INSTRUCTIONS: LANGUAGE OF THE COMPUTER

Prof. Sebastian Eslava M.Sc. Ph.D.

jseslavag@unal.edu.co

Universidad Nacional de Colombia

Facultad de Ingeniería

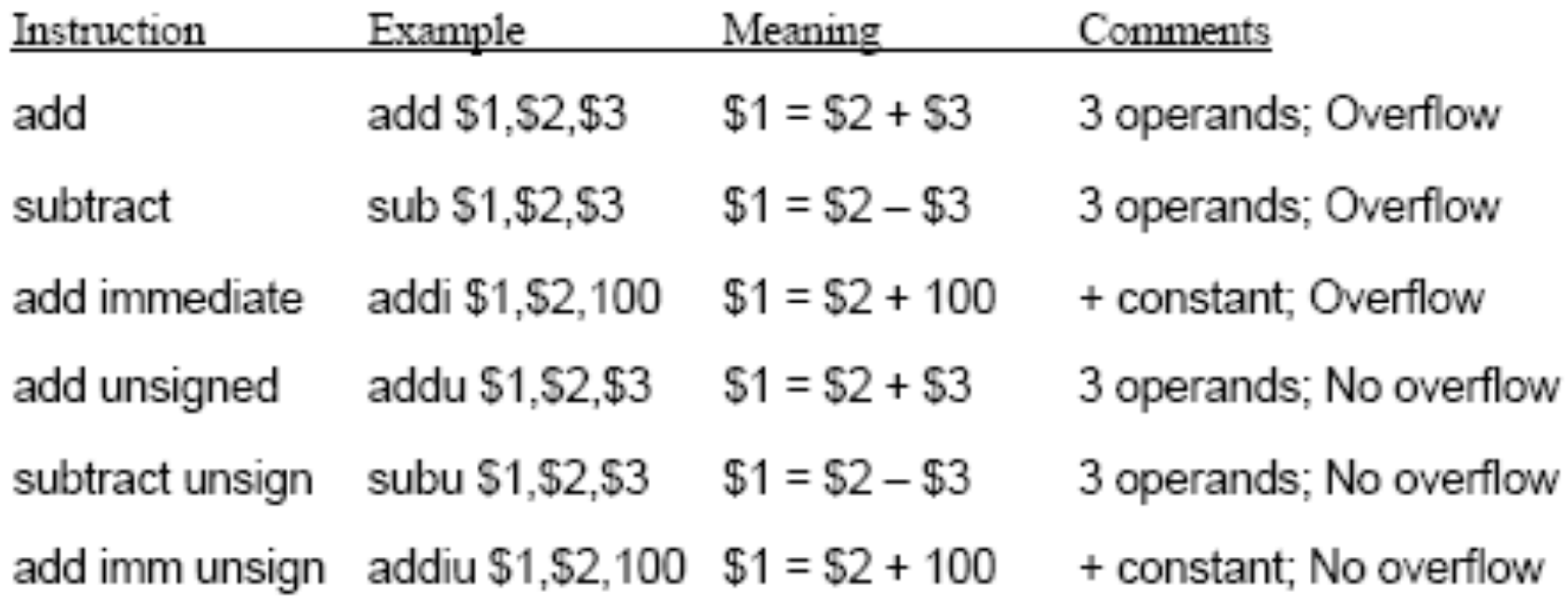
Departamento de Ingeniería Eléctrica y Electrónica

MIPS instructions and formats so far

- Arithmetic operations
  - add, sub and addi
- Data transfer
  - Load word: lw and Store word: sw
- Instruction formats
  - Each instruction starts with a 6-bit opcode, which is useful for the control logic to decode the instruction type.
  - R-format: used for arithmetic instructions
  - I-format: used for loads and stores but also for instructions with immediate operands.

