

EE 204 Computer Architecture

MIPS: Case Study of Instruction Set Architecture

Instructor: Dr. Hassan Jamil Syed

Courtesy : Prof. Hong Jiang

Prof. Yifeng Zhu @ U. of Maine

Fall 2019

Outline - Instruction Sets

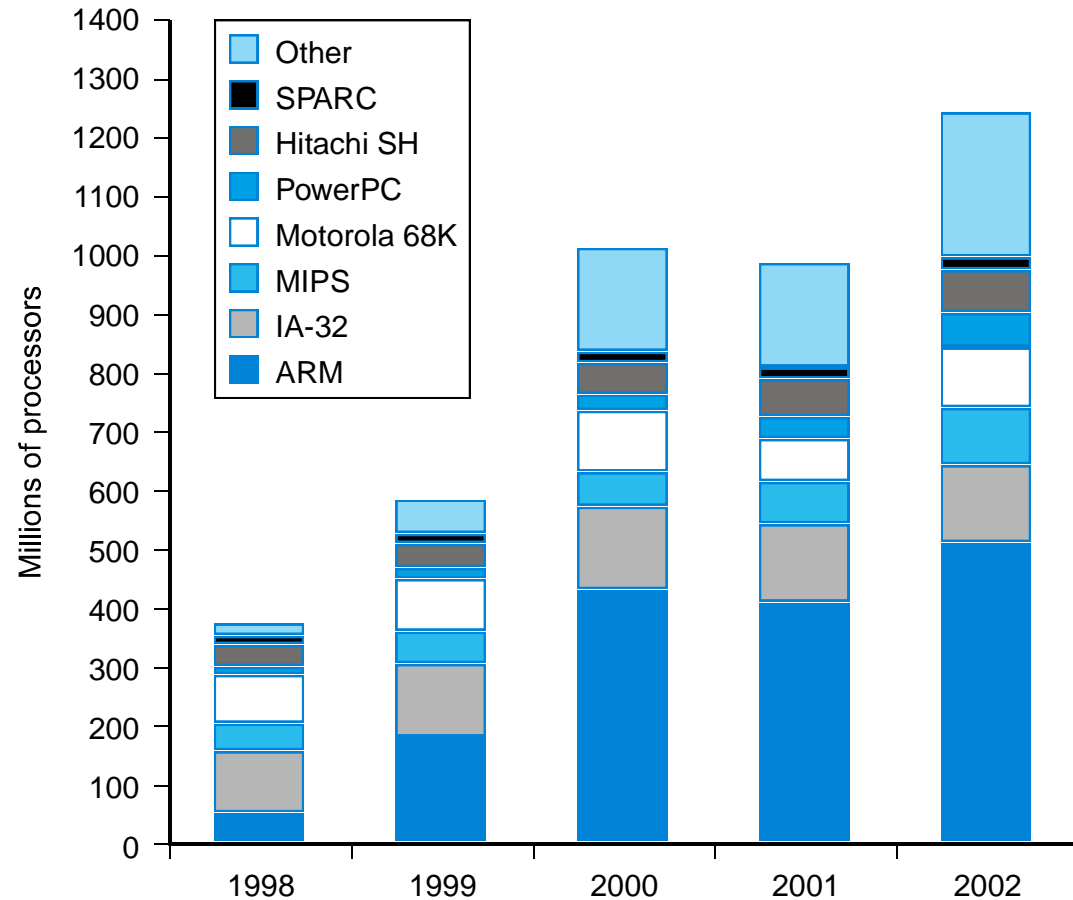
- **Instruction Set Overview**
- **MIPS Instruction Set**
 - **Overview** ↩
 - **Registers and Memory**
 - **MIPS Instructions**

MIPS

- **MIPS** (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies.
- The current revisions are MIPS32 (for 32-bit implementations) and MIPS64 (for 64-bit implementations).
- MIPS32 and MIPS64 define a **control register set** as well as the **instruction set**.

MIPS

- **MIPS**: **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
- We'll be working with the MIPS instruction set architecture
 - similar to other architectures developed since the 1980's
 - Almost 100 million MIPS processors manufactured in 2002
 - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...



MIPS Design Principles

1. **Simplicity Favors Regularity**

- Keep all instructions a single size
- Always require three register operands in arithmetic instructions

2. **Smaller is Faster**

- Has only 32 registers rather than many more

3. **Good Design Makes Good Compromises**

- Providing larger addresses and constants in instruction and keeping instruction the same length

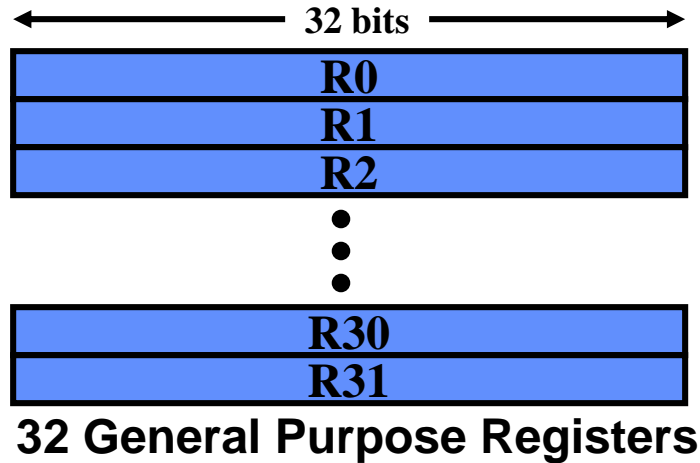
4. **Make the Common Case Fast**

- PC-relative addressing for conditional branches
- Immediate addressing for constant operands

Outline - Instruction Sets

- **Instruction Set Overview**
- **MIPS Instruction Set**
 - **Overview**
 - **Registers and Memory** ⇐
 - **MIPS Instructions**

MIPS32 Registers and Memory



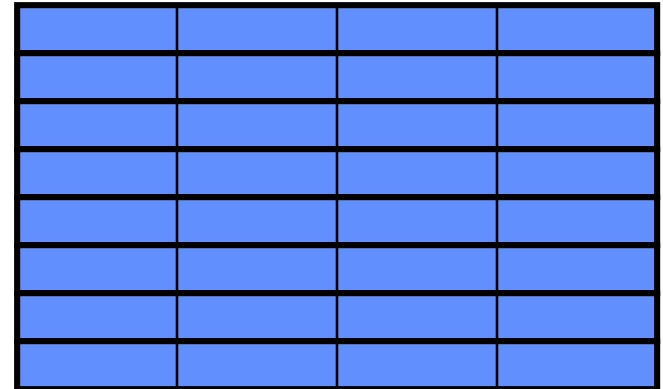
PC = 0x0000001C

Registers

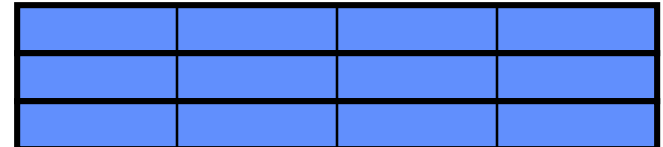
0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C

⋮

0xfffffffff4
0xfffffffffc
0xfffffffffc



⋮



Memory
4GB Max
(Typically 64MB-1GB)

MIPS64

Registers for MIPS MIPS64 has 32 64-bit general-purpose registers (GPRs), named R0, R1, . . . , R31.

GPRs are also sometimes known as integer registers.

