Computer Organization

Lecture 6 - Instruction Sets: MIPS

Reading: 2.3-2.7

Homework: In UCLASS

Image Source: MIPS, Inc. www.mips.com

Outline - Instruction Sets

- Instruction Set Overview
- MIPS Instruction Set
  - Overview
  - Registers and Memory
  - MIPS Instructions
- Software Concerns
- Summary

Top 5 Reasons to Study MIPS

5. It's in the book

4. It's used in many applications

3. Learning its architecture and implementation exposes you to important concepts

2. It's relatively simple and easy to implement (compared to other architectures)

1. Ideas presented using MIPS generalize to other architectures (even IA-32!)

MIPS Architecture - Applications

- **Workstations/Servers**
  - **SGI Workstations**
  - **SGI Servers**
- **Embedded Applications - examples**
  - **Network Routers**
  - **Laser Printers**
  - **Digital Cameras**
  - **Media Players**
  - **Game Consoles: Sony PS 2 / PSP**
  - **PDAs / Tablet Computers**
  - **Sony AIBO & QRIO Robots**

Image Source: www.aibo-life.com

Images Source: www.sony.com

Android "Ice Cream Sandwich" Tablet
Image Source: www.talkandroid.com

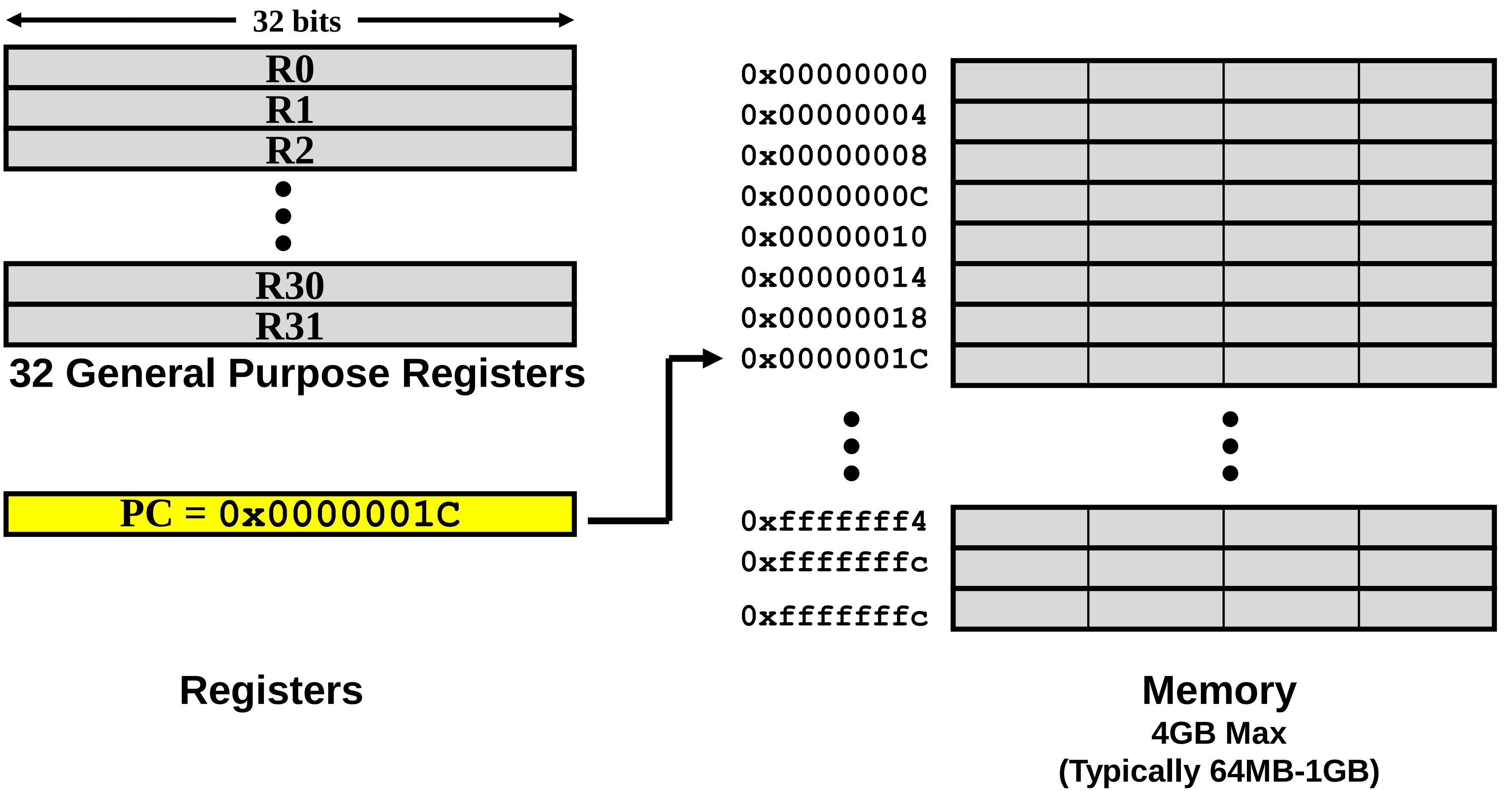
MIPS Design Principles

1. Simplicity Favors Regularity
   (규칙성 유지)
2. Smaller is Faster
   (더 작은 주소, 적은 수의 소자)
3. Good Design Makes Good Compromises
   (경우에 따라서는 부분적으로 원칙을 깨는 디자인)
4. Make the Common Case Fast
   (사용 빈도가 높은 것을 최적화)