MIPS simple arithmetic operations

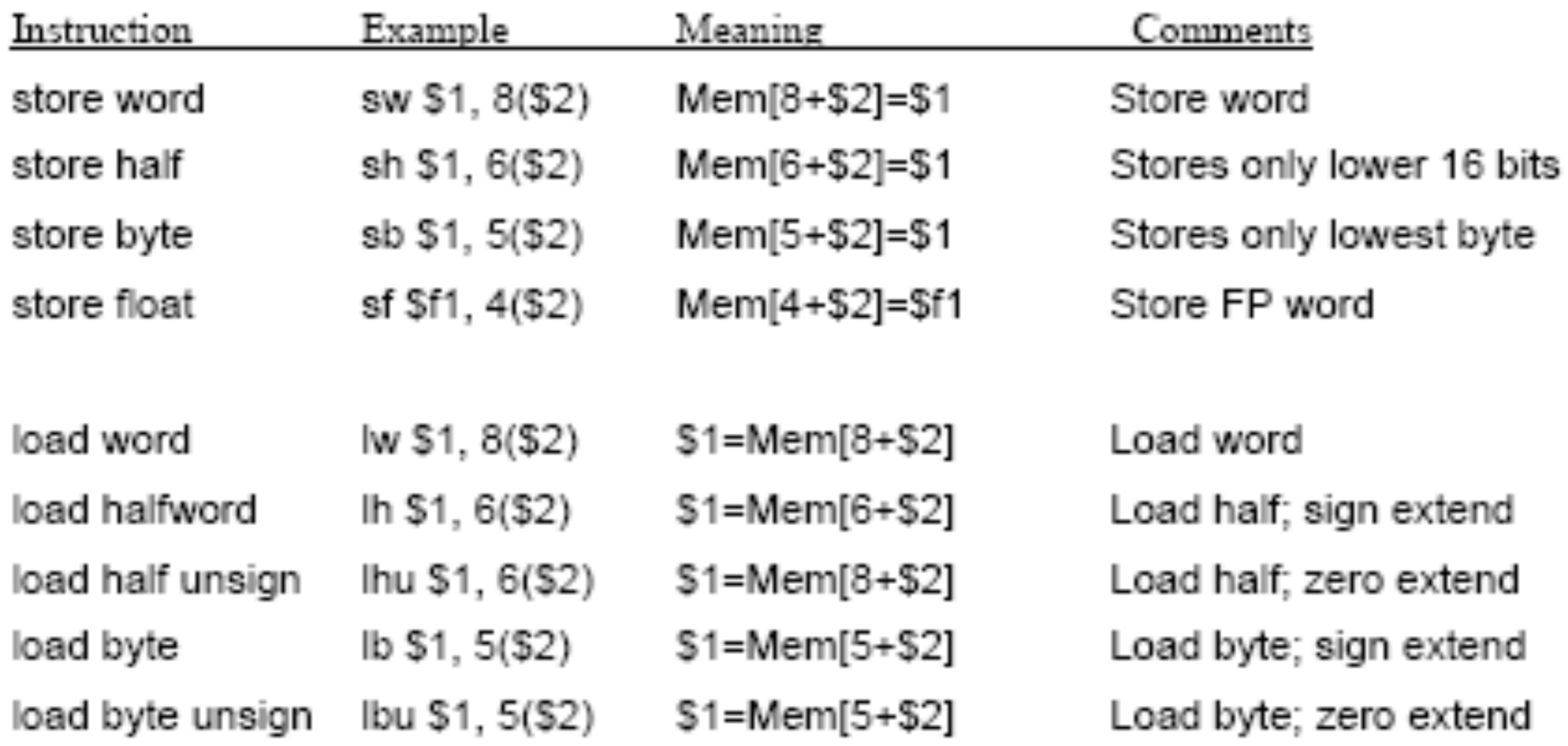
- Signed vs. unsigned numbers
  - Immediate filed is signed (+/- constants)
- Overflow?



