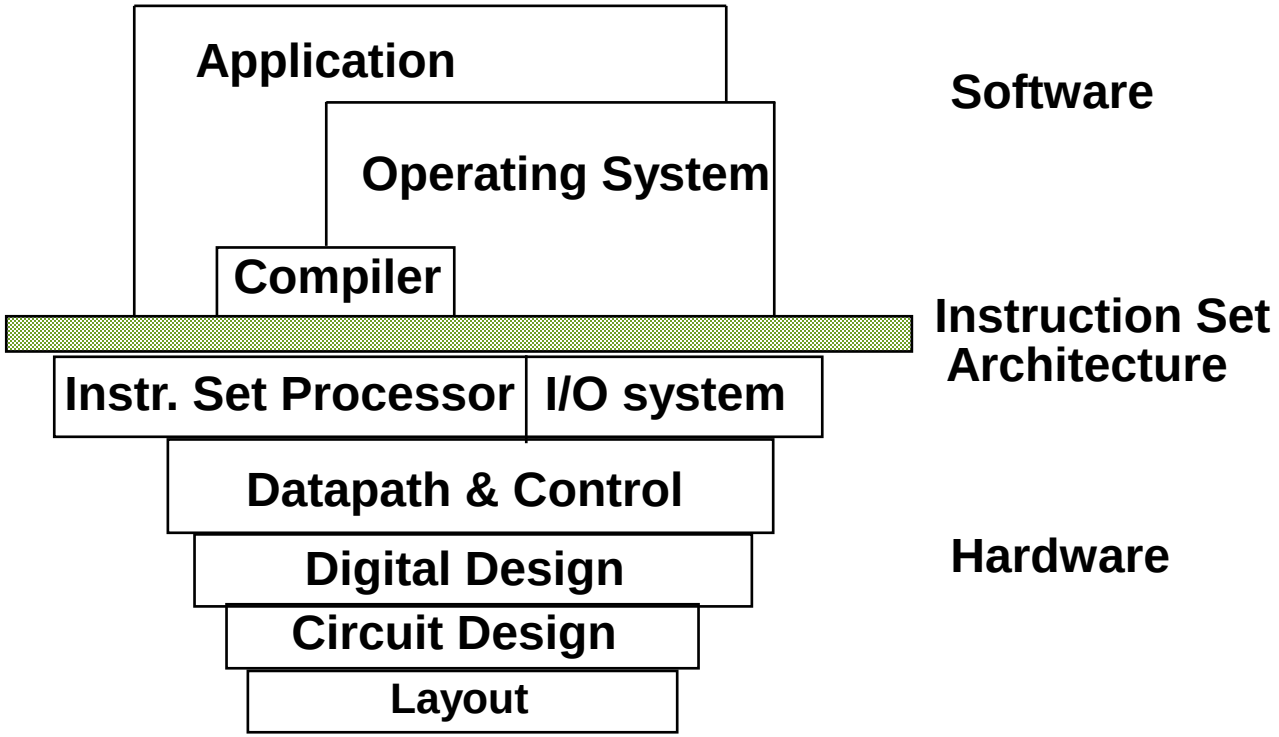
# CSE 331
# Computer Organization
# Lecture 2
# Instruction Set Architecture

# Instruction Set Architecture (ISA)



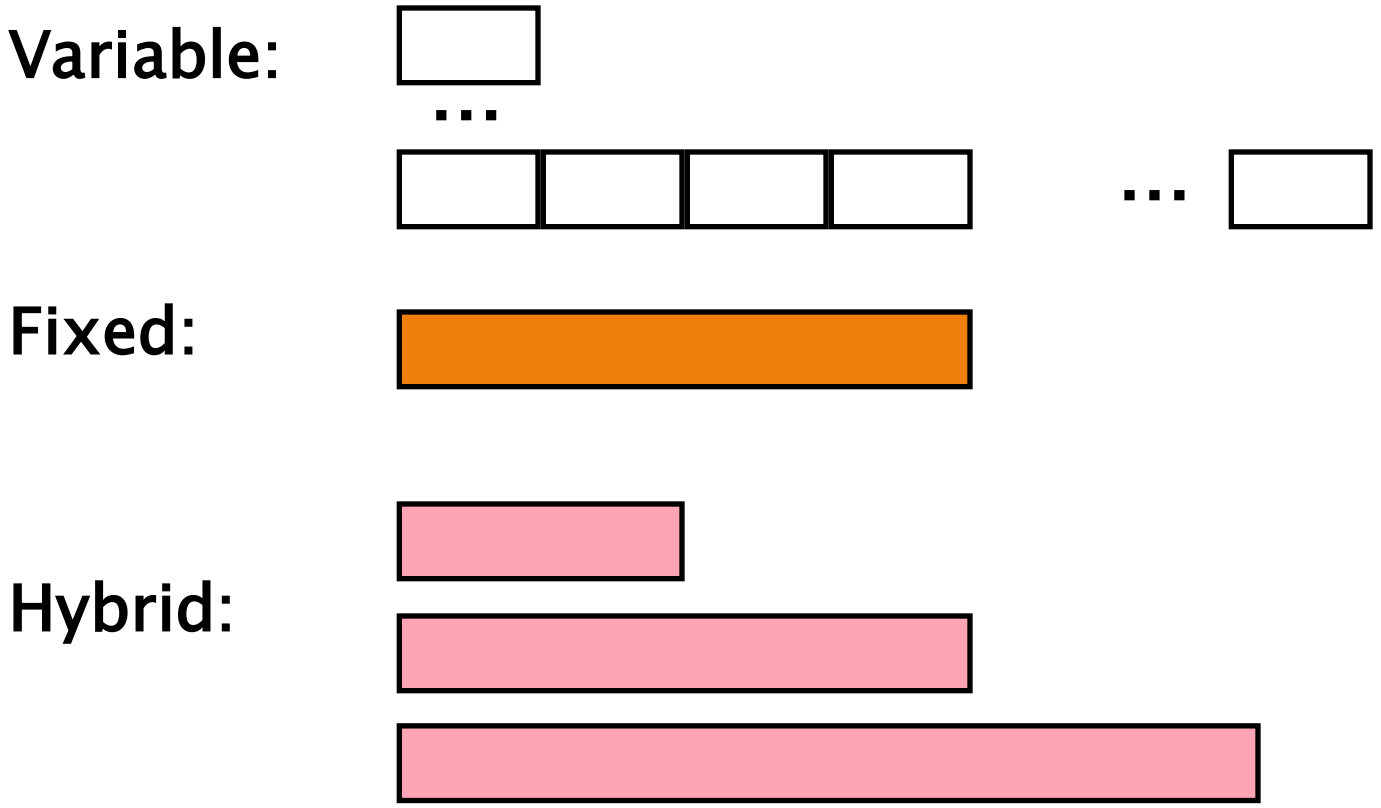
# Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

# The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

# Generic Examples of Instruction Format Widths



# Basic ISA Classes

- **Memory to Memory Machines**
  - We need storage for temporaries
  - Memory is slow
  - Memory is big (lots of address bits)

- **Architectural Registers**
  - Registers can hold temporary variables
  - Registers are faster than memory
  - Memory traffic is reduced, so program is speed up (since registers are faster than memory)
  - Code density improves (since register named with fewer bits than memory location)

# Basic ISA Classes (cont'd)

- **Stack (not a register file but an operand stack)**
  - 0 address  add  tos = tos + next
- **Accumulator (1 register):**
  - 1 address  add A  acc = acc + mem[A]
- **General Purpose Register File (Register-Memory):**
  - 2 address  add A B  EA(A) = EA(A) + EA(B)
  - 3 address  add A B C  EA(A) = EA(B) + EA(C)
- **General Purpose Register File (Load/Store):**
  - 3 address  add Ra Rb Rc  Ra = Rb + Rc
  - load Ra Rb  Ra = mem[Rb]
  - store Ra Rb  mem[Rb] = Ra
- **Comparison:**
  - Bytes per instruction? Number of Instructions? Cycles per instruction?