Outline - Instruction Sets

- Instruction Set Overview
- MIPS Instruction Set
  - Overview
  - Registers and Memory
  - Review: Unsigned & Signed Binary Numbers
  - MIPS Instructions
- Software Concerns
- Summary

MIPS Registers and Memory



MIPS Registers

- **Fast access to program data**
- **Register `R0/$0/$zero`: hardwired to constant zero**
- **Register names:**
  - **`$0-$31 or R0-R31`**
  - **Specialized names based on usage convention**
    - **`$zero ($0)` - always zero**
    - **`$s0-$s7 ($16-$23)` - "saved" registers**
    - **`$t0-$t7 ($8-$15)` - "temporary" registers**
    - **`$sp` - stack pointer**
    - **Other special-purpose registers**

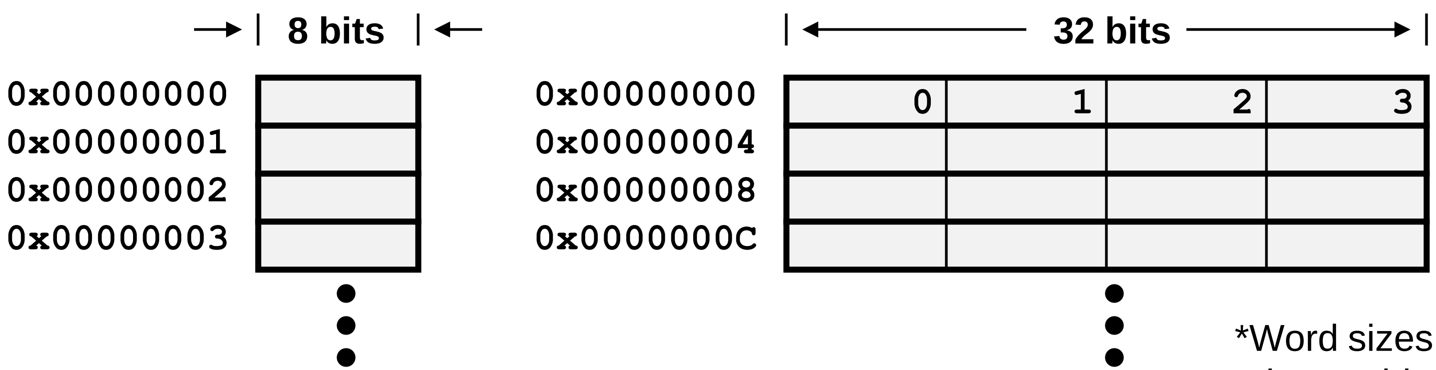
MIPS Register Usage

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

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**
  - **Review: Unsigned & Signed Binary Numbers**
  - **MIPS Instructions**
- Software Concerns
- Summary

Review: 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**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

$= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Range of Unsigned Binary Integers

Number of Digits	Smallest Value	Largest Value
n	0	2^n-1
8	0	2^8-1 = 255
16	0	2^{16}-1 = 65,535
32	0	2^{32}-1 = 4,294,967,295
64	0	2^{64}-1 = 1.8446 X 10^{19}

When n = 8, the largest unsigned value is:

1	1	1	1	1	1	1	1

$$2^8 - 1 = 255$$

Review: Unsigned vs. Signed Integers

- **Basic binary - allows representation of non-negative numbers only**

  In C, Java, etc: `unsigned int x;`

  Useful for unsigned quantities like addresses

- **Most of us need negative numbers, too!**

  In C, Java, etc: `int x;`

How can we do this?

… Use a signed representation

Signed Number Representations

- **4 methods to represent a number**
  - **Sign/Magnitude**
  - **Two's Complement - the one almost everyone uses**
  - **One's Complement**
  - **Biased - used for exponent in Floating Point**

Sign/Magnitude Representation

- **Approach: Use binary number and added <u>sign</u> bit**

  Ex) 8-bit representation

Sign	Magnitude							
1	0	0	1	1	0	0	1	= -25

- **Problems:**
  - **Two values of zero**
  - **Difficult to implement in hardware - consider addition**
    - **Must first check signs of operands**
    - **Then compute value**
    - **Then compute sign of result**

Two's Complement Signed Integers

- Goal: make the hardware easy to design
- Approach: explicitly represent result of "borrow" in subtract
  - Borrow results in "leading 1's"
  - Weight leftmost "sign bit" with -2^{n-1}
  - All negative numbers have a "1" in the sign bit
  - Single representation of zero (do not worry about +0 and -0)
  - Range: -2^{n-1} to $+2^{n-1}-1$

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

Negative value	-0011
Two's complement	1101

```
 1000        Borrowed (2^(n-1))
–0011 (= -3)
======
 0101

 1101 (= -3)    Return (-2^(n-1))
```

sign bit

- We borrowed MSB 1 in representing a negative value as two's complement
- Thus, when we get the actual value from two's complement, we return the borrowed MSB 1 by representing the sign bit (MSB) as a negative value