Instruction	Example	Meaning	Comments
add	add \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; Overflow
subtract	sub \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; Overflow
add immediate	addi \$1,\$2,100	\$1 = \$2 + 100	+ constant; Overflow
add unsigned	addu \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; No overflow
subtract unsign	subu \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; No overflow
add imm unsign	addiu \$1,\$2,100	\$1 = \$2 + 100	+ constant; No overflow

MIPS Load/Store operations

- *Displacement-based* addressing mode
    - Base address + constant offset



Instruction	Example	Meaning	Comments
store word	sw $1, 8($2)	Mem[8+$2]=$1	Store word
store half	sh $1, 6($2)	Mem[6+$2]=$1	Stores only lower 16 bits
store byte	sb $1, 5($2)	Mem[5+$2]=$1	Stores only lowest byte
store float	sf $f1, 4($2)	Mem[4+$2]=$f1	Store FP word
load word	lw $1, 8($2)	$1=Mem[8+$2]	Load word
load halfword	lh $1, 6($2)	$1=Mem[6+$2]	Load half; sign extend
load half unsign	lhu $1, 6($2)	$1=Mem[8+$2]	Load half; zero extend
load byte	lb $1, 5($2)	$1=Mem[5+$2]	Load byte; sign extend
load byte unsign	lbu $1, 5($2)	$1=Mem[5+$2]	Load byte; zero extend

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

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

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

AND Operations

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

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

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

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

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

```
$t2   0000 0000 0000 0000 0000 1101 1100 0000
$t1   0000 0000 0000 0000 0011 1100 0000 0000
$t0   0000 0000 0000 0000 0011 1101 1100 0000
```

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

Control Flow Instructions

- Decision making instructions
  - Change control flow
- 2 types:
  - Conditional branches
    - beq: branch if equal
    - bne: branch if not equal
    - Slt: set on less than
  - Unconditional branches
    - j: jump
    - jr: jump register

Conditional Operations

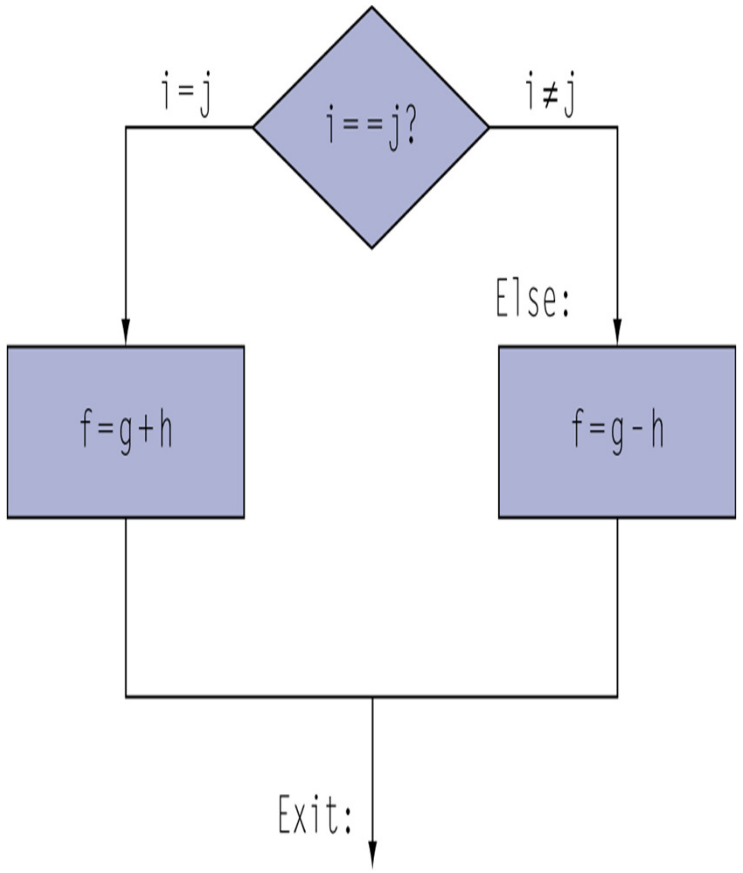
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

If-then-else Statements

- C code:

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

- Assume *f, g, h, i, j* in \$s0, \$s1, \$s2, \$s3 and \$s4, respectively.