# Comparing Number of Instructions

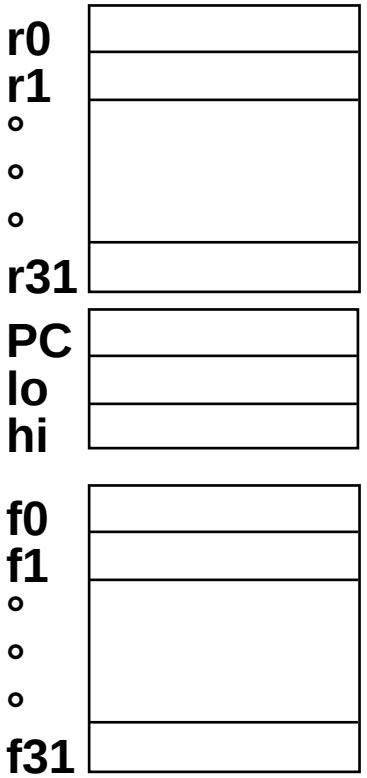
- Code sequence for **C** = **A** + **B** for four classes of instruction sets:

| Stack | Accumulator | Register (register-memory) | Register (load-store) |
|---|---|---|---|
| Push A | Load A | Load R1, A | Load R1, A |
| Push B | Add B | Add R1, B | Load R2 ,B |
| Add | Store C | Store C, R1 | Add R3, R1, R2 |
| Pop C | | | Store C, R3 |

MIPS is one of these: this is what we'll be learning

# MIPS Architecture – Registers

- The MIPS architecture is considered to be a typical RISC architecture.
  - Simplified instruction set => easier to study
  - Most new machines are RISC
- Programmable storage
  - 31 x 32-bit GPRs (r0 = 0)
  - special purpose – HI, LO, PC
  - 32 x 32-bit FP regs
  - 2^32 x bytes of memory



# Register Names in MIPS Assembly Language

- With MIPS, there is a convention for mapping register names into general purpose register numbers.

| Name | Register Address | Usage | Preserved on call? |
|---|---|---|---|
| $zero | 0 | the constant value 0 | n.a. |
| $v0-$v1 | 2-3 | values for result and expression evaluation | no |
| $a0-$a3 | 4-7 | arguments | no |
| $t0-$t7 | 8-15 | temporaries | no |
| $s0-$s7 | 16-23 | saved | yes |
| $t8-$t9 | 24-25 | more temporaries | no |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return address | yes |

# MIPS Arithmetic

- Arithmetic instructions have 3 operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

# MIPS Arithmetic

- *Design Principle:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

- Of course this complicates some things...

C code:

```
A = B + C + D;
E = F - A;
```

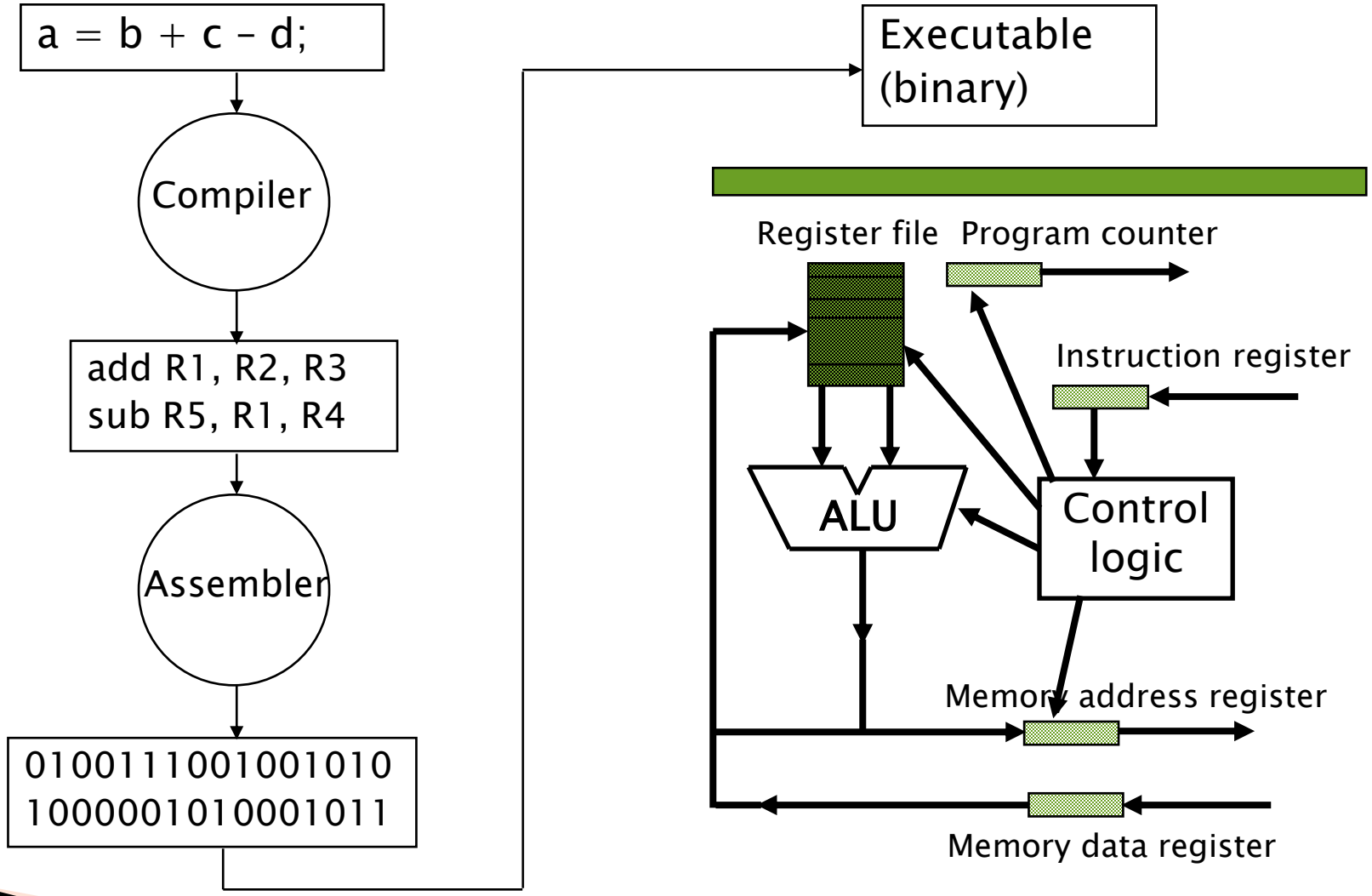
MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operands must be registers, only 32 registers provided

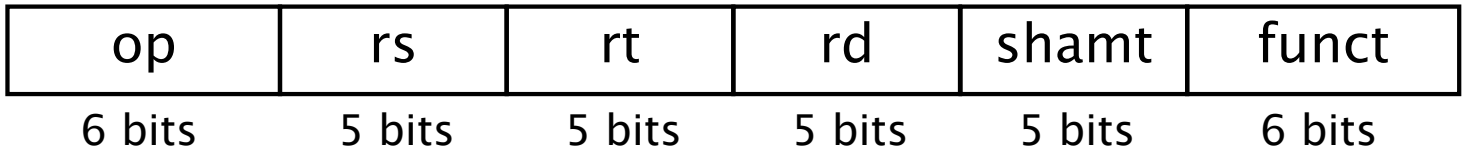
- *Design Principle:* Smaller is faster.

# Big Picture


a = b + c – d;
Compiler
add R1, R2, R3
sub R5, R1, R4
Assembler
0100111001001010
1000001010001011
Executable
(binary)
Register file
Program counter
Instruction register
ALU
Control
logic
Memory address register
Memory data register

# MIPS R-Type Instructions

```
instr $rd, $rs, $rt
```




- 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-Type 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 |

$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 0232\ 4020_{16}$

# Hexadecimal

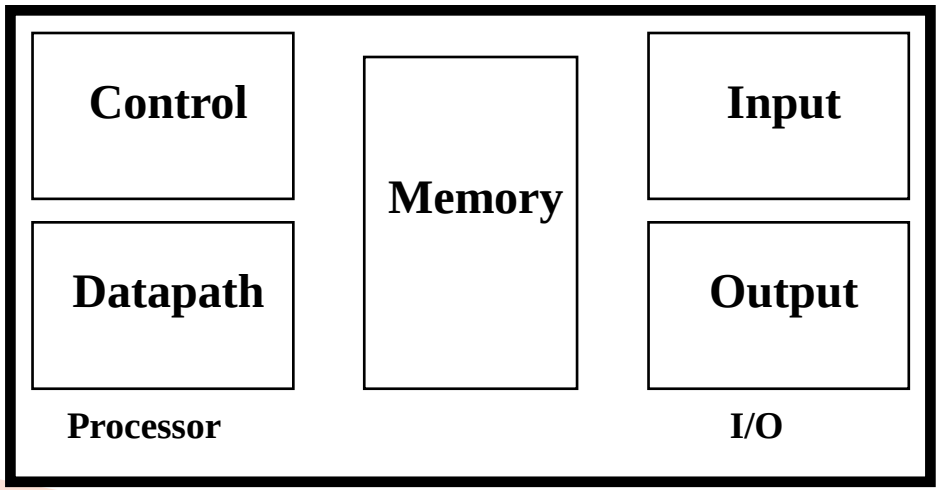
- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

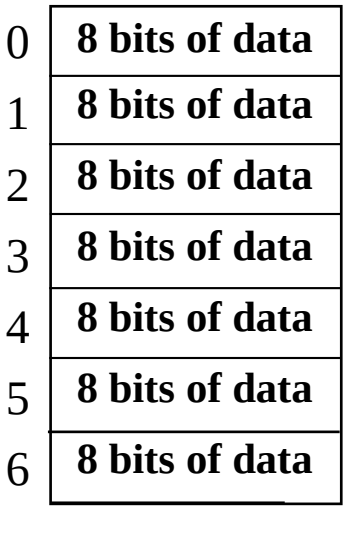
# Registers vs. Memory

- Registers are faster to access than memory
- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler associates variables with registers
- What about programs with lots of variables ??