Two's Complement Examples

$N = 1111_{tc} = 1\times2^0 + 1\times2^1 + 1\times2^2 + 1\times-2^3 = 7_{10} + -8_{10}$
$= -1_{10}$

$N = 1001_{tc} = 1\times2^0 + 0\times2^1 + 0\times2^2 + 1\times-2^3 = 1_{10} + -8_{10}$
$= -7_{10}$

$N = 0101_{tc} = 1\times2^0 + 0\times2^1 + 1\times2^2 + 0\times-2^3 = 1_{10} + 4_{10}$
$= 5_{10}$

$N = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{tc}$
$= 0\times2^0 + 0\times2^1 + 1\times2^2 + \ldots + 1\times2^{30} + 1\times-2^{31}$
$= 2,147,483,644 + -2,147,483,648$
$= -4_{10}$

Range of Two's Complement Integers

Number of Digits	Most Negative Value	Most Positive Value
n	-2^{n-1}	$+2^{n-1}-1$
8	$-2^{7} = -128$	$+2^{7}-1 = +127$
16	$-2^{15} = -32,768$	$+2^{15}-1 = +32,767$
32	$-2^{31} = -2,147,483,648$	$+2^{31}-1 = +2,147,483,647$
64	$-2^{63} = -9.22 \times 10^{18}$	$+2^{63}-1 = +9.22 \times 10^{18}$

Two's Complement Negation Shortcut
(high school version)

- **Simple Procedure:**
  1. Complement (Invert) the individual bits
  2. Add 1

- **Examples (with 4 bits):**

```
- (0111tc) = 1000 + 1 = 1001   = -7ten
- (1100tc) = 0011 + 1 = 0100   = +4ten
- (1111tc) = 0000 + 1 = 0001   = +1ten
```

- **Example (with 32 bits):**

```
+2 = 0000 0000 … 0010tc
−2 = 1111 1111 … 1101tc + 1
   = 1111 1111 … 1110tc
```

Sign Extension of TC Integers

- **To convert a "narrower" number to a "wider" one:**
  - **Copy the bits of the narrower number into the lower bits**
  - **Copy the sign bit from the narrower number into <u>all</u> of the remaining bits of the result**
- **Example: Converting 8-bit byte to 32-bit word:**

`11111010` **Orignal byte: `–6`**

`111111111111111111111111` `11111010` **Result word: `–6`**

`00101011` **Orignal byte: `45`**

`000000000000000000000000` `00101011` **Result word: `45`**

Other Signed Binary Representations

- One's Complement
  - Use one's complement (inverted bits) to represent negated numbers

```
+1 = 0001
-1 = Invert(0001) = 1110
```

  - Problem: two values of zero (`0000`, `1111`)
- Biased
  - Add a bias (offset) of $2^{n-1}-1$ to represent all numbers

biased representation   real value

    - Most negative number: `000...0` = $-(2^{n-1}-1)$
    - Zero: `011...1` = 0
    - Most positive number: `111...1` = $+2^{n-1}$
  - Used in IEEE floating point representation for exponent (more about this later)

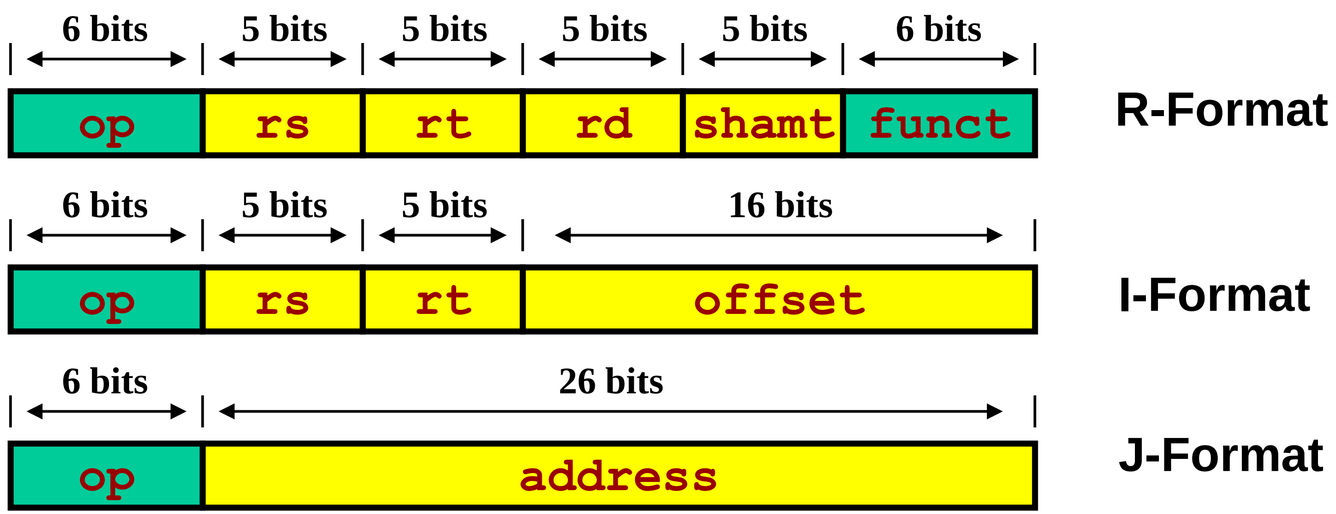
$$2^n-1 - (2^{n-1}-1) = +2^{n-1}$$

Outline - Instruction Sets

- Instruction Set Overview
- MIPS Instruction Set
  - Overview
  - Registers and Memory
  - Review: Unsigned & Signed Binary Numbers
  - MIPS Instructions
- Software Concerns
- Summary