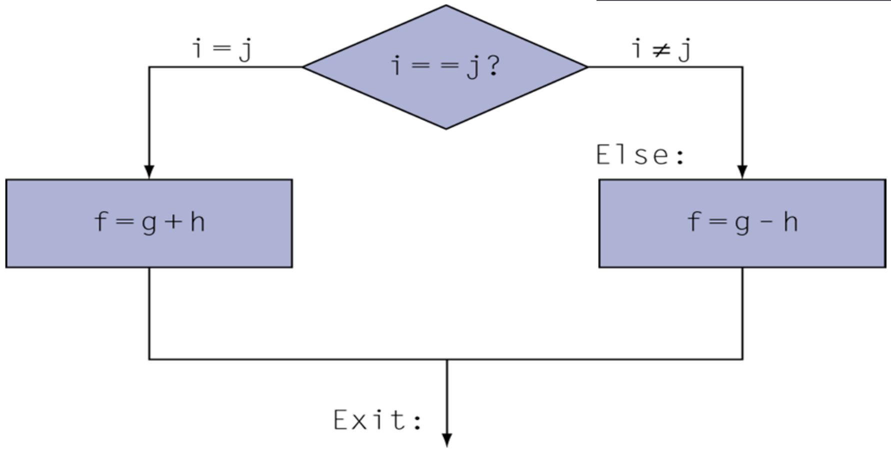
If-then-else Statements

- MIPS code:
  - Using inequality

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

Else: sub  $s0, $s1, $s2

Exit: …
```

- Using equality

```
      beq  $s3, $s4, Then
      sub  $s0, $s1, $s2
      j    Exit

Then: add  $s0, $s1, $s2

Exit: …
```



Compiling Loop Statements

- C code:

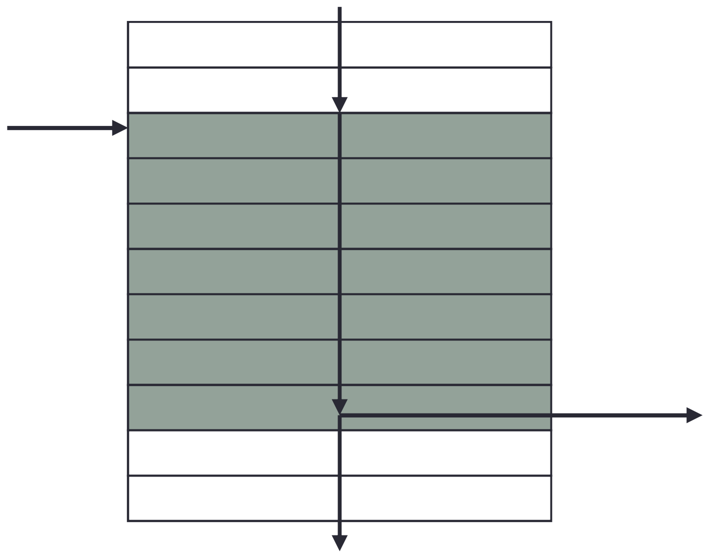
```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - $-1 < +1 \Rightarrow$ \$t0 = 1
  - `sltu $t0, $s0, $s1  # unsigned`
    - $+4,294,967,295 > +1 \Rightarrow$ \$t0 = 0

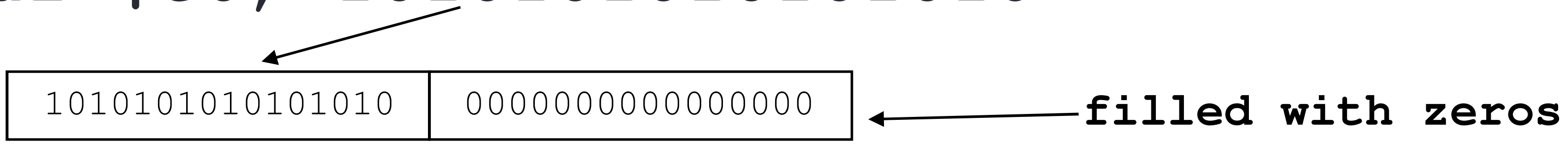
MIPS Addressing

- Recall: All MIPS instructions are 32-bit long. How to specify a 32-bit constant operands or 32-bit memory address?