# 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.

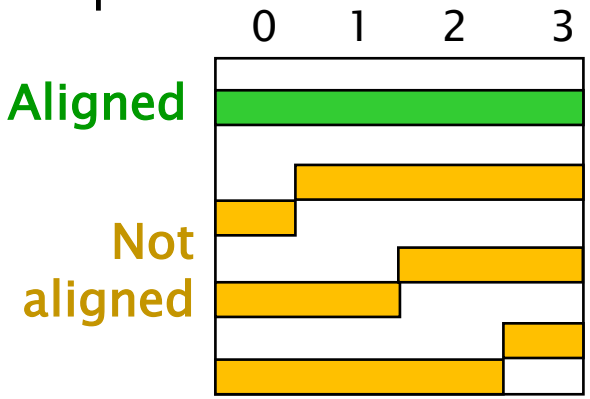


# Memory Organization

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

| | |
|---|---|
| 0 | 32 bits of data |
| 4 | 32 bits of data |
| 8 | 32 bits of data |
| 12 | 32 bits of data |

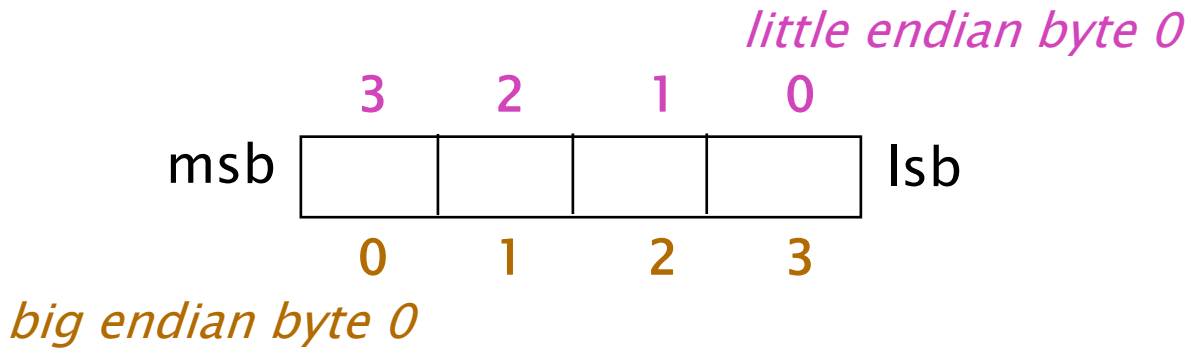
- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32} - 4$
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?




**Alignment: require that objects fall on address that is multiple of their size.**

# Addressing Objects: Endianess and Alignment

- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha



# Memory Operand Example 1

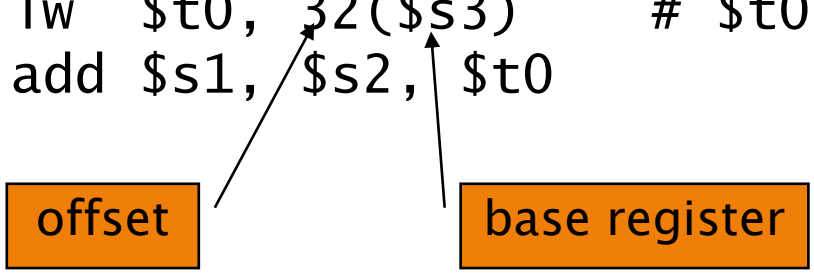
- C code:

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

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

- Compiled MIPS code:
  - Index 8 requires offset of 32?
    - 4 bytes per word

```
lw  $t0, 32($s3)     # $t0 <= Mem[$s3 + 32]
add $s1, $s2, $t0
```



# Memory Operand Example 2

- Arrays are often stored in memory – why?
- Replace the C code for
  A[11] = A[10] + b
  by equivalent MIPS instructions.
- Assume `b` is in register `$s5`, the starting address for array `A` is in `$s6`, using and 32-bit integer data.

| Instruction | Comment |
|---|---|
| `lw $t3, 40($s6)` | `$t3 = A[10]` |
| `add $t4, $t3, $s5` | `$t4 = A[10] + b` |
| `sw $t4, 44($s6)` | `A[11] = $t4` |

- Why are array indices multiplied by 4?

# MIPS I-Type Instructions


- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - immed: immediate value

```
lw  $t0, 32($s3)   # $t0 <= Mem[$s3 + 32]
```

# So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

| Instruction | Meaning |
| --- | --- |
| `add $s1, $s2, $s3` | `$s1 = $s2 + $s3` |
| `sub $s1, $s2, $s3` | `$s1 = $s2 - $s3` |
| `lw  $s1, 100($s2)` | `$s1 = mem[$s2+100]` |
| `sw  $s1, 100($s2)` | `mem[$s2+100] = $s1` |

# Constants, i.e. Immediates

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- MIPS Instructions:

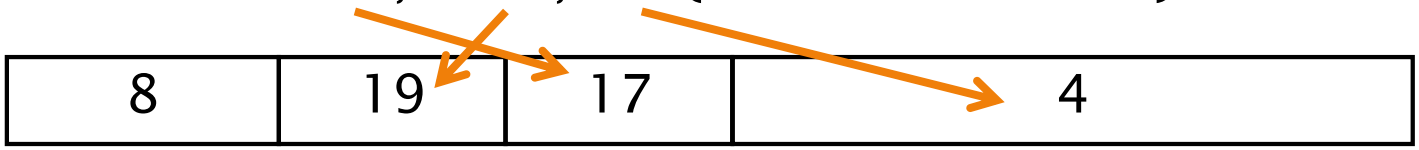
```
addi $29, $29, 4
slti $8,  $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

- How do we make this work?

# Immediate (I-Type) Arithmetic


- Constant data specified in an instruction

  `addi $s1, $s3, 4 ($s1 <= $s3 + 4)`




- No subtract immediate instruction
  - Just use a negative constant

    `addi $s2, $s1, -1`

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

# MIPS Arithmetic Instructions

| *Instruction* | *Example* | *Meaning* | *Comments* |
|---|---|---|---|
| add | add $1,$2,$3 | $1 = $2 + $3 | 3 operands; exception possible |
| subtract | sub $1,$2,$3 | $1 = $2 – $3 | 3 operands; exception possible |
| add immediate | addi $1,$2,100 | $1 = $2 + 100 | + constant; exception possible |
| add unsigned | addu $1,$2,$3 | $1 = $2 + $3 | 3 operands; no exceptions |
| subtract unsigned | subu $1,$2,$3 | $1 = $2 – $3 | 3 operands; no exceptions |
| add imm. unsign. | addiu $1,$2,100 | $1 = $2 + 100 | + constant; no exceptions |
| multiply | mult $2,$3 | Hi, Lo = $2 x $3 | 64-bit signed product |
| multiply unsigned | multu $2,$3 | Hi, Lo = $2 x $3 | 64-bit unsigned product |
| divide | div $2,$3 | Lo = $2 ÷ $3, Hi = $2 mod $3 | Lo = quotient, Hi = remainder |
| divide unsigned | divu $2,$3 | Lo = $2 ÷ $3, Hi = $2 mod $3 | Unsigned quotient & remainder |
| Move from Hi | mfhi $1 | $1 = Hi | Used to get copy of Hi |
| Move from Lo | mflo $1 | $1 = Lo | Used to get copy of Lo |

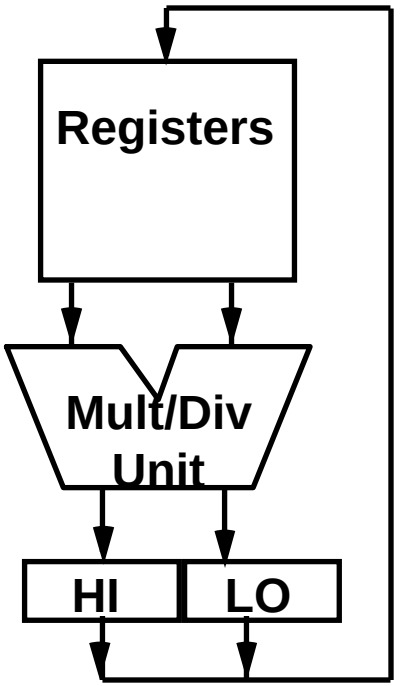
Note: Move from Hi and Move from Lo are really data transfers

# Branch Instruction Design

- Why not `blt`, bge, etc?
- Hardware for $<$, $\geq$, … slower than $=$, $\neq$
  - 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

# Multiply / Divide

- Perform multiply, divide
  - mult rs, rt
  - multu rs, rt
  - div rs, rt
  - divu rs, rt
- Move result from multiply, divide
  - mfhi rd
  - mflo rd



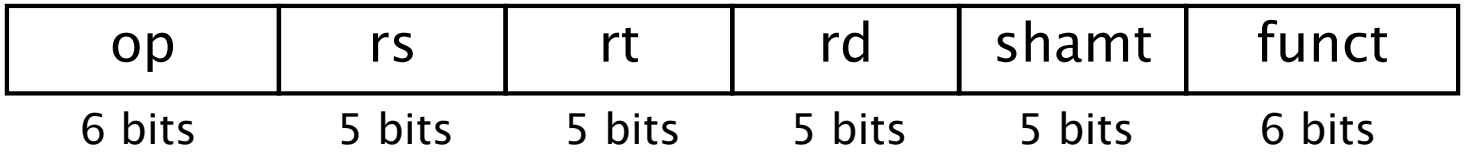
# Logical Operations

- Instructions for bitwise manipulation

| Operation | C | MIPS |
|---|---|---|
| Shift left | << | `sll` |
| Shift right | >> | `srl` |
| Bitwise AND | & | `and, andi` |
| Bitwise OR | \| | `or, ori` |

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

# Shift Operations




- shamt: how many positions to shift

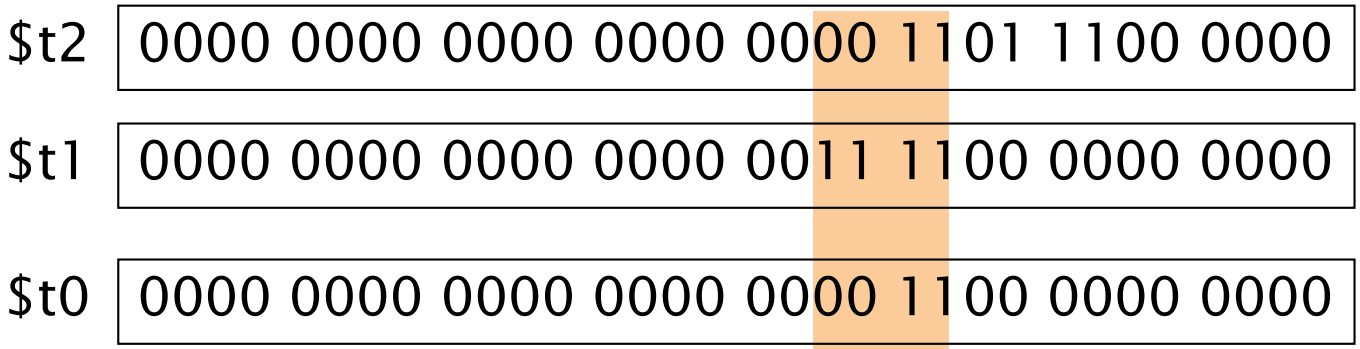
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by $2^i$
  - Ex: **`sll $t0, $t0, 5`**

- 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

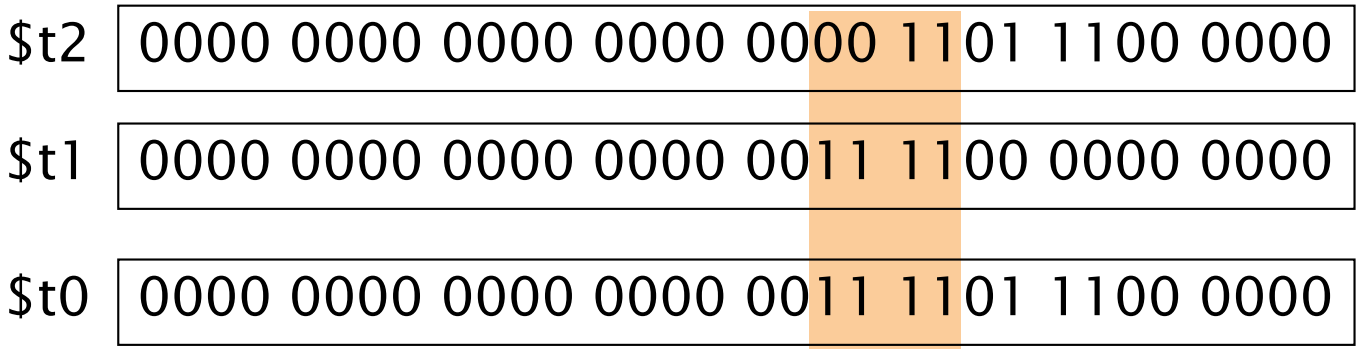
```
and $t0, $t1, $t2
```



# OR Operations

- Useful to 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

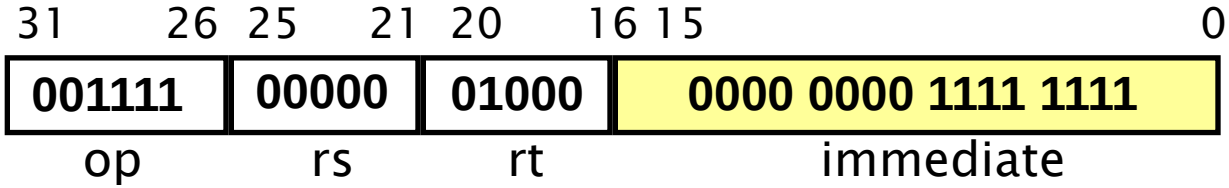
| | |
|---|---|
| $t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| $t0 | 1111 1111 1111 1111 1100 0011 1111 1111 |

# MIPS logical instructions

| *Instruction* | *Example* | *Meaning* | *Comment* |
|---|---|---|---|
| **and** | **and $1,$2,$3** | **$1 = $2 & $3** | **3 reg. operands; Logical AND** |
| **or** | **or $1,$2,$3** | **$1 = $2 \| $3** | **3 reg. operands; Logical OR** |
| **xor** | **xor $1,$2,$3** | **$1 = $2 ⊕ $3** | **3 reg. operands; Logical XOR** |
| **nor** | **nor $1,$2,$3** | **$1 = ~($2 \| $3)** | **3 reg. operands; Logical NOR** |
| **and immediate** | **andi $1,$2,10** | **$1 = $2 & 10** | **Logical AND reg, constant** |
| **or immediate** | **ori $1,$2,10** | **$1 = $2 \| 10** | **Logical OR reg, constant** |
| **xor immediate** | **xori $1, $2,10** | **$1 = $2 ⊕ 10** | **Logical XOR reg, constant** |
| **shift left logical** | **sll $1,$2,10** | **$1 = $2 << 10** | **Shift left by constant** |
| **shift right logical** | **srl $1,$2,10** | **$1 = $2 >> 10** | **Shift right by constant** |
| **shift right arit.** | **sra $1,$2,10** | **$1 = $2 >> 10** | **Shift right (sign extend)** |
| **shift left logical** | **sllv $1,$2,$3** | **$1 = $2 << $3** | **Shift left by variable** |
| **shift right logical** | **srlv $1,$2, $3** | **$1 = $2 >> $3** | **Shift right by variable** |
| **shift right arit.** | **srav $1,$2, $3** | **$1 = $2 >> $3** | **Shift right arith. by variable** |

# Load Upper Immediate

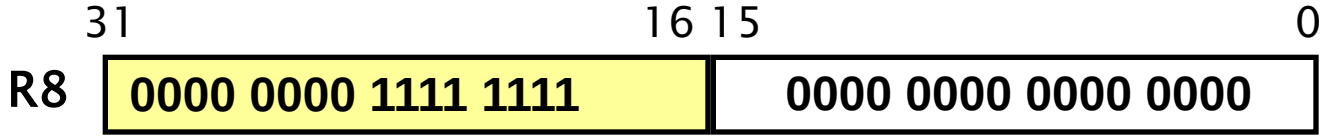
- Example: lui R8, 255




- Transfers the immediate field into the register's top 16 bits and fills the register's lower 16 bits with zeros

R8[31:16] <-- 255

R8[15:0] <-- 0

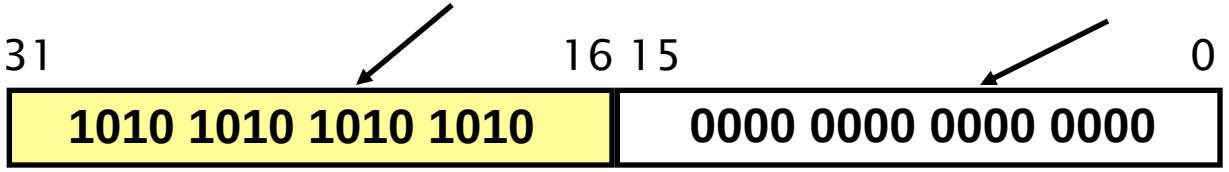


# Large Constants

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

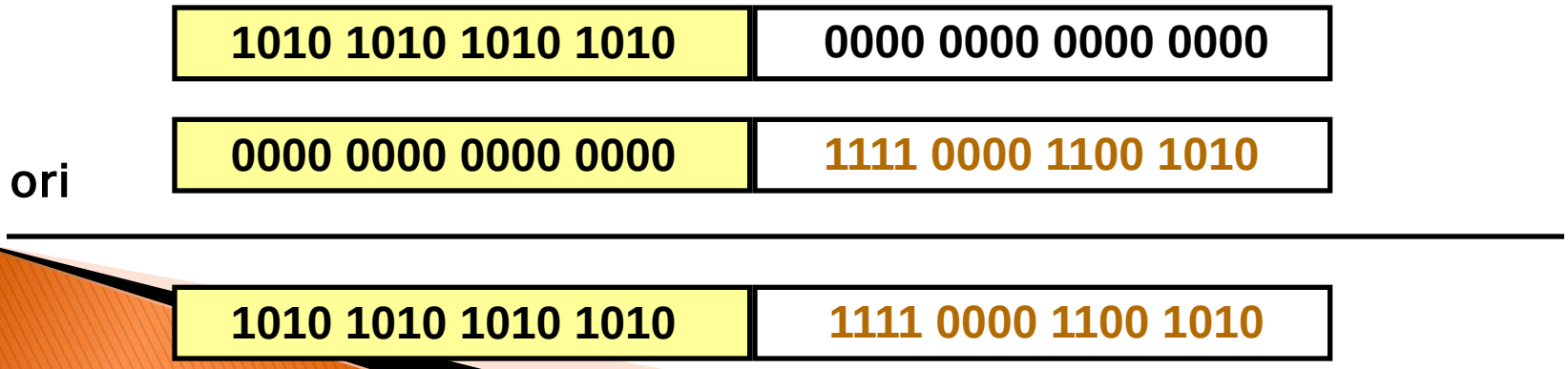
```
lui $t0, 1010101010101010
```

filled with zero




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