MIPS Instructions

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



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
```

- 모든 명령어는 4바이트 단위로 표현되기 때문에 명령어 주소는 4의 배수가 됨. 따라서 명령어 주소의 마지막 2비트는 항상 00임.

- 명령어를 기계어로 표현할 때는 공간 절약을 위해서 실제 branch할 offset 값을 4로 나눈 후에 표현. 25는 실제는 100이 됨.

MP가 명령어를 fetch하는 PC값은 자동으로 4만큼 증가하기 때문에 실제 branch할 주소는 (PC+4)에서 branch할 값 (4*25)을 더하게 됨

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**

5 bits to address any register

Arithmetic Instruction Examples

- **C simple addition and assignment**

  C code: `A = B + C`

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

  변수 B를 $s1에 읽어오고
  변수 c를 $s2에 읽어오는 명령어는 생략
  전체 프로그램은 다음 페이지에 있음

- **Complex arithmetic assignment:**

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

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

- **Compiler keeps track of mapping variables to registers (and, when necessary, "spills" to memory)**

Assembly programming

```
# ----------------------------------------
#                  A = B + C
# ----------------------------------------
        .text
        .global _start

_start:
        lw $s1, B
        lw $s2, C
        add $s0, $s1, $s2        # A = B + C
        sw $t0, A

        .data
A:      .word 0
B:      .word 20
C:      .word 30
```

Arithmetic & Logical Instructions - Binary Representation

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

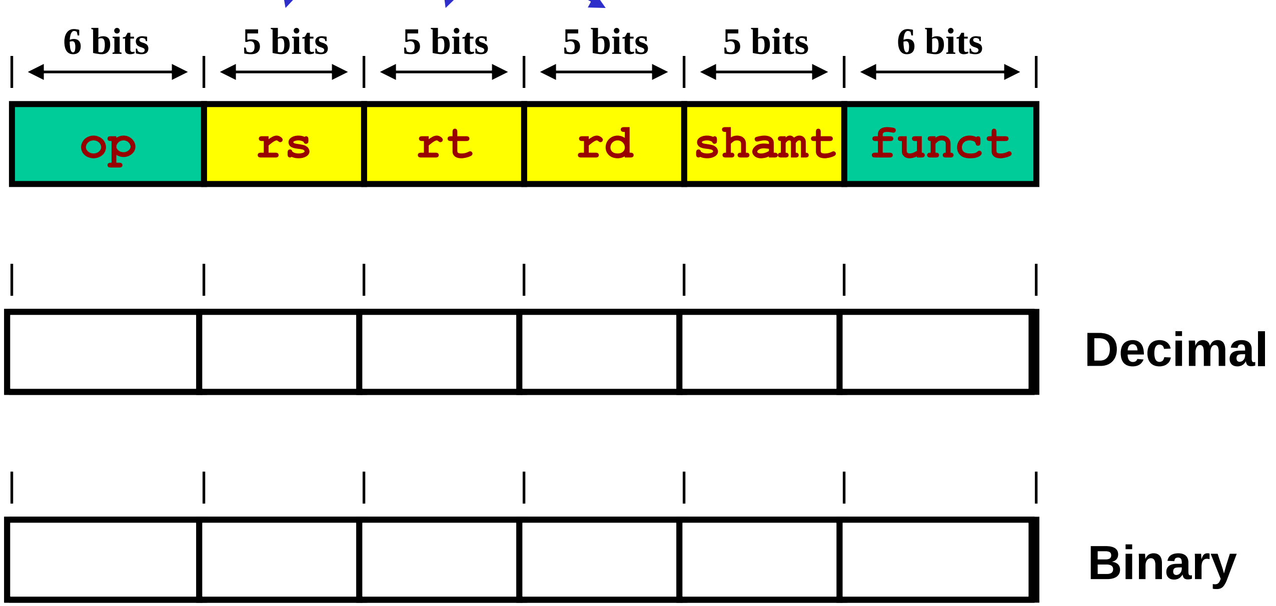
- **Used for arithmetic, logical, shift instructions**
  - **`op`: Basic operation of the instruction (*opcode*)**
  - **`rs`: first register source operand**
  - **`rt`: second register source operand**
  - **`rd`: register destination operand**
  - **`shamt`: shift amount (00000 for now - more about 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

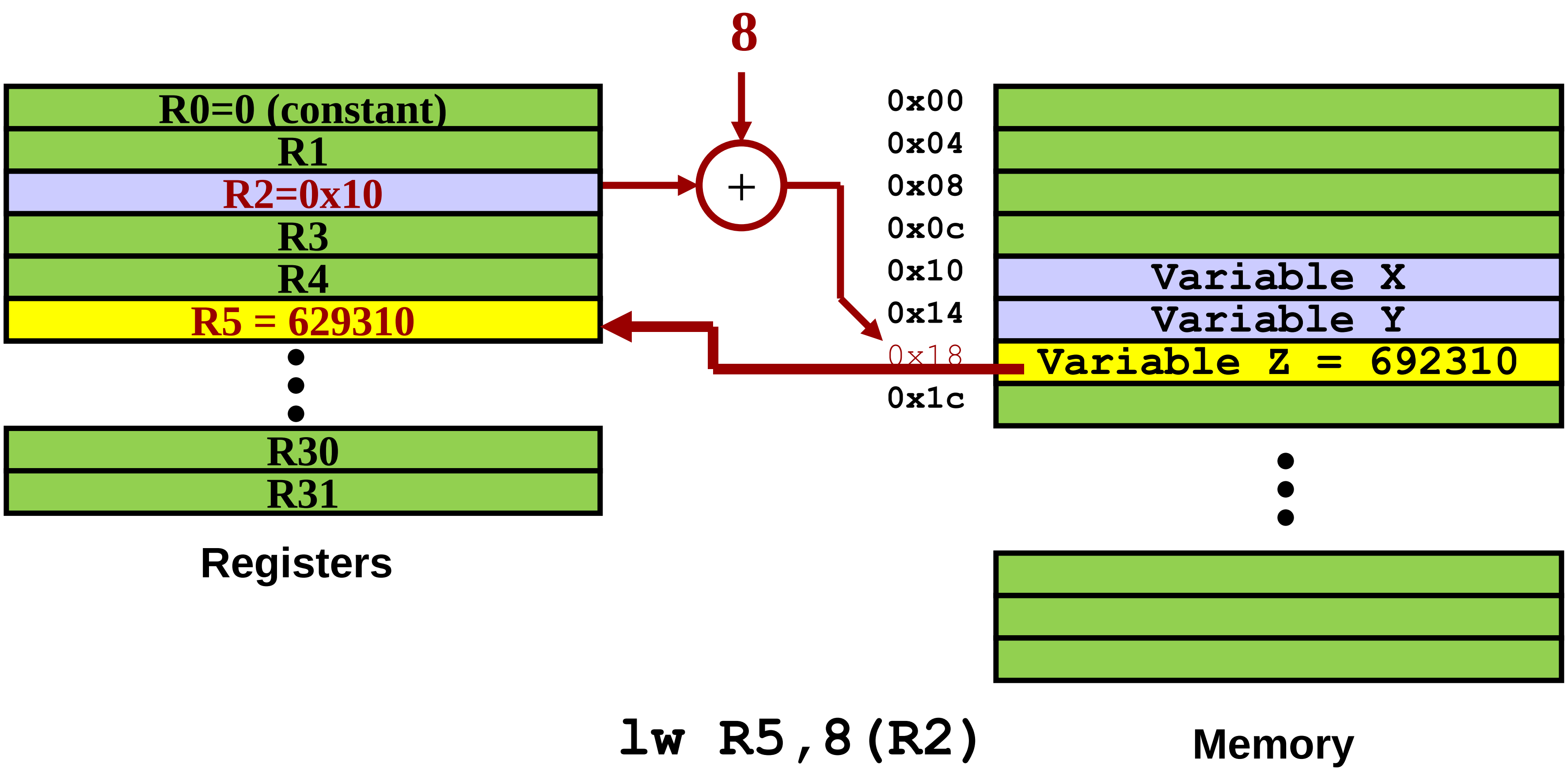
- Transfer data between registers and memory
- Instruction format (assembly)