32-bit Immediate Operands

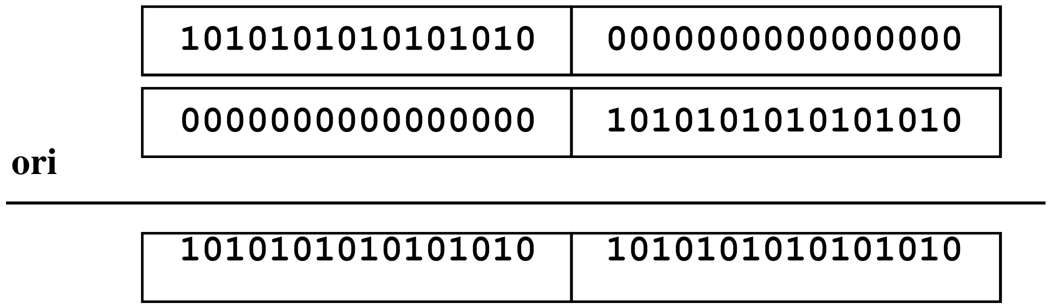
- How to load a 32 bit constant into a register?
  → Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Addresses in Branches and Jumps

- Instructions:

  bne \$t4,\$t5,Label # Next instruction is at Label if \$t4<>\$t5

  beq \$t4,\$t5,Label # Next instruction is at Label if \$t4=\$t5

  j Label # Next instruction is at Label

- Formats:

I	`op`	`rs`	`rt`	`16 bit address`

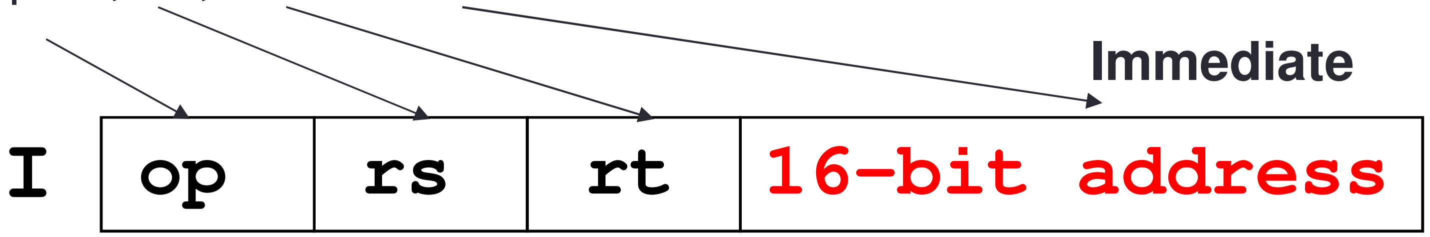
J	`op`	`26 bit address`

- In both cases, address field is not 32 bits. How to obtain a 32-bit address?

Addresses in Branches

- Consider the conditional branch instructions:

bne $t4,$t5,Label # 16-bit address
beq $t4,$t5,Label # 16-bit address



- Could specify a register (like lw and sw) and add it to address
  - Implicit register: use Instruction Address Register (PC = program counter)→ most branches are local (principle of locality)
- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