```
ori $t0, $t0, 1111000011001010
```



# Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

# Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

# 2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: $-2^{n-1}$ to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_2$
    $= -4_{10}$
- Using 32 bits
  - -2,147,483,648 to +2,147,483,647

# 2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - -1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

# Sign Extension

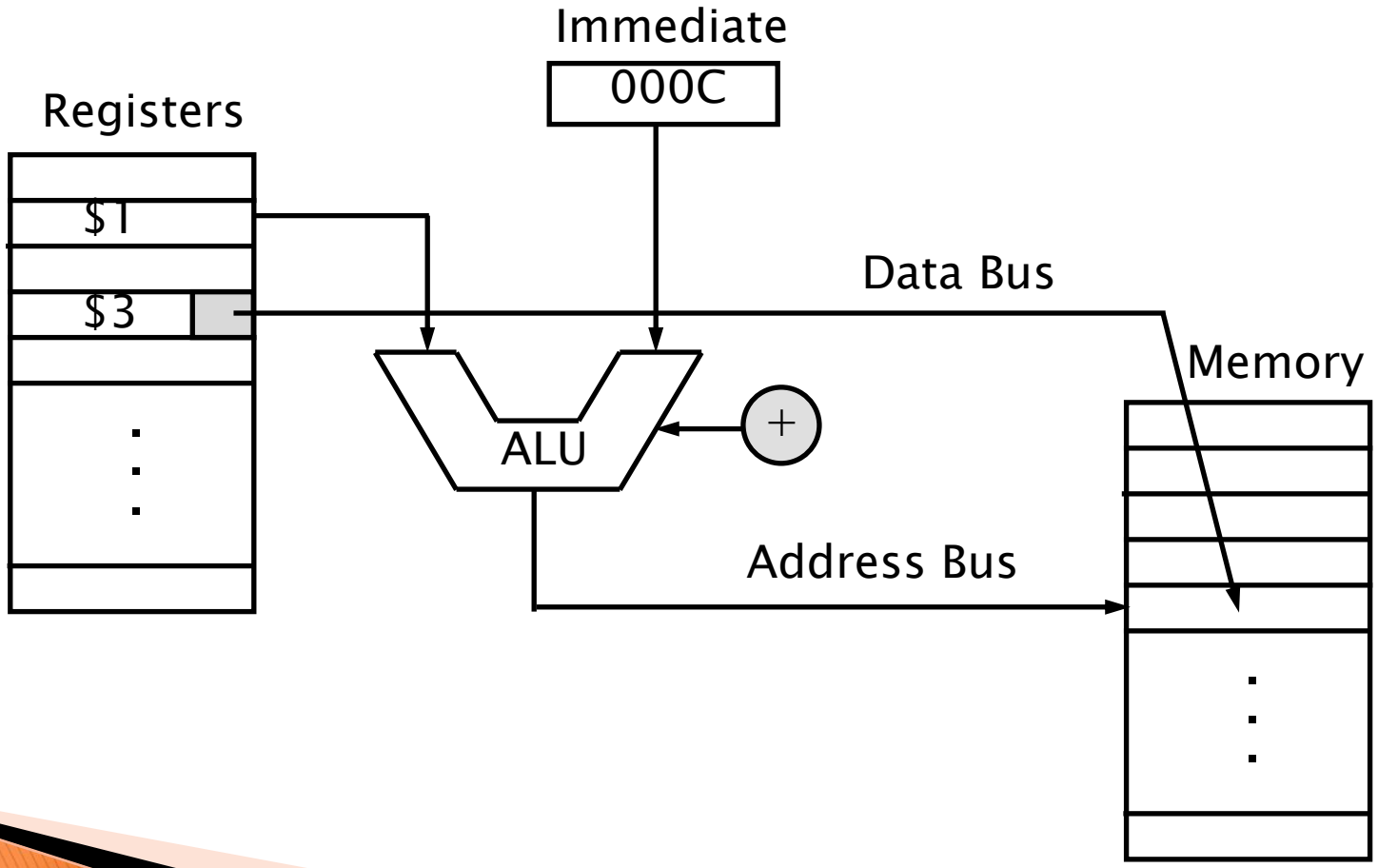
- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - addi: extend immediate value
  - lb, lh: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - -2: 1111 1110 => 1111 1111 1111 1110

# MIPS Data Transfer Instructions

| *Instruction* | *Comment* |
|---|---|
| **sw $3, 500($4)** | **Store word** |
| **sh $3, 502($2)** | **Store half** |
| **sb $2, 41($3)** | **Store byte** |
| | |
| **lw $1, 30($2)** | **Load word** |
| **lh $1, 40($3)** | **Load halfword** |
| **lhu $1, 40($3)** | **Load halfword unsigned** |
| **lb $1, 40($3)** | **Load byte** |
| **lbu $1, 40($3)** | **Load byte unsigned** |
| | |
| **lui $1, 40** | **Load Upper Immediate (16 bits shifted left by 16)** |

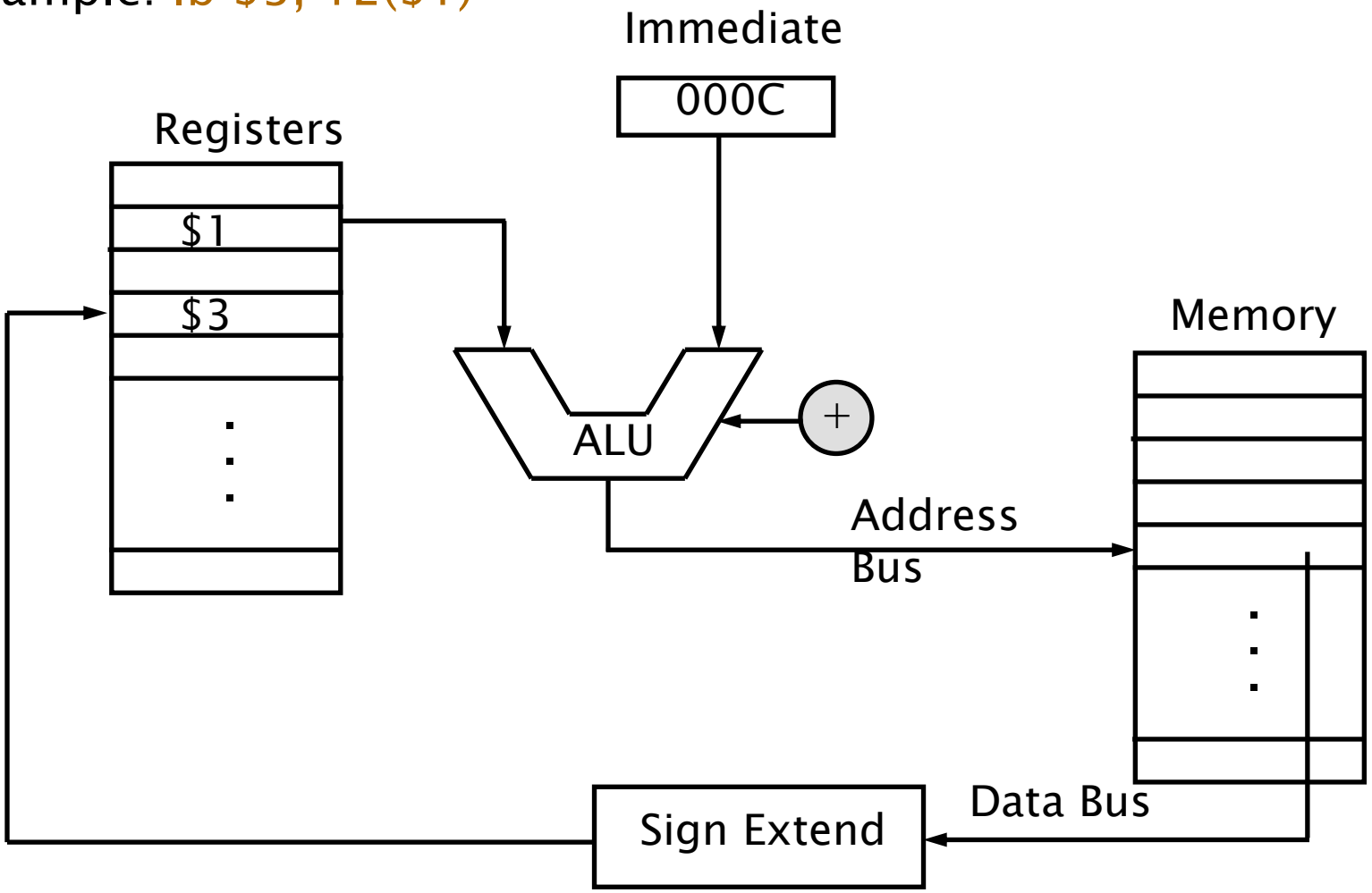
# Store Byte (sb) instruction

- Example: sb $3, 12($1)



# Load Byte (lb) instruction

- Example: lb $3, 12($1)



# 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`

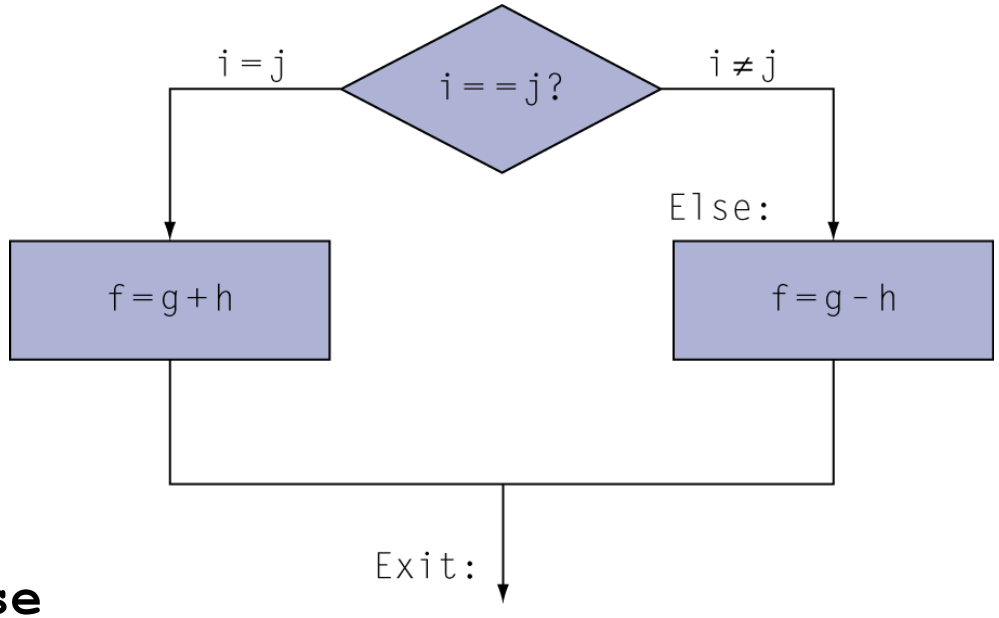
| Instruction | Comment |
| --- | --- |
| `bne $t0, $t1, Label` | `if (t0 != t1) goto Label` |
| `beq $t0, $t1, Label` | `if (t0 == t1) goto Label` |

# 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

# 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: …
```