```
lw $dest, offset($addr)    # load word
sw $src, offset($addr)     # store word
```

- Uses:
  - Accessing a variable in main memory
  - Accessing an array element in main memory

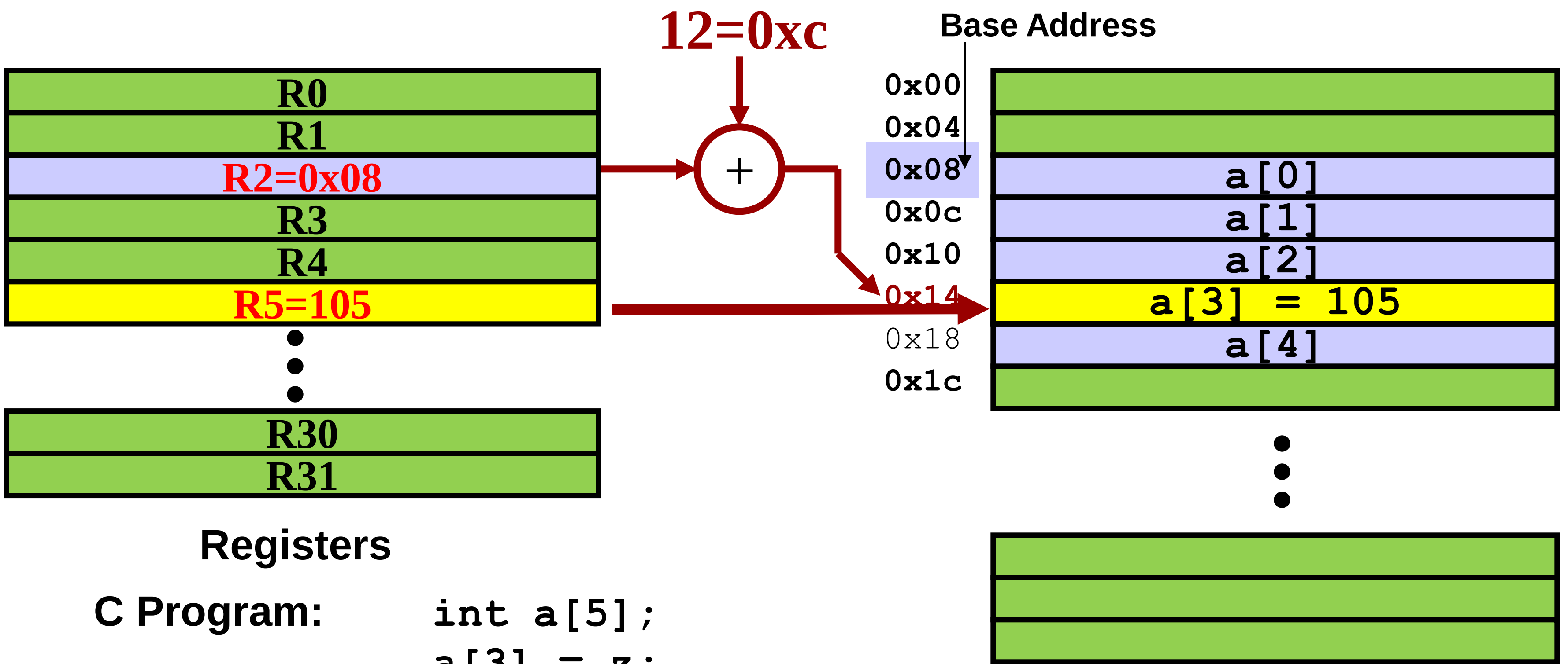
- Notes about `offset`:
  - `offset` must be a constant!
  - `offset` is signed (positive or negative)

Example - Loading a Simple Variable



`lw R5,8(R2)`

Data Transfer Example - Array Variable



Memory

C Program:
```
int a[5];
a[3] = z;
```

scaled offset

Assembly:
```
sw $5,12($2)
```

Data Transfer Instructions - Binary Representation