The value of R0 is always 0

Set of 32 floating-point registers (FPRs), named F0, F1, . . . , F31, which can hold 32 single-precision (32-bit) values or 32 double-precision (64-bit) values.

The data types are 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit double words for integer data and 32-bit single precision and 64-bit double precision for floating point.

MIPS64

The MIPS64 operations work on 64-bit integers and 32- or 64-bit floating point.

Bytes, half words, and words are loaded into the general-purpose registers with either zeros or the sign bit replicated to fill the 64 bits of the GPRs.

Once loaded, they are operated on with the 64-bit integer operations.

MIPS Registers and Usage

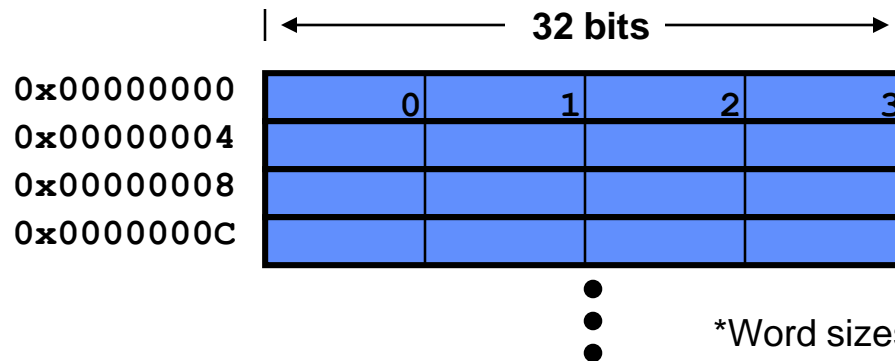
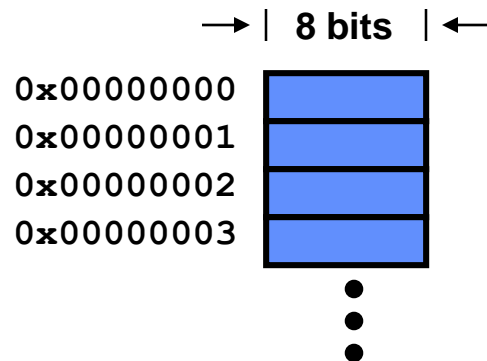
Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	reserved for assembler
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporary registers
\$s0-\$s7	16-23	saved registers
\$t8-\$t9	24-25	more temporary registers
\$k0-\$k1	26-27	reserved for Operating System kernel
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Each register can be referred to by number or name.

More about MIPS Memory Organization

- **Two** views of memory:
 - 2^{32} **bytes** with addresses 0, 1, 2, ..., $2^{32}-1$
 - 2^{30} 4-byte **words*** with addresses 0, 4, 8, ..., $2^{32}-4$
- Both views use **byte** addresses
- Word address must be multiple of 4 (**aligned**)

Not all architectures require this



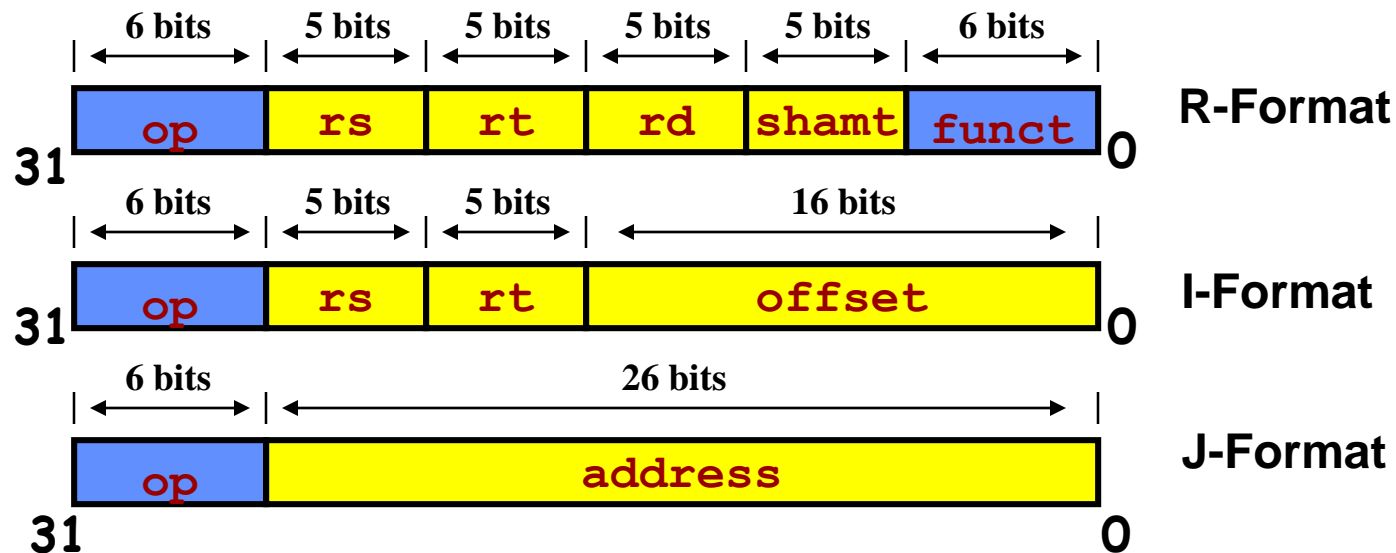
*Word sizes vary in other architectures

Outline - Instruction Sets

- **Instruction Set Overview**
- **MIPS Instruction Set**
 - Overview
 - Registers and Memory
 - MIPS Instructions ⇐
- **Summary**

MIPS Instructions

- All instructions exactly 32 bits wide
- Different formats for different purposes
- Similarities in formats ease implementation



MIPS instructions

- The only **data addressing modes** are immediate and displacement, both with 16-bit fields.
 - Register indirect is accomplished simply by placing 0 in the 16-bit displacement field, and
 - absolute addressing with a 16-bit field is accomplished by using register 0 as the base register.

MIPS Instruction Types

- **Arithmetic & Logical** - manipulate data in registers

add \$s1, \$s2, \$s3 \$s1 = \$s2 + \$s3
or \$s3, \$s4, \$s5 \$s3 = \$s4 OR \$s5

- **Data Transfer** - move register data to/from memory

lw \$s1, 100(\$s2) \$s1 = Memory[\$s2 + 100]
sw \$s1, 100(\$s2) Memory[\$s2 + 100] = \$s1

- **Branch** - alter program flow

beq \$s1, \$s2, 25 if (\$s1==\$s2) PC = PC + 4 + 4*25

MIPS Arithmetic & Logical Instructions

- **Instruction usage (assembly)**

add dest, src1, src2

dest=src1 + src2

sub dest, src1, src2

dest=src1 - src2

and dest, src1, src2

dest=src1 AND src2

- **Instruction characteristics**

- Always 3 operands: destination + 2 sources
- Operand order is fixed
- Operands are always general purpose registers

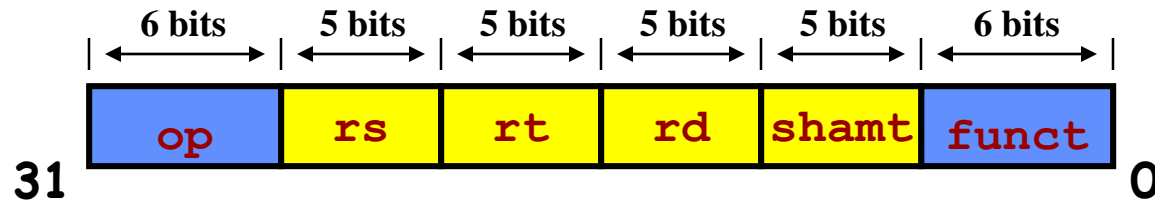
- **Design Principles:**

- Design Principle 1: **Simplicity favors regularity**
- Design Principle 2: **Smaller is faster**