# Branch Instructions

- Branch instructions end up the way we implement C-style loops

```
for ( j = 0; j < 10; j++) {
   a = a + j;
}
```

assume `s0 == j;  s1 == a;  t0 == temp;`

| Instruction | | Comment |
|---|---|---|
| | `addi $s0, $zero, 0` | `j = 0 + 0` |
| | `addi $t0, $zero, 10` | `temp = 0 + 10` |
| `Loop:` | `beq $s0, $t0, Exit` | `if (j == temp) goto Exit` |
| | `add $s1, $s1, $s0` | `a = a + j` |
| | `addi $s0, $s0, 1` | `j = j + 1` |
| | `j Loop` | `goto Loop` |
| `Exit:` | … | `exit from loop and continue` |

# 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
```

# 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

# Signed vs. Unsigned Comparison Example

R1 = 0…00 0000 0000 0000 $0001_2$ = $1_{10}$
R2 = 0…00 0000 0000 0000 $0010_2$ = $2_{10}$
R3 = 1…11 1111 1111 1111 $1000_2$ = $-8_{10}$

- After executing these instructions:

```
slt  R4,R2,R1 ;  if (R2 < R1) R4=1; else R4=0
slt  R5,R3,R1 ;  if (R3 < R1) R5=1; else R5=0
sltu R6,R2,R1 ;  if (R2 < R1) R6=1; else R6=0
sltu R7,R3,R1 ;  if (R3 < R1) R7=1; else R7=0
```

- What are values of registers R4 – R7? Why?

  R4 = 0 ; R5 = 1 ; R6 = 0 ; R7 = 0 ;

# MIPS Compare and Branch

- Compare and Branch
  - **beq rs, rt, offset** **if R[rs] == R[rt] then PC-relative branch**
  - **bne rs, rt, offset** <>

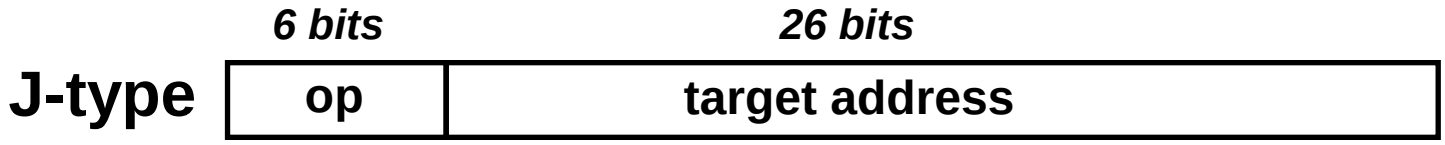
- Compare to Zero and Branch
  - blez rs, offset if R[rs] <= 0 then PC-relative branch
  - bgtz rs, offset >
  - bltz rs, offset <
  - bgez rs, offset >=

- Remaining set of compare and branch take two instructions

# MIPS Compare and Jump Instructions

| *Instruction* | *Example* | *Meaning* |
|---|---|---|
| set on less than | slt $1,$2,$3 | if ($2 < $3) $1=1; else $1=0 |
| | *Compare less than; 2's complement* | |
| set less than imm | slti $1,$2,100 | if ($2 < 100) $1=1; else $1=0 |
| | *Compare < constant; 2's complement* | |
| set less than uns. | sltu $1,$2,$3 | if ($2 < $3) $1=1; else $1=0 |
| | *Compare less than; unsigned numbers* | |
| set l. t. imm. uns. | sltiu $1,$2,100 | if ($2 < 100) $1=1; else $1=0 |
| | *Compare < constant; unsigned numbers* | |
| jump | j 10000 | go to (PC[31,28], 40000) |
| | *Jump to target address* | |
| jump and link | jal 10000 | $31 = PC + 4; go to (PC[31,28], 40000) |
| | *For procedure/subroutine call* | |
| jump register | jr $31 | go to $31 |
| | *For switch, procedure/subroutine return* | |

# J-Type Operands




- Constant data specified in an instruction