6 bits	5 bits	5 bits	16 bits
`op`	`rs`	`rt`	`offset`

- **Used for memory access - 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)**

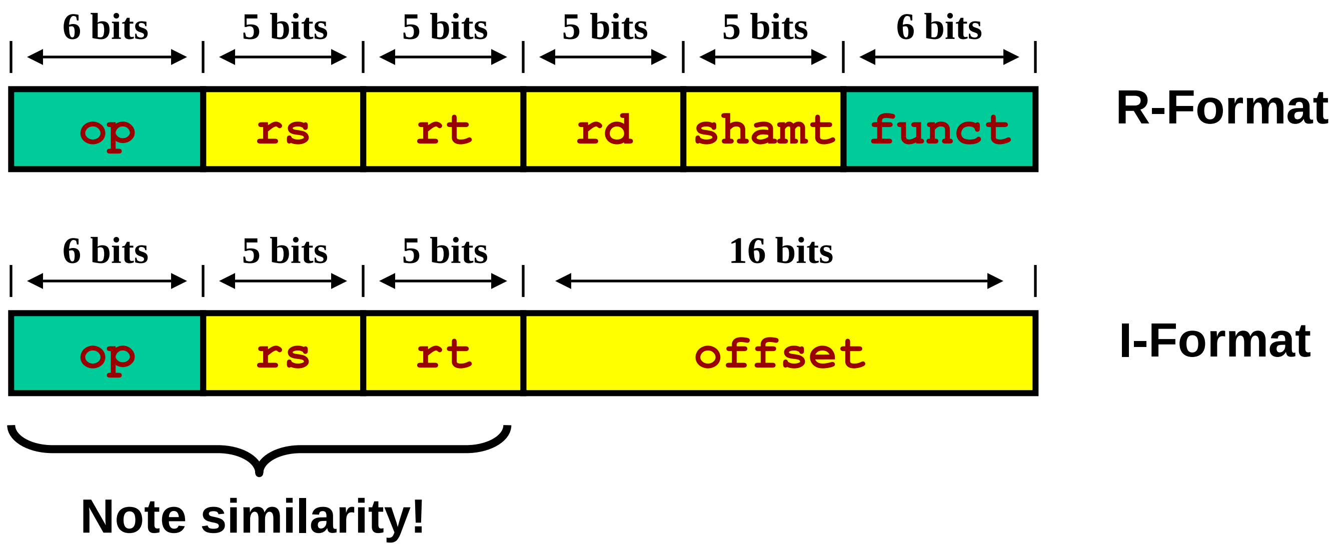
Address

source addr for `sw`
destination addr for `lw`

- **Also called "I-Format" or "I-Type" instructions**

I-Format vs. R-Format Instructions

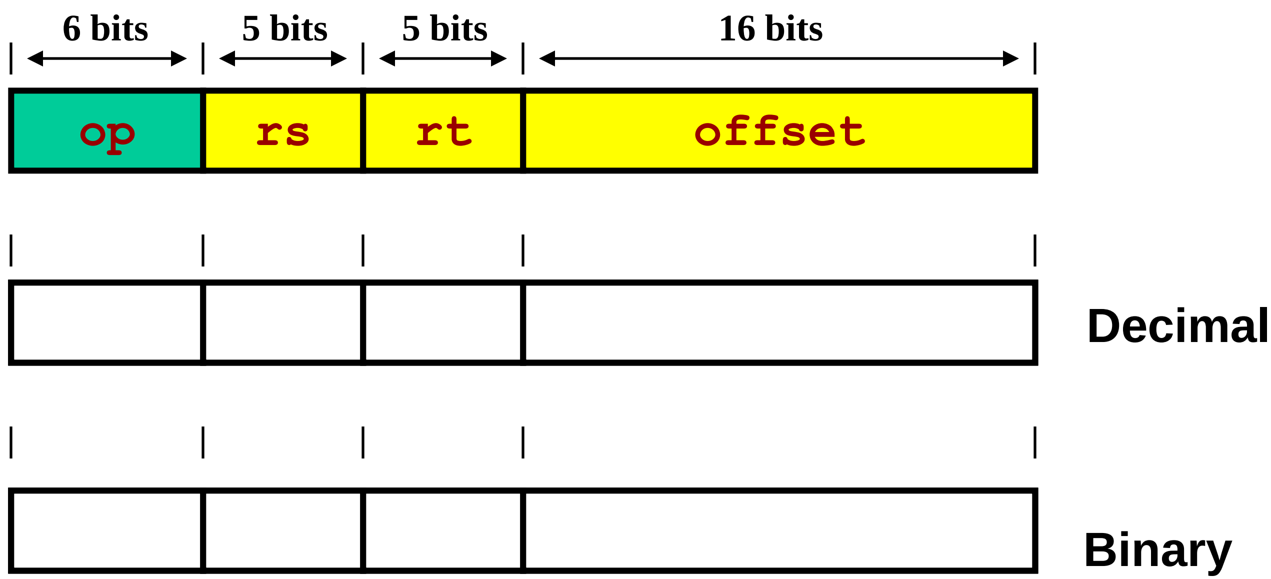
- **Compare with R-Format**



I-Format Example

- **Machine language for**

  `lw $9, 1200($8) == lw $t1, 1200($t0)`



MIPS Conditional Branch Instructions
(stop here)

- **Conditional branches allow <u>decision making</u>**