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + \text{Regs}[R3]$
DADDIU R1,R2,#3	Add immediate unsigned	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] + 3$
LUI R1,#42	Load upper immediate	$\text{Regs}[R1] \leftarrow 0^{32} \# \# 42 \# \# 0^{16}$
DSLL R1,R2,#5	Shift left logical	$\text{Regs}[R1] \leftarrow \text{Regs}[R2] \ll 5$
SLT R1,R2,R3	Set less than	$\text{if } (\text{Regs}[R2] < \text{Regs}[R3])$ $\text{Regs}[R1] \leftarrow 1 \text{ else } \text{Regs}[R1] \leftarrow 0$

Figure A.24 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.

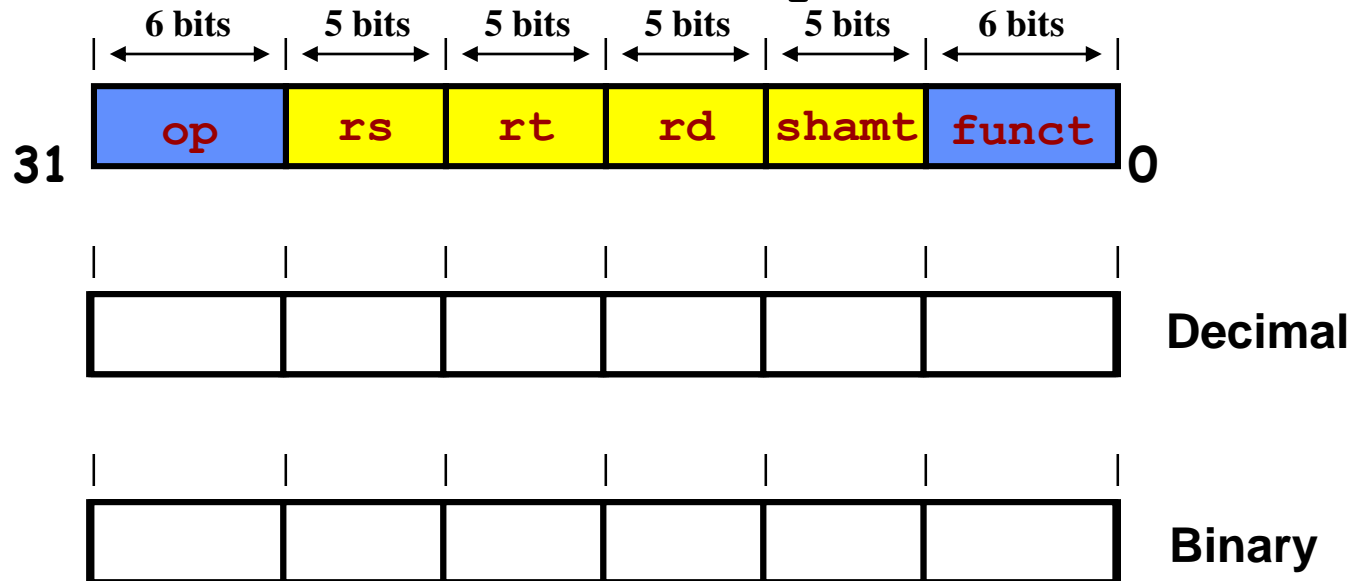
Arithmetic & Logical Instructions - Binary Representation



- Used for arithmetic, logical, shift instructions
 - **op**: Basic operation of the instruction (*opcode*)
 - **rs**: first register source operand
 - **rt**: second register source operand
 - **rd**: register destination operand
 - **shamt**: shift amount (more about this later)
 - **funct**: function - specific type of operation
- Also called “**R-Format**” or “**R-Type**” Instructions

Arithmetic & Logical Instructions - Binary Representation Example

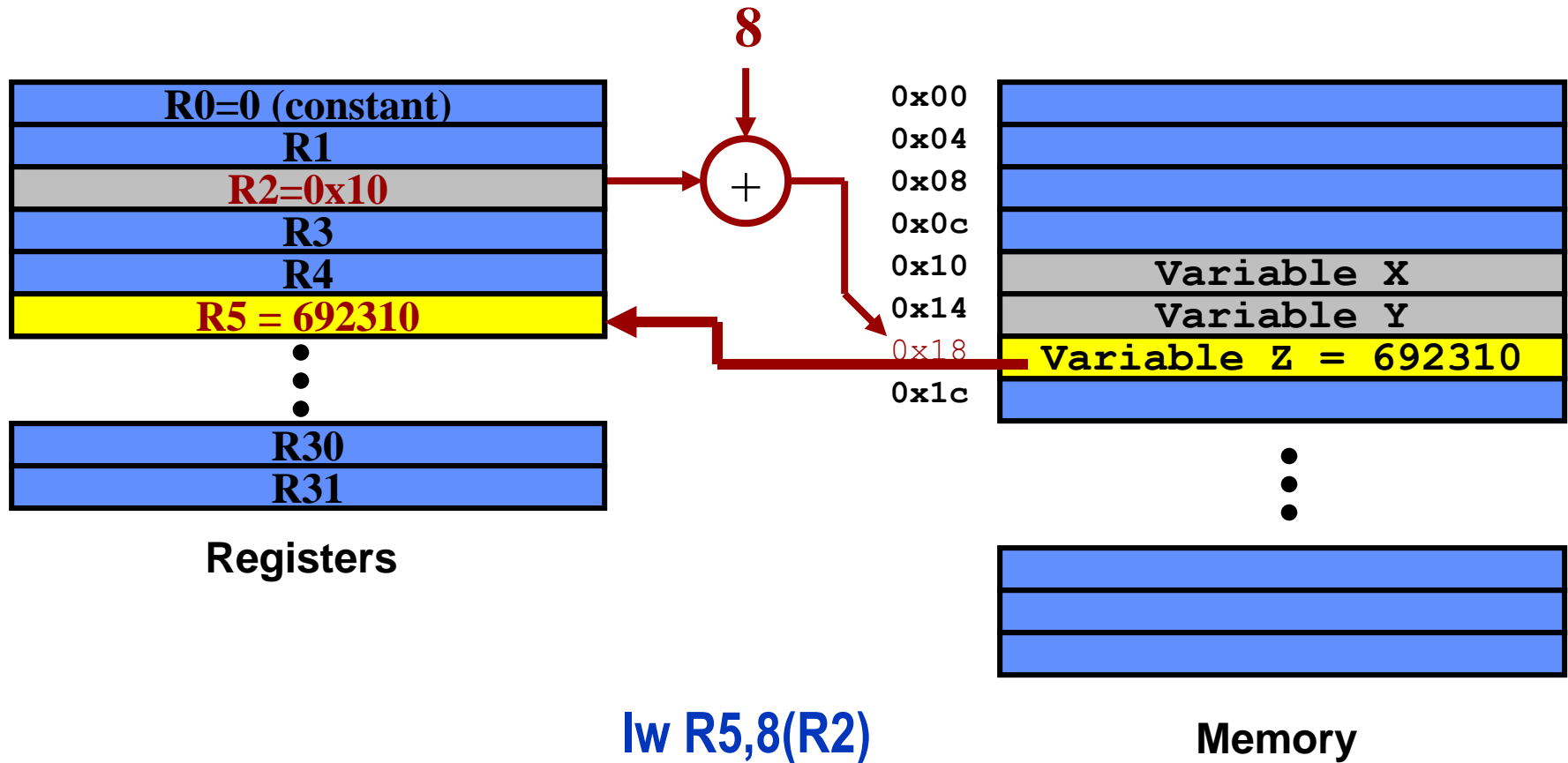
- Machine language for
`add $8, $17, $18`
- See reference card for `op`, `funct` values