```
j 10000 ($PC <= $PC[31:28]:40000)
```


- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

# Target Addressing Example

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

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

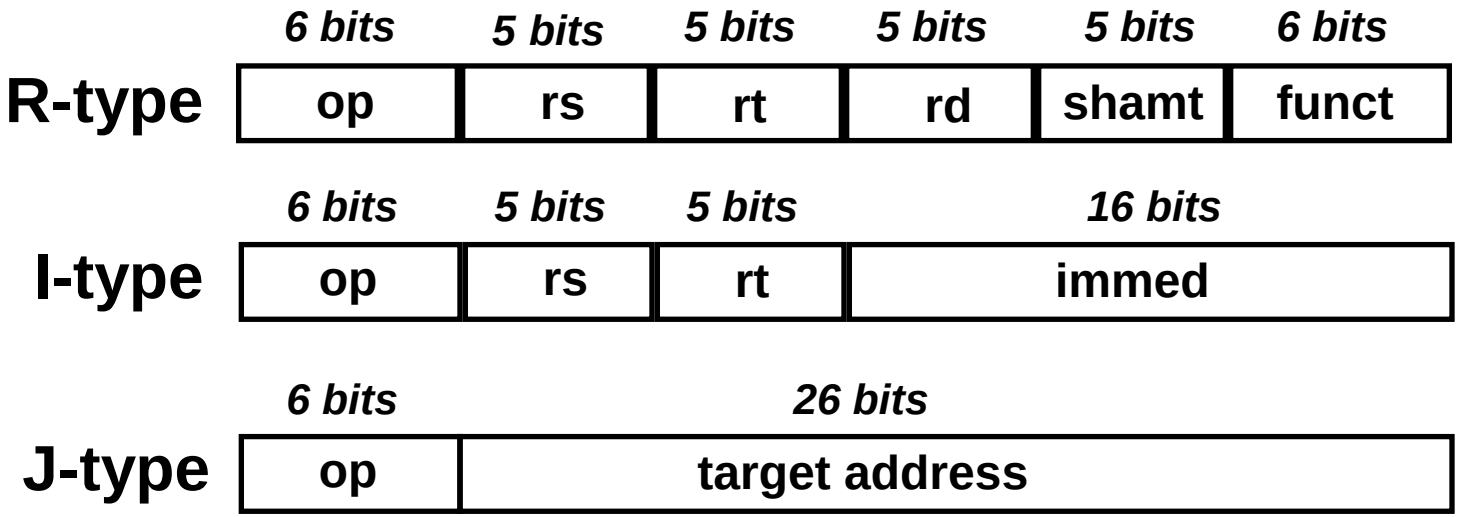
| | | | | | | |
|---|---|---|---|---|---|---|
| 80000 | 0 | 0 | 19 | 9 | 2 | 0 |
| 80004 | 0 | 9 | 22 | 9 | 0 | 32 |
| 80008 | 35 | 9 | 8 | 0 | | |
| 80012 | 5 | 8 | 21 | 2 | | |
| 80016 | 8 | 19 | 19 | 1 | | |
| 80020 | 2 | 20000 | | | | |
| 80024 | | | | | | |

# 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:     …
```

# MIPS Instruction Types

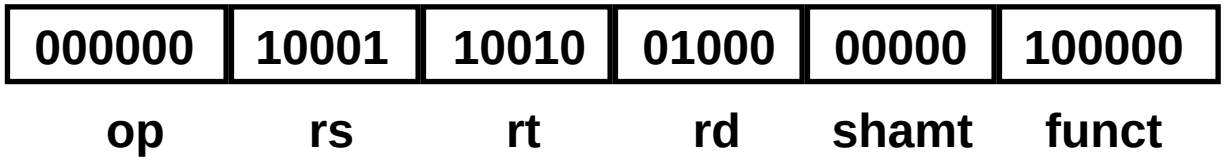



| Field | Meaning |
|---|---|
| op | Basic operation of the instruction (opcode) |
| rs | First register source operand |
| rt | Second register source operand |
| rd | Register destination operand (gets result) |
| shamt | Shift amount |
| funct | Function field – selects the variant of the operation in the op field (function code) |
| immed | Immediate value |

# Translating MIPS Assembly into Machine Language

- Humans see instructions as words (assembly language), but the computer sees them as ones and zeros (machine language).
- An assembler translates from assembly language to machine language.
- For example, the MIPS instruction add $t0, $s1, $s2 is translated as follows:

| Assembly | Comment |
|---|---|
| add | op = 0, shamt = 0, funct = 32 |
| $t0 | rd = 8 |
| $s1 | rs = 17 |
| $s2 | rt = 18 |

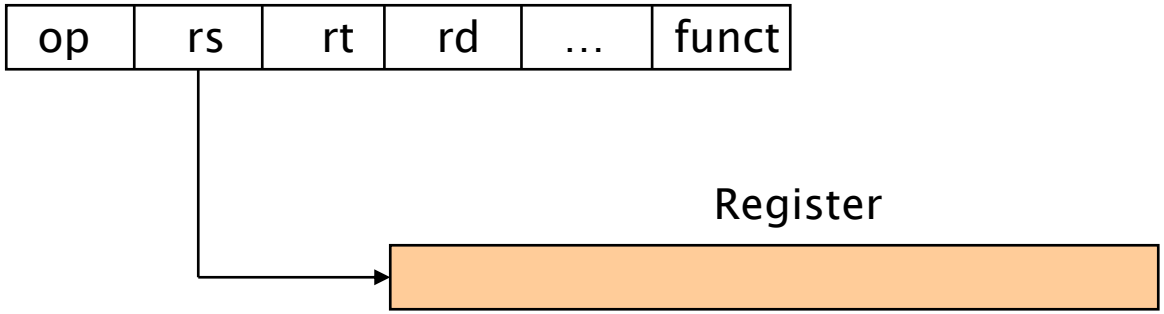


# MIPS Addressing Modes

- Addressing modes specify where the data used by an instruction is located.
- Often, the type of addressing mode depends on the type of operation being performed (e.g., branches all use PC relative)
- Five addressing modes are used in MIPS 3000:
  - Register addressing
  - Immediate addressing
  - Base addressing
  - PC-relative addressing
  - Pseudodirect addressing

# MIPS Addressing Modes (Cont'd)

## 1. Register addressing (R-Type)



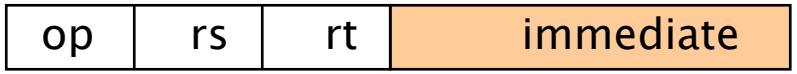

Example:

add $s1, $s2, $s3 ⟹ $s1 = $s2 + $s3

| 0 | 18 | 19 | 17 | 0 | 32 |
|---|---|---|---|---|---|

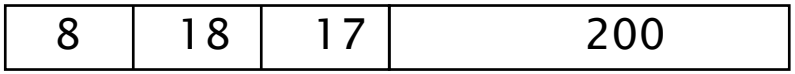
# MIPS Addressing Modes (Cont'd)

## 2. Immediate addressing (I-Type)



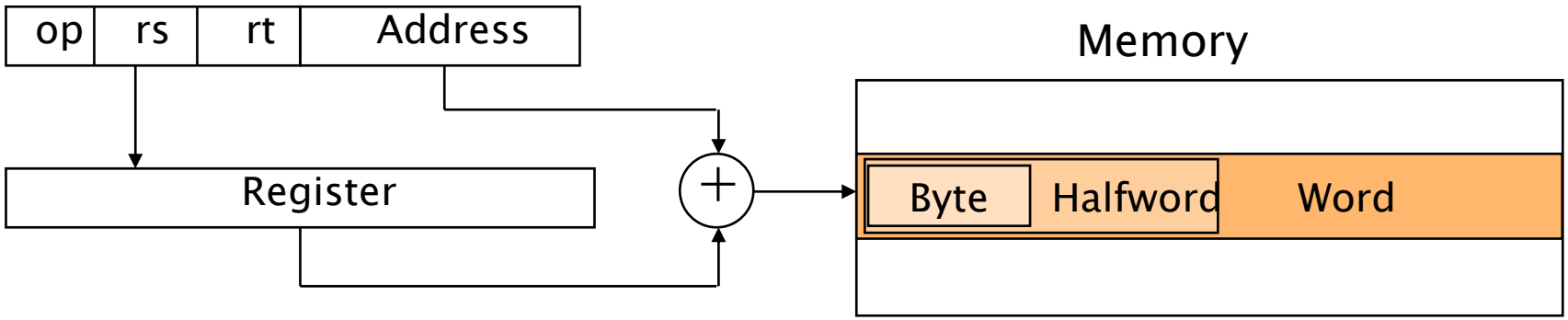

Example:

addi $s1, $s2, 200 ⟹ $s1 = $s2 + 200



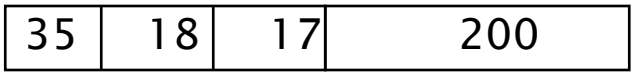
# MIPS Addressing Modes (Cont'd)