```
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

6 bits	5 bits	5 bits	16 bits
op	rs	rt	offset

- Branch instructions use I-Format
- `offset` is added to PC when branch is <u>taken</u>

```
beq r0, r1, offset
```

has the effect:

Conversion to word offset

```
if (r0==r1) pc = pc + 4 + (offset << 2)
else pc = pc + 4;
```

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

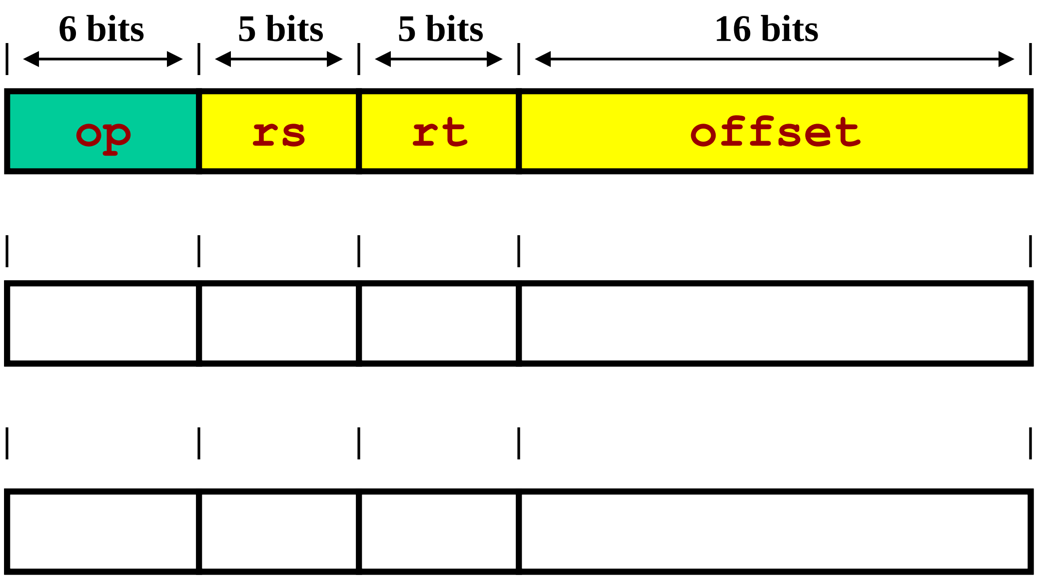
Branch Example

- Machine language for

$19 $20

```
PC            beq $s3, $s4, L1
PC+4          add $s0, $s1, $s2
Target of beq L1: sub $s0, $s0, $s3
```

1-instruction <u>offset</u>



Decimal

Binary

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

- `bne`, `beq` test equality of two numbers
- `slt`, `slti` compare magnitude of signed numbers