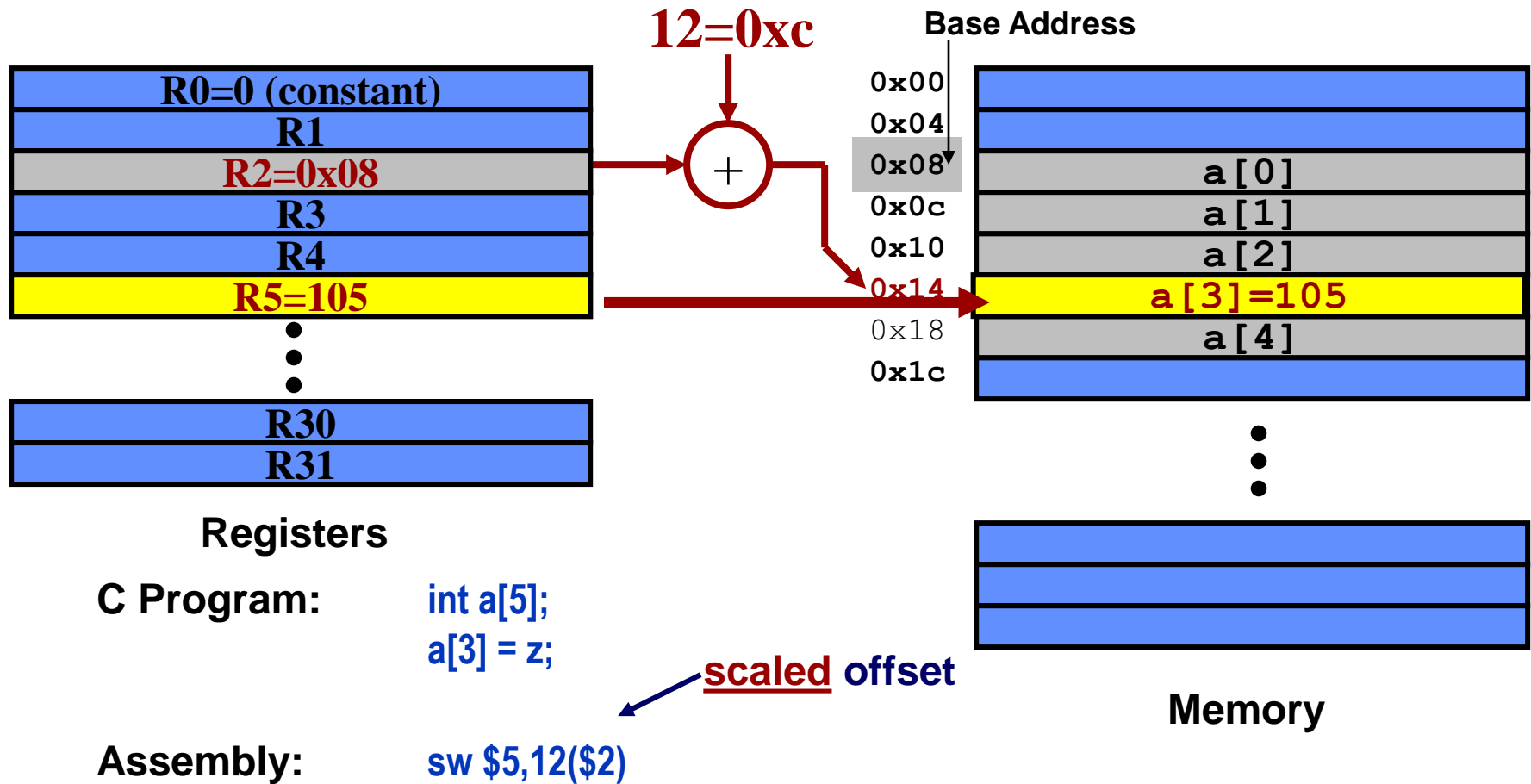
MIPS Data Transfer Instructions

- Transfer data between registers and memory
- Instruction format (assembly)
 - `lw $dest, offset($addr)` load word
 - `sw $src, offset($addr)` store word
- Uses:
 - Accessing a variable in main memory
 - Accessing an array element

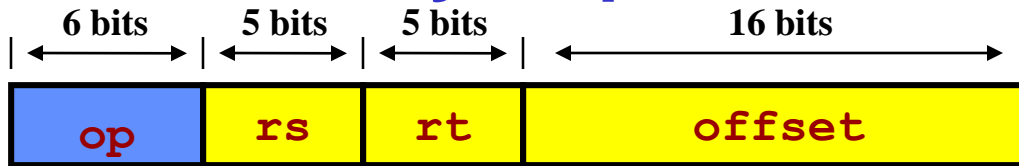
Example - Loading a Simple Variable



Data Transfer Example - Array Variable



Data Transfer Instructions - Binary Representation

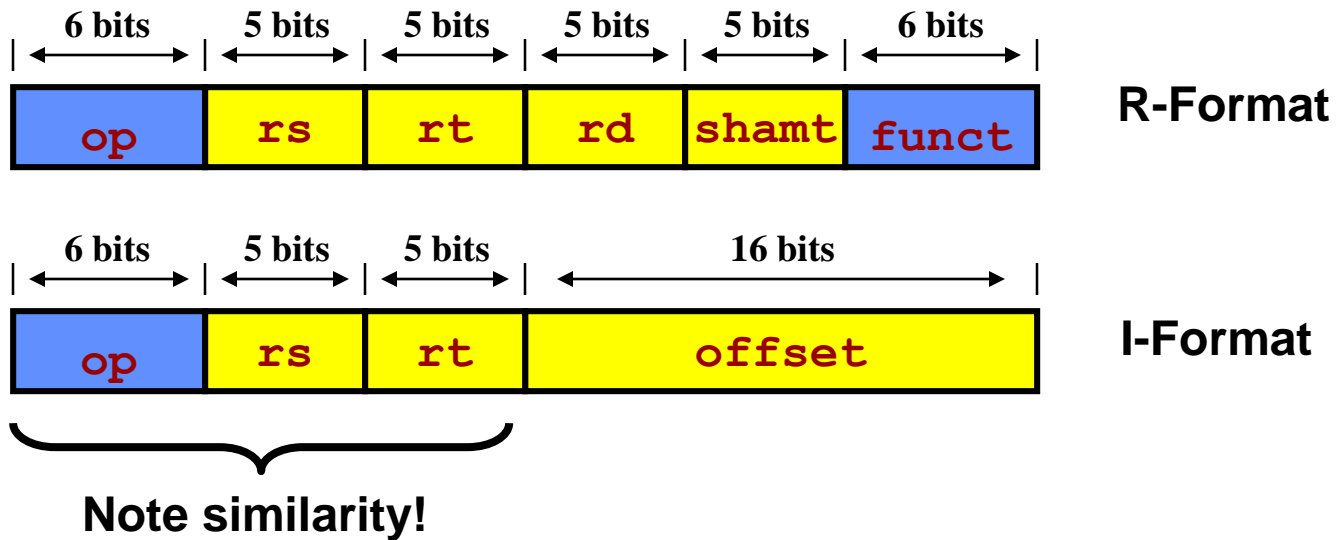


- Used for load, store instructions
 - **op**: Basic operation of the instruction (*opcode*)
 - **rs**: first register source operand
 - **rt**: second register source operand
 - **offset**: 16-bit signed address offset (-32,768 to +32,767)
- Also called “**I-Format**” or “**I-Type**” instructions