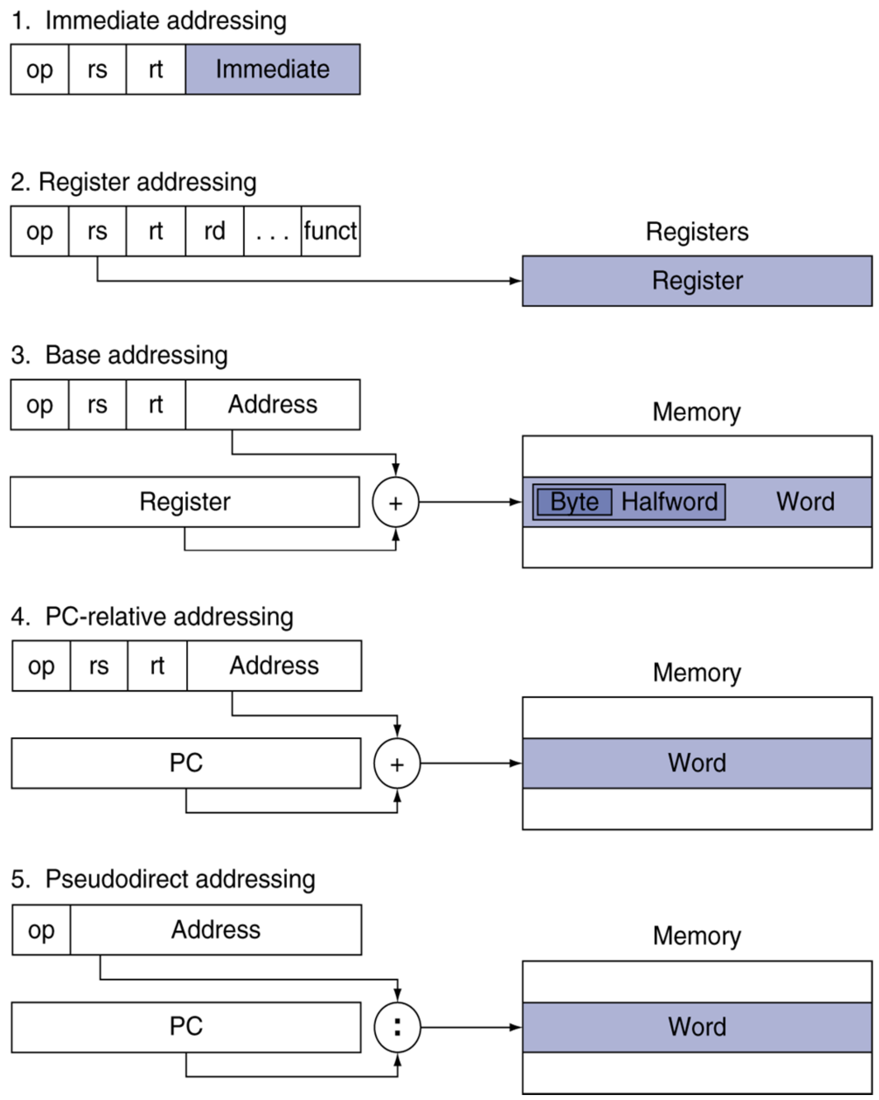
`Loop:`	`sll  $t1, $s3, 2`	80000	0	0	19	9	4	0
	`add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
	`lw   $t0, 0($t1)`	80008	35	9	8	0		
	`bne  $t0, $s5, Exit`	80012	5	8	21	2		
	`addi $s3, $s3, 1`	80016	8	19	19	1		
	`j    Loop`	80020	2	20000				
`Exit:`	`…`	80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
            beq $s0,$s1, L1
                   ↓
            bne $s0,$s1, L2
            j L1
    L2:     …
```

Addressing Modes



- Immediate: operand is a constant *within the instruction*
- Register: operand is a register
- Base or Displacement: operand is at memory location that is constant offset of a base address
- PC-relative: address is sum of PC and some constant
- Pseudodirect: address is low bits (26) concatenated with upper bits of PC

Addressing Modes

- Some instructions use more than one addressing mode (different operands are addressed differently)
- MIPS uses PC-relative addressing for all conditional branches because destination is likely to be close to PC.
- Jump-and-link uses pseudodirect addressing. This allows instructions to jump "far away".

So far:

• Instruction	Meaning
`add $s1,$s2,$s3`	*$s1 = $s2 + $s3*
`sub $s1,$s2,$s3`	*$s1 = $s2 – $s3*
`lw $s1,100($s2)`	*$s1 = Memory[$s2+100]*
`sw $s1,100($s2)`	*Memory[$s2+100] = $s1*
`bne $s4,$s5,L`	*Next instr. is at Label if $s4 <> $s5*
`beq $s4,$s5,L`	*Next instr. is at Label if $s4 = $s5*
`j Label`	*Next instr. is at Label*

• Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				