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

condition register

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

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

pseudo instruction

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

Signed vs. Unsigned Comparisons

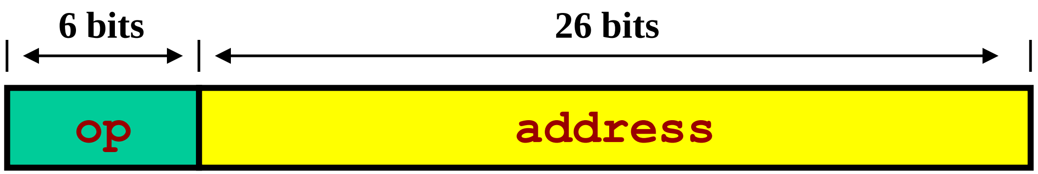
- `slt`, `slti` compare signed numbers
- `sltu`, `sltui` compare unsigned numbers
- Example

```
$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 ⇒ $t0 = 1

sltu $t0, $s0, $s1  # unsigned
+4,294,967,295 > +1 ⇒ $t0 = 0
```

Binary Representation - Jump



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

```
PC = PC[31:28] : (IR[25:0] << 2)
```

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

Conversion to word offset

Jump Example

- **Machine language for**
  `j L5`

```
0x00400020 L5: add $5, $6, $7
0x00400024        .
0x00400028        .
0x0040002C        .

0x03FFFFFF        j  L5
0x04000003        .
0x04000007        .
```

$\underbrace{0000\,0000\,0100\,0000\,0000\,0000\,0010\,0000}$

Assume L5 is at address `0x00400020`

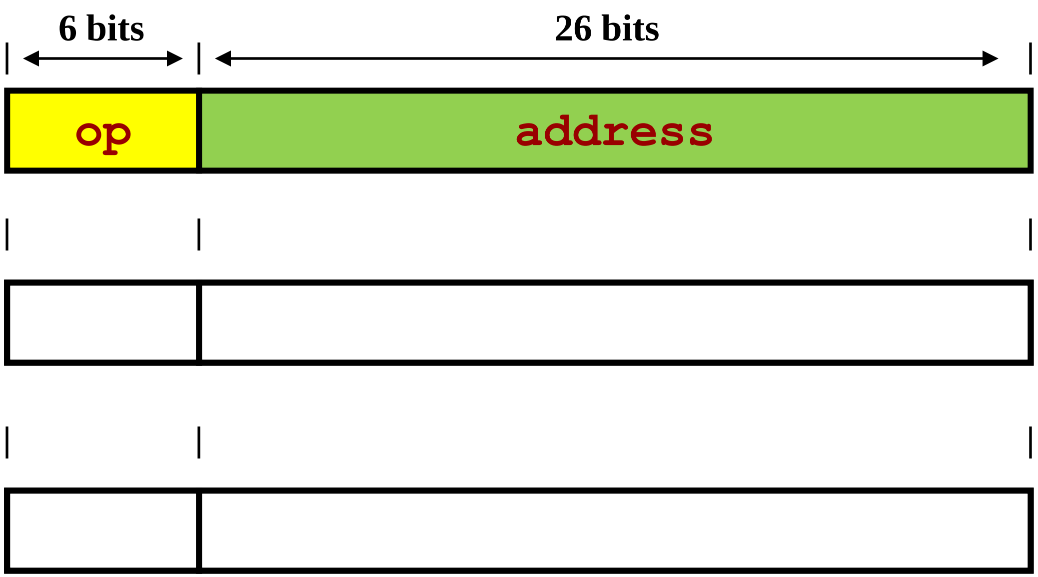
and

`PC <= 0x03FFFFFF`

lower 28 bits

>>2

`0x0100008`



Decimal/Hex

Binary

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):**

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

Arithmetic instructions sign-extend immed. (addi, slti)

Logical instructions don't sign-extend immed. (andi, ori)

- **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

```
$s0  11011111010110100100100011110101
$s1  11110000111100001111000011110000
```

`and $s2,$s0,$s1`

```
$s2  11010000010100000100000011110000
     00000000000000000000000011111100
```

(252_{10})

`ori $s3,s2,252`

```
$s3  11010000010100000100000011111100
```

Logical Operations - Applications

- **Masking - clear, set or test**
  - **Individual bits**
  - **Groups of bits**

x = 01010010

Want to mask 2^{nd} ~ 4^{th} bits

Mask y = 00011100 = 0x1B

z = x & y = 00010000

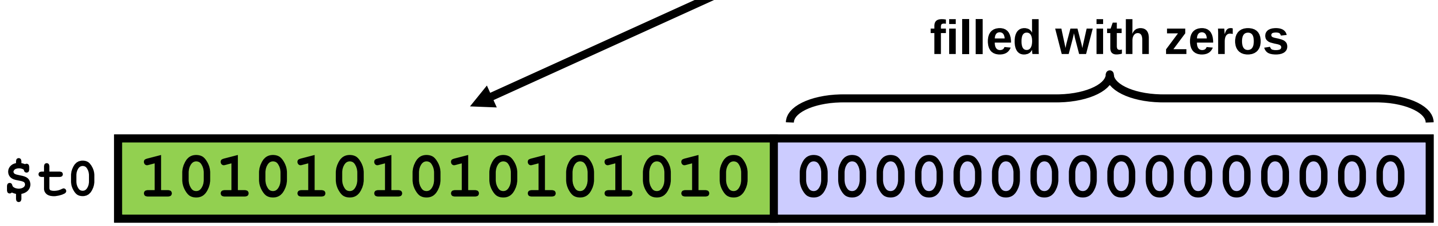
z = z >> 2

z = 00000100

Larger Constants