Address

I-Format vs. R-Format Instructions

- Compare with R-Format

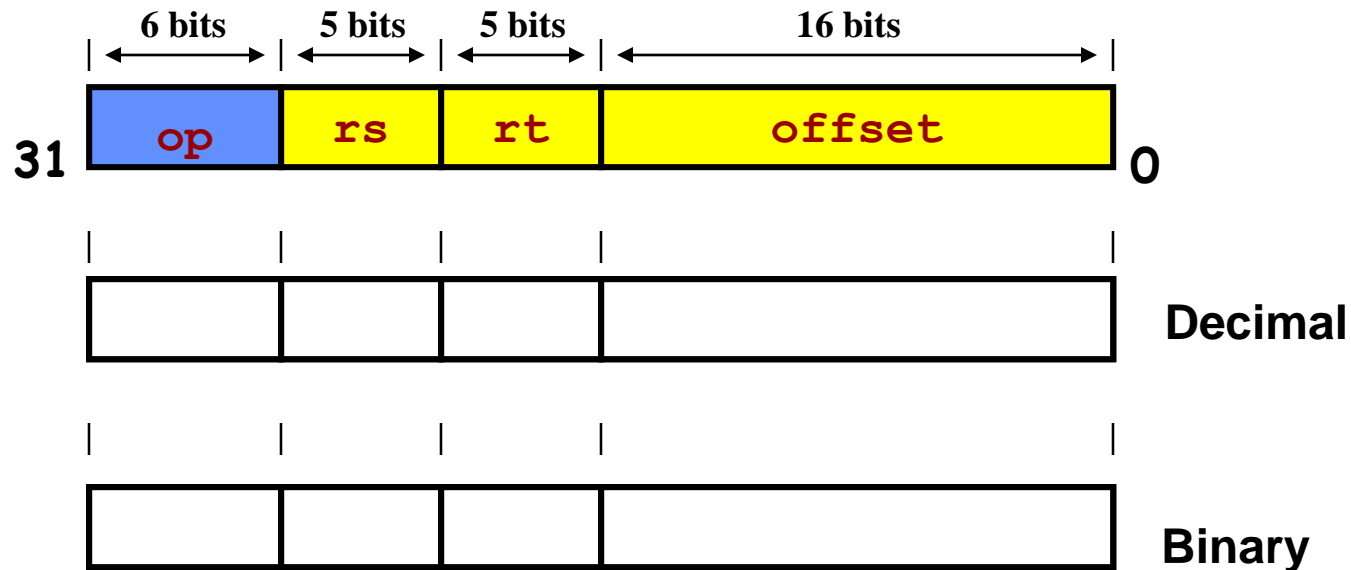


Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	$\text{Regs}[R1] \leftarrow_{64} \text{Mem}[30+\text{Regs}[R2]]$
LD R1,1000(R0)	Load double word	$\text{Regs}[R1] \leftarrow_{64} \text{Mem}[1000+0]$
LW R1,60(R2)	Load word	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[60+\text{Regs}[R2]])_0^{32} \text{ ## Mem}[60+\text{Regs}[R2]]$
LB R1,40(R3)	Load byte	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{56} \text{ ## Mem}[40+\text{Regs}[R3]]$
LBU R1,40(R3)	Load byte unsigned	$\text{Regs}[R1] \leftarrow_{64} 0^{56} \text{ ## Mem}[40+\text{Regs}[R3]]$
LH R1,40(R3)	Load half word	$\text{Regs}[R1] \leftarrow_{64} (\text{Mem}[40+\text{Regs}[R3]])_0^{48} \text{ ## Mem}[40+\text{Regs}[R3]] \text{ ## Mem}[41+\text{Regs}[R3]]$
L.S F0,50(R3)	Load FP single	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R3]] \text{ ## } 0^{32}$
L.D F0,50(R2)	Load FP double	$\text{Regs}[F0] \leftarrow_{64} \text{Mem}[50+\text{Regs}[R2]]$
SD R3,500(R4)	Store double word	$\text{Mem}[500+\text{Regs}[R4]] \leftarrow_{64} \text{Regs}[R3]$
SW R3,500(R4)	Store word	$\text{Mem}[500+\text{Regs}[R4]] \leftarrow_{32} \text{Regs}[R3]_{32..63}$
S.S F0,40(R3)	Store FP single	$\text{Mem}[40+\text{Regs}[R3]] \leftarrow_{32} \text{Regs}[F0]_{0..31}$
S.D F0,40(R3)	Store FP double	$\text{Mem}[40+\text{Regs}[R3]] \leftarrow_{64} \text{Regs}[F0]$
SH R3,502(R2)	Store half	$\text{Mem}[502+\text{Regs}[R2]] \leftarrow_{16} \text{Regs}[R3]_{48..63}$
SB R2,41(R3)	Store byte	$\text{Mem}[41+\text{Regs}[R3]] \leftarrow_8 \text{Regs}[R2]_{56..63}$

Figure A.23 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

I-Format Example

- Machine language for
`lw $9, 1200($8) == lw $t1, 1200($t0)`



MIPS Conditional Branch Instructions

- **Conditional branches allow decision making**

`beq R1, R2, LABEL` if R1==R2 goto LABEL
`bne R3, R4, LABEL` if R3!=R4 goto LABEL

- **Example**

C Code if (i==j) goto L1;
 f = g + h;
 L1: f = f - i;

Assembly `beq $s3, $s4, L1`
 `add $s0, $s1, $s2`
 L1: `sub $s0, $s0, $s3`

Example: Compiling C `if-then-else`

- **Example**

C Code

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

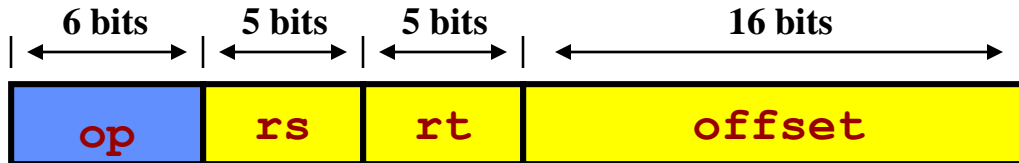
Assembly

```
bne $s3, $s4, Else  
add $s0, $s1, $s2  
j Exit;          # new: unconditional jump  
Else:  
sub $s0, $s0, $s3  
Exit:
```

- **New Instruction: Unconditional jump**

```
j LABEL  # goto Label
```

Binary Representation - Branch



- Branch instructions use **I-Format**
- **offset** is added to PC when branch is taken

`beq r0, r1, offset`

has the effect:

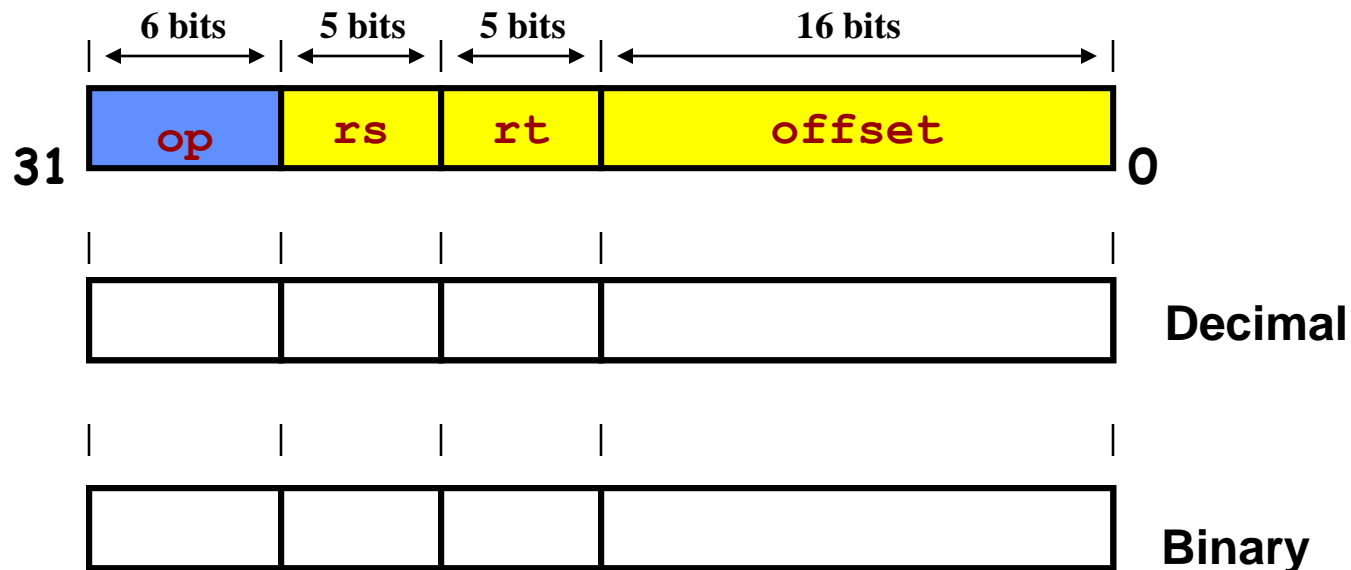
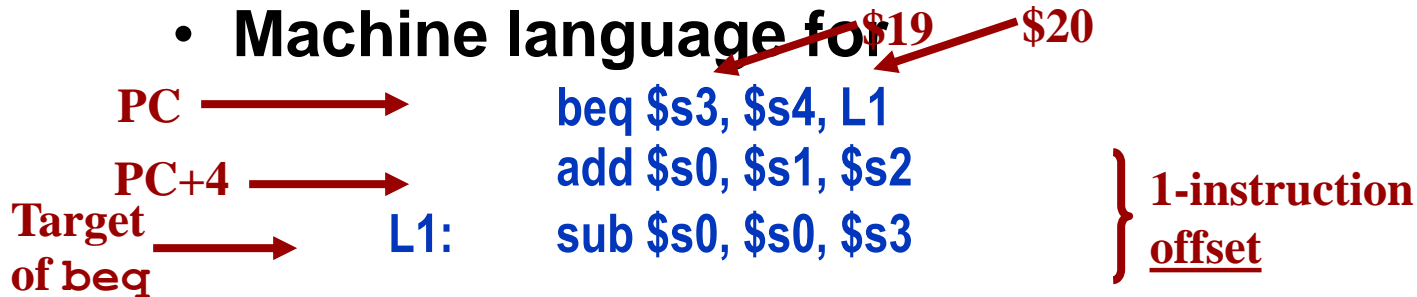
Conversion to
word offset

if ($r0 == r1$) $pc = pc + 4 + (\text{offset} \ll 2)$
else $pc = pc + 4;$

- Offset is specified in instruction **words** (why?)
- What is the range of the branch target addresses?

Branch Example

- Machine language for



Comparisons - What about <, <=, >, >=?

- bne, beq provide equality comparison
- slt provides magnitude comparison

condition register  `slt $t0,$s3,$s4` # if \$s3<\$s4 \$t0=1;
else \$t0=0;

- Combine with **bne** or **beq** to branch:

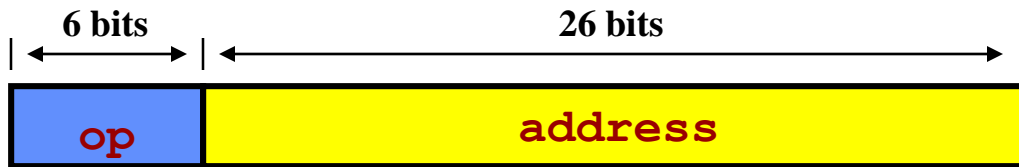
```
slt $t0,$s3,$s4      # if (a<b)
bne $t0,$zero, Less  # goto Less;
```

- Why not include a b1t instruction in hardware?
 - Supporting in hardware would lower performance
 - Assembler provides this function if desired (by generating the two instructions)

Example			
instruction		Instruction name	Meaning
J	name	Jump	$PC_{36..63} \leftarrow \text{name}$
JAL	name	Jump and link	$\text{Regs}[R31] \leftarrow PC+8$; $PC_{36..63} \leftarrow \text{name}$; $((PC+4)-2^{27}) \leq \text{name} < ((PC+4)+2^{27})$
JALR	R2	Jump and link register	$\text{Regs}[R31] \leftarrow PC+8$; $PC \leftarrow \text{Regs}[R2]$
JR	R3	Jump register	$PC \leftarrow \text{Regs}[R3]$
BEQZ	R4, name	Branch equal zero	if $(\text{Regs}[R4] == 0)$ $PC \leftarrow \text{name}$; $((PC+4)-2^{17}) \leq \text{name} < ((PC+4)+2^{17})$
BNE	R3, R4, name	Branch not equal zero	if $(\text{Regs}[R3] \neq \text{Regs}[R4])$ $PC \leftarrow \text{name}$; $((PC+4)-2^{17}) \leq \text{name} < ((PC+4)+2^{17})$
MOVZ	R1, R2, R3	Conditional move if zero	if $(\text{Regs}[R3] == 0)$ $\text{Regs}[R1] \leftarrow \text{Regs}[R2]$

Figure A.25 Typical control flow instructions in MIPS. All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are longer than the address field would suggest; since MIPS instructions are all 32 bits long, the byte branch address is multiplied by 4 to get a longer distance.