### 3. Base addressing (I-Type)



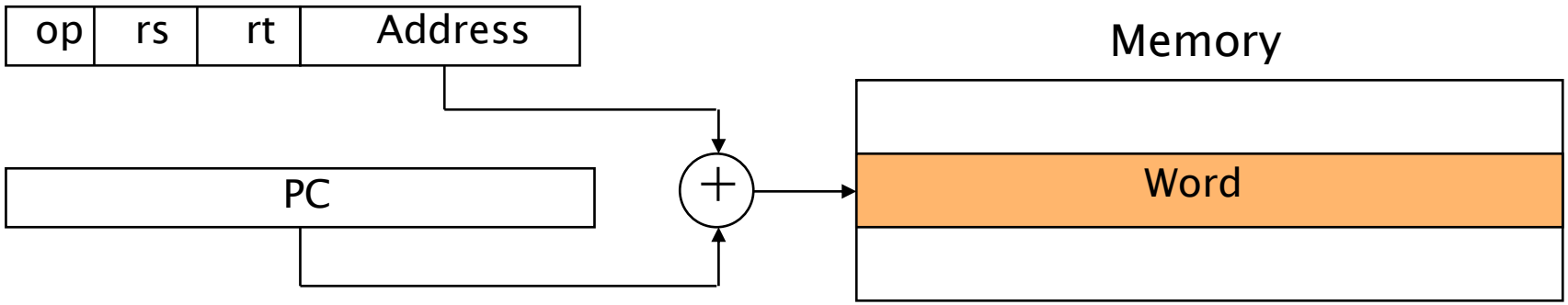

Example:

lw $s1, 200($s2) ⟹ $s1 = mem[200 + $s2]



# MIPS Addressing Modes (Cont'd)

## 4. PC-relative addressing (I-Type)



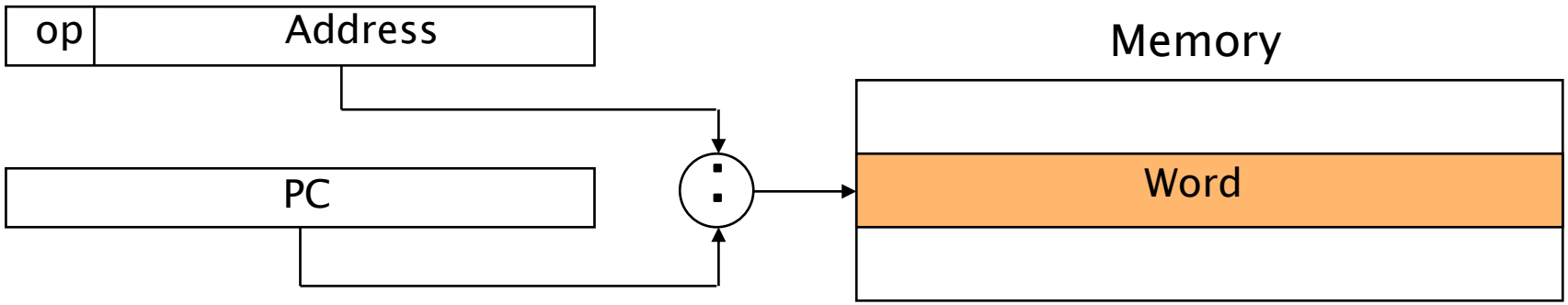

Example:

beq $s1, $s2, 200 ⟹ if ($s1 == $s2) PC = PC+4+200*4

| 4 | 18 | 17 | 200 |
|---|---|---|---|

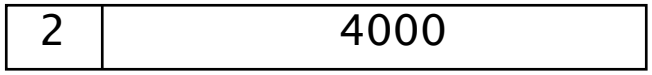
# MIPS Addressing Modes (Cont'd)

## 5. Pseudodirect addressing (J-Type)

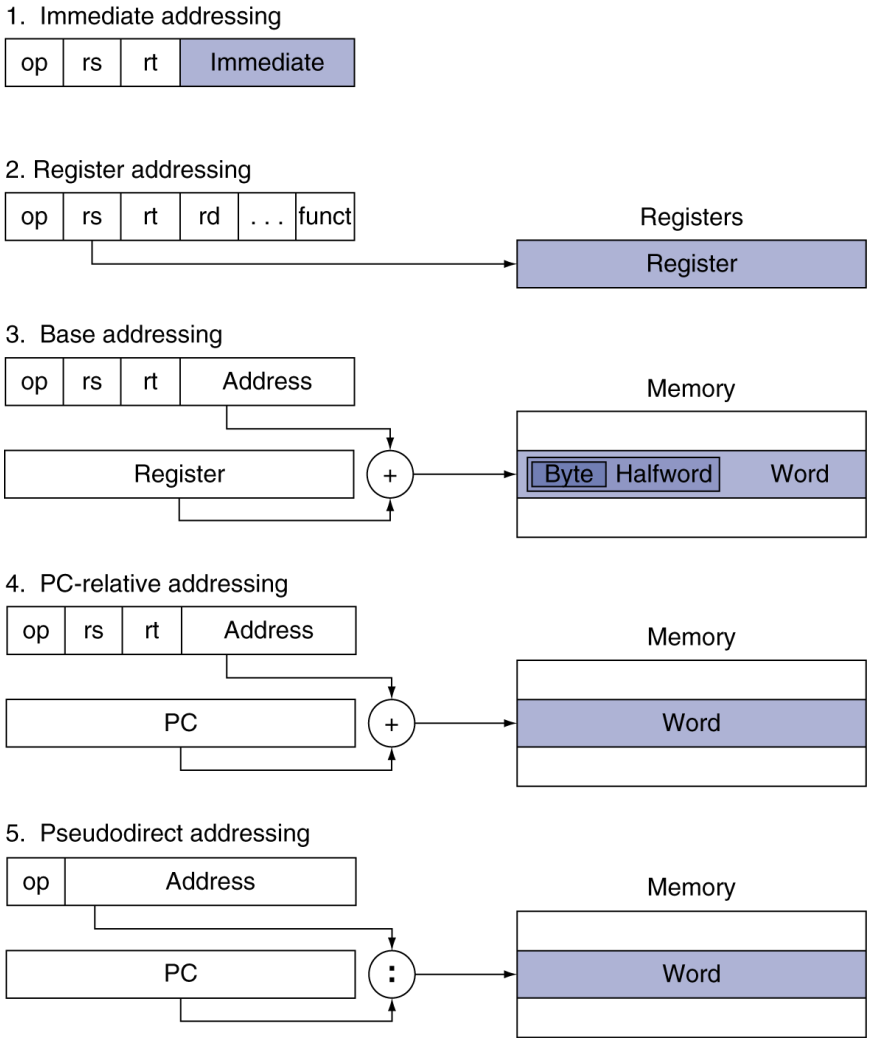



Example:

j 4000 ⟹ PC = (PC[31:28], 4000*4)



# Addressing Mode Summary



# An Assembly Example

```
int A[100], B[100], C[100];

for (i=1; i < 99; i++) {
        if (A[i] == B[i])
                C[i] = A[i];
        else    C[i] = A[i-1];
}
```

Assume base address of A is in $a0, B is in $a1, C is in $a2.
Use $t0 for variable i.

**Find bugs in this code and correct**

```
        addi$t0, $zero, 1
        addi $t5, $zero, 99
loop:
        lw $t1, 4($a0)
        lw $t2, 4($a1)
        beq $t1, $t2, equ
        lw $t1, 0($a0)
equ:    sw $t1, 0($a2)
        addi $t0, $t0, 1
        addi $a0, $a0, 4
        addi $a1, $a1, 4
        bne $t0, $t5, loop
exit:
        nop
```

# Pseudo-instructions

- The MIPS assembler supports several pseudo-instructions:
  - not directly supported in hardware
  - implemented using one or more supported instructions
  - simplify assembly language programming and translation

- For example, the pseudo-instruction

```
move $t0, $t1
```

  is implemented as

```
add $t0, $zero, $t1
```

- The pseudo-instruction

```
blt $s0, $s1, Else
```

  is implemented as

```
slt $at, $s0, $s1
bne $at, $zero, Else
```

# Details of The MIPS Instruction Set

- Register zero always has the value zero (even if you try to write it)
- Branch/jump and link put the return addr. PC+4 into the link register (R31)
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
- it cannot occur in
  - addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

# Summary: Features of MIPS ISA

- **32-bit fixed format inst** (3 formats)
- **31 32-bit GPR** (R0 contains zero) and 32 FP registers (and PC, HI, and LO)
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
- **Simple branch conditions**
  - compare one register against zero or two registers for $=, \neq$