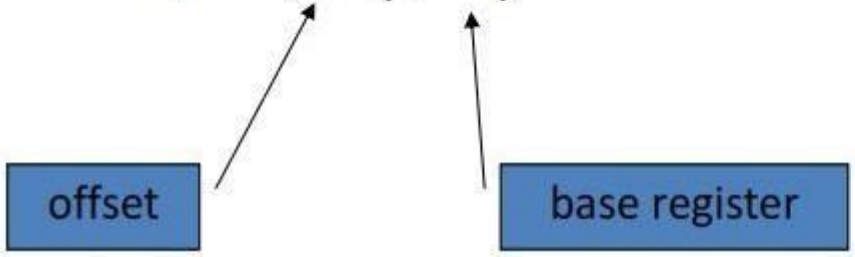
special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2$

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

Memory Operands

- *Load* instruction
  - lw refers to *load word*
  - lw registerName, offset (registerWithBaseAddress)
  - lw $t0 , 8 ($s3)



Memory Operand Example 1

- C code:

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

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

memory address= 4*offset + base address

4*8 + $s3

32 + $s3

32($s3)

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

Shift Operations

- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - sll $t2,$s0,3

0000 0000 0000 0000 0000 0000 0000 0001

0000 0000 0000 0000 0000 0000 0000 1000

  - sll by *i* bits multiplies by 2^i

Shift Operations

- sll $t2,$s0,3

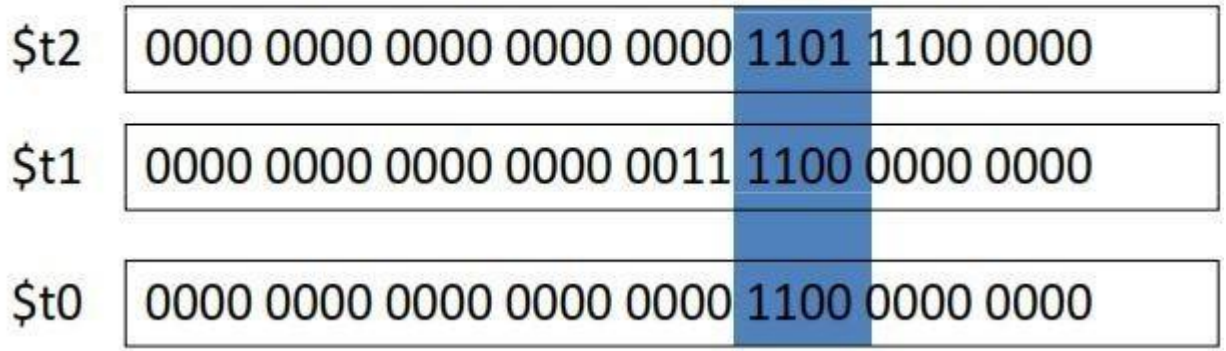
Source register $s0
Destination register $t2
Shift bit = 3

- shamt: how many positions to shift
- Similarly, …
  - Shift right logical

AND Operations

- Mask bits in a word
  - Select some bits, clear others to 0

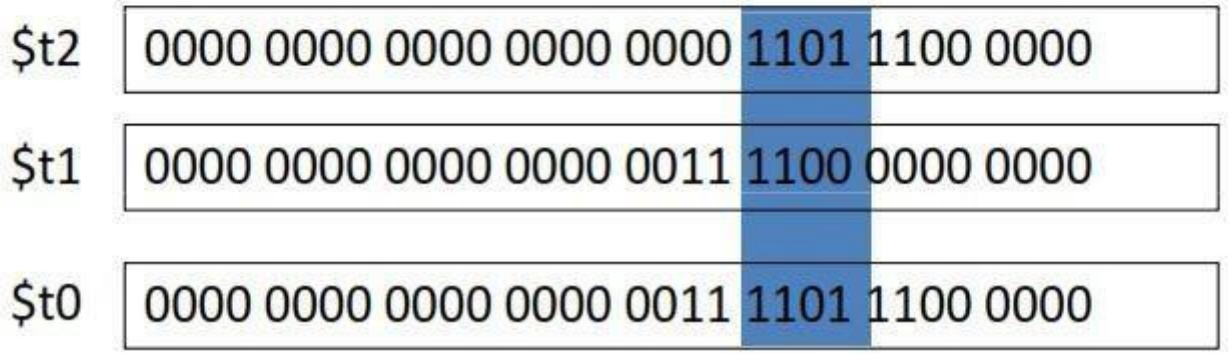
```
and $t0, $t1, $t2
```



OR Operations

- Include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```



NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

MIPS Conditional Branch Instructions

- MIPS **compare and branch** instructions:

  `beq Rs, Rt, label` if (`Rs == Rt`) branch to `label`

  `bne Rs, Rt, label` if (`Rs != Rt`) branch to `label`

- MIPS **compare to zero & branch** instructions:

  Compare to zero is used frequently and implemented efficiently

  `bltz Rs, label` if (`Rs < 0`) branch to `label`

  `bgtz Rs, label` if (`Rs > 0`) branch to `label`

  `blez Rs, label` if (`Rs <= 0`) branch to `label`

  `bgez Rs, label` if (`Rs >= 0`) branch to `label`

- `beqz` and `bnez` are defined as pseudo-instructions.

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Program to calculate Absolute value of difference between 2 input numbers: |A - B| (demonstrates if)

Program reads A from 4 bytes of memory starting at address 12345670_{16}.

Program reads B from 4 bytes of memory starting at address 12345674_{16}.

Program writes |A-B| to 4 bytes of memory starting at address 12345678_{16}.

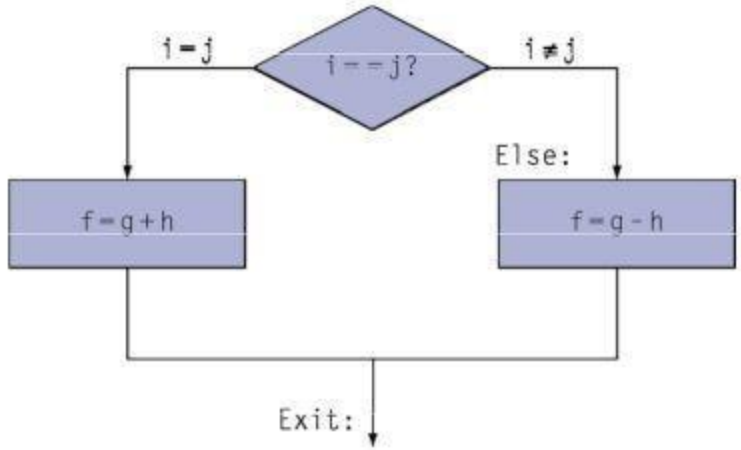
```
Assembler                          # Comment
lui   $10, 0x1234
ori   $10, $10, 0x5670    # put address of A into register $10
lw    $4,  0($10)         # read A from memory into register $4
lw    $5,  4($10)         # read B from memory into register $5 (A address+4)
sub   $12, $5, $4         # subtract A from B => B-A into register $12
bgez  $12,+1              # branch if B-A is positive to 'sw' instruction
sub   $12, $4, $5         # subtract B from A => A-B into register $12
sw    $12, 8($10)         # store register $12 value, |A-B|, into memory
```

Compiling If Statements

- C code:

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

  - f, g, … in $s0, $s1, …



- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

MIPS 'for loop' example

Let the variable *i* be stored in register $4, and variable *ap* in $6
Let 'array' of integers be stored at addresses 12345678_{16}-$1234569F_{16}$

Use $8 for temporary storage

ap = array

```
        lui     $6, 0x1234      : $6 <- 0x12340000
        ori     $6, $6, 0x5678  : $6 <- $6 | 0x5678  : $6 =0x12345678
i=0     add     $4, $0, $0      :  set  $4=0 : 0 → i          i<10
loop:   slti    $8, $4, 10      : set $8=1 if $4 < 10 otherwise 0
        beq     $8, $0, end     : if $8=0 ($4>=10) branch to end label
*ap=0   sw      $0, 0($6)       : store 32-bits of zero in $0 into array[i]
ap++    addui   $6, $6, 4       : ap++; add4 to $6 to point to array[i+1]
i++     addui   $4, $4, 1       : i++ ; increment loop variable

        beq     $0, $0, loop    : branch to label loop - always branches
end:
```

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