Binary Representation - Jump



- Jump Instruction uses J-Format (op=2)
- What happens during execution?

$$PC = PC[31:28] : (IR[25:0] \ll 2)$$

Concatenate
upper 4 bits
of PC to form
complete
32-bit address

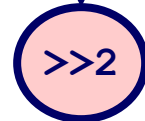
Conversion to
word offset

Jump Example

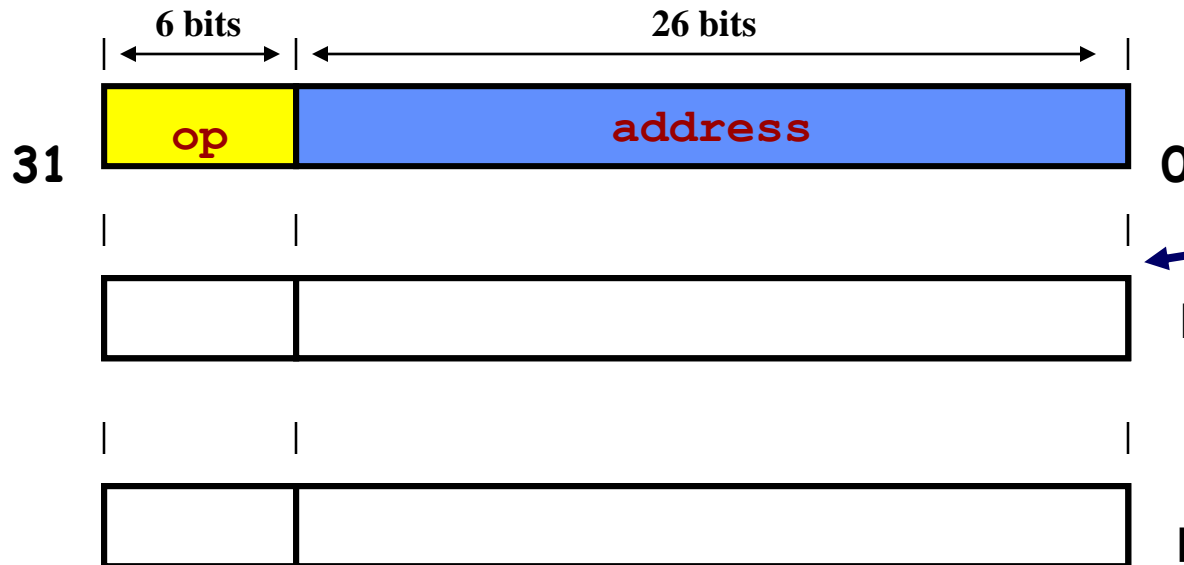
- Machine language for **j L5** ← Assume L5 is at address 0x00400020 and

PC <= 0x03FFFFFF

lower 28 bits



0x0100008



26 bit offset

- The MIPS uses 32-bit addresses, but only allows 26 bits to specify an offset in the jump instruction.
- The 26-bit target is extended to 28 bits by adding two 0's on the right.
- This works because all MIPS instructions are 32-bits, and must begin at a word boundary (an address which is a multiple of 4). Such an address always ends in 00.
- Hence, the jump instruction offset is limited to 2^{28} bytes, or 2^{26} instructions.
- This is not a problem, since most programs are nowhere near this large.
- Most memory is used for data, not code.
- If such a huge program were created, it could simply use multiple jumps to get from one end to the other.

Constants / Immediate Instructions

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

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

- MIPS Immediate Instructions (I-Format):

$\text{addi } \$29, \$29, 4$	}	Arithmetic instructions sign-extend immed.
$\text{slti } \$8, \$18, 10$		
$\text{andi } \$29, \$29, 6$	}	Logical instructions <u>don't</u> sign extend immed.
$\text{ori } \$29, \$29, 4$		

- Allows up to 16-bit constants
- How do you load just a constant into a register?

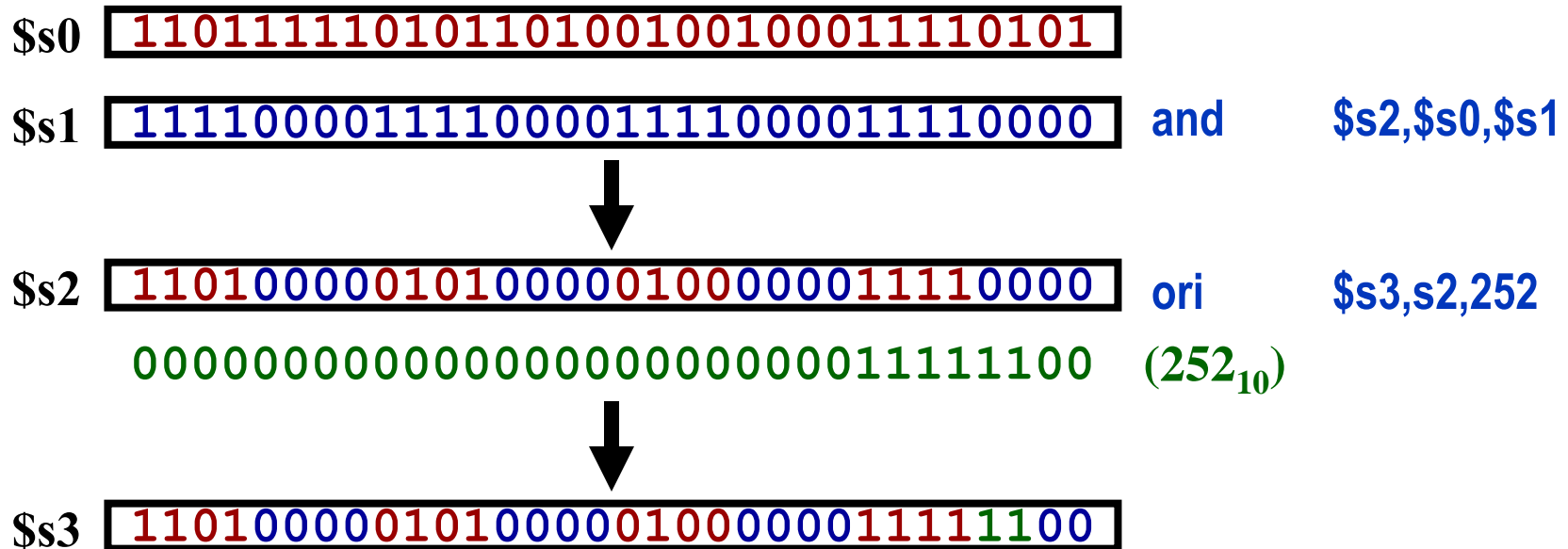


Why are immediates only 16 bits?

- Because 16 bits fits neatly in a 32-bit instruction
- Because most constants are small (i.e. < 16 bits)
- Design Principle 4: **Make the Common Case Fast**

MIPS Logical Instructions

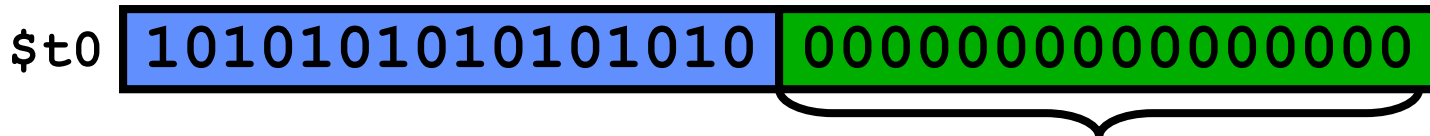
- **and, andi** - bitwise AND
- **or, ori** - bitwise OR
- **Example**



32-Bit Immediates and Address

- Immediate operations provide for 16-bit constants.
- What about when we need larger constants?
- Use "load upper immediate - lui" (I-Format) to set the upper 16 bits of a constant in a register.

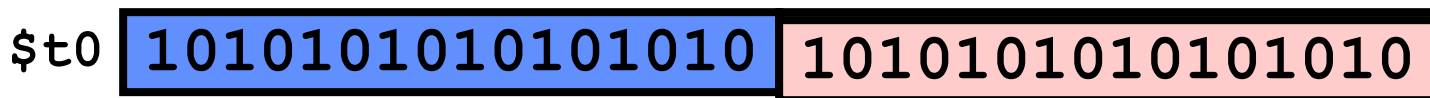
`lui $t0, 1010101010101010`



filled with zeros

- Then use `ori` to fill in lower 16 bits:

`ori $t0, $t0, 1010101010101010`

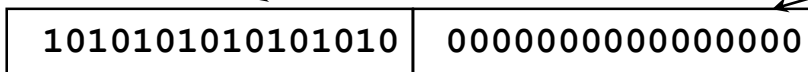


How about larger constants?

- 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 zeros

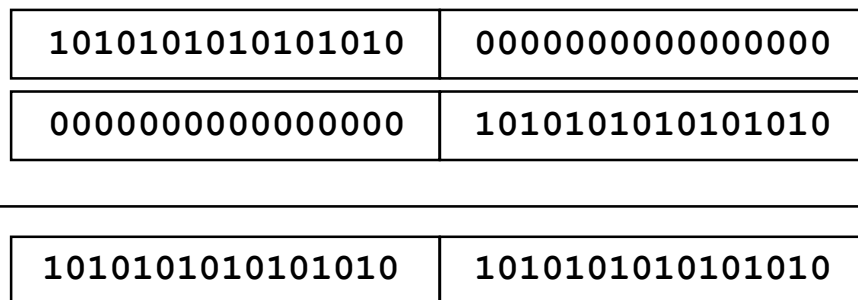


1010101010101010	0000000000000000
------------------	------------------

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

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

ori

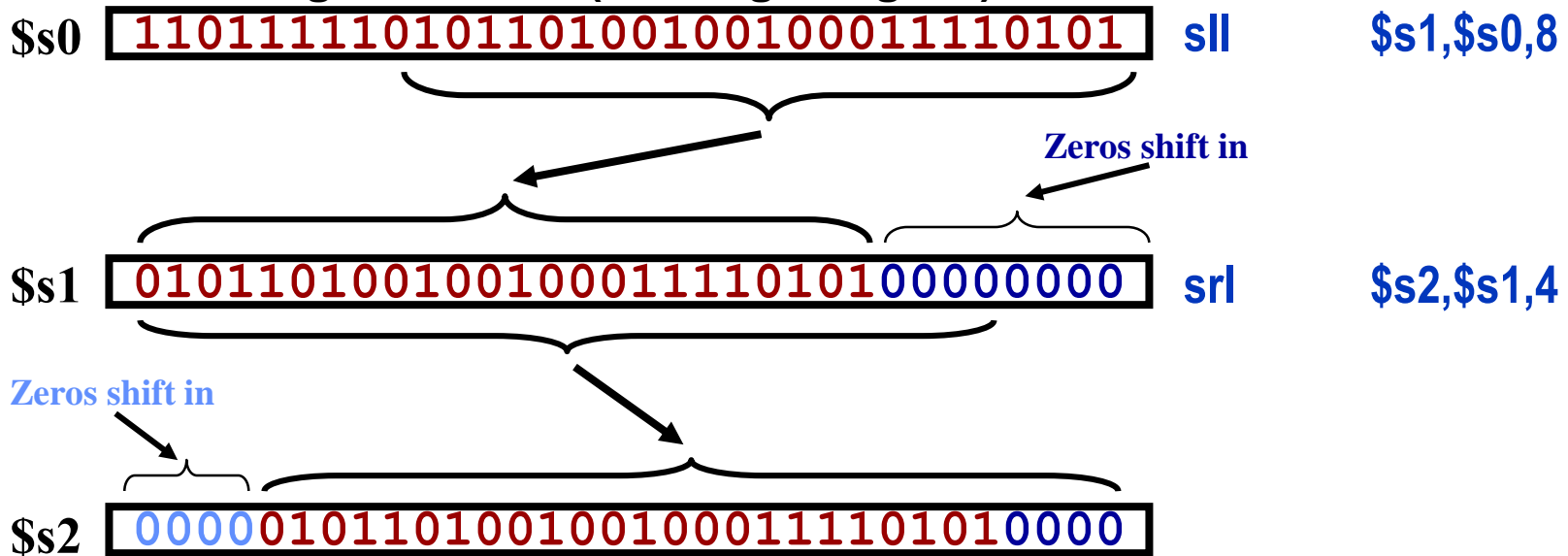


1010101010101010	0000000000000000
0000000000000000	1010101010101010
<hr/>	
1010101010101010	1010101010101010

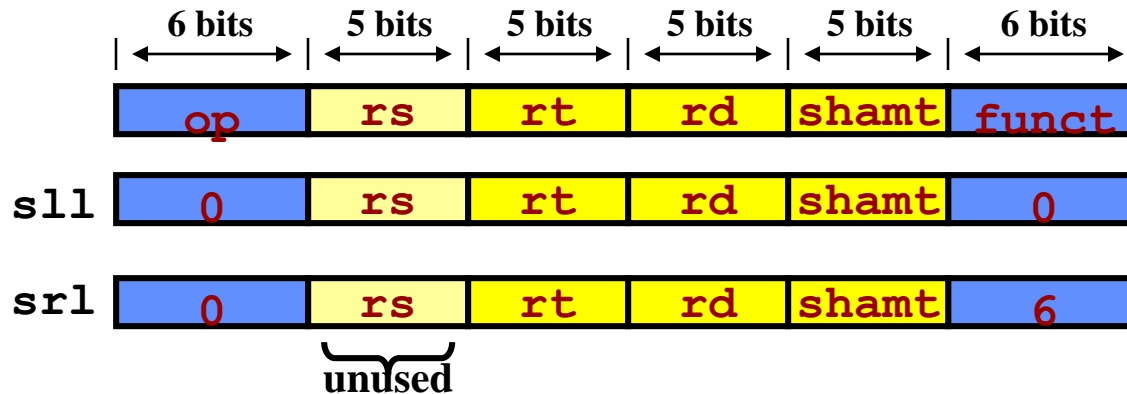
MIPS Shift Instructions

- MIPS Logical Shift Instructions

- Shift left: sll (shift-left logical) instruction
- Right shift: srl (shift-right logical) instruction



Shift Instruction Encodings



- **Applications**

- Bitfield access (see book)
- Multiplication / Division by power of 2
- Example: array access

```
sll $t0,$t1,2 # $t0=$t1*4  
add $t3,$t1,$t2  
lw $t3, 0($t3)
```

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register.
Immediate	Add R4,#3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants.
Displacement	Add R4,100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4,(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1 + R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$.
Autoincrement	Add R1,(R2)+	$\begin{aligned} \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \\ \text{Regs}[R2] &\leftarrow \text{Regs}[R2] + d \end{aligned}$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d .
Autodecrement	Add R1,-(R2)	$\begin{aligned} \text{Regs}[R2] &\leftarrow \text{Regs}[R2] - d \\ \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \end{aligned}$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	$\begin{aligned} \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] \\ &\quad + \text{Regs}[R3] * d] \end{aligned}$	Used to index arrays. May be applied to any indexed addressing mode in some computers.

How to Decode?

- What is the assembly language statement corresponding to this machine instruction?

0x00af8020

- Convert to binary

0000 0000 1010 1111 1000 0000 0010 0000

- Decode

- op: 00000
- rs: 00101
- rt: 01111
- rd: 10000
- shamt: 00000
- funct: 100000

- Solution: **add \$s0, \$a1, \$t7**

Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.

Summary - MIPS Instruction Set

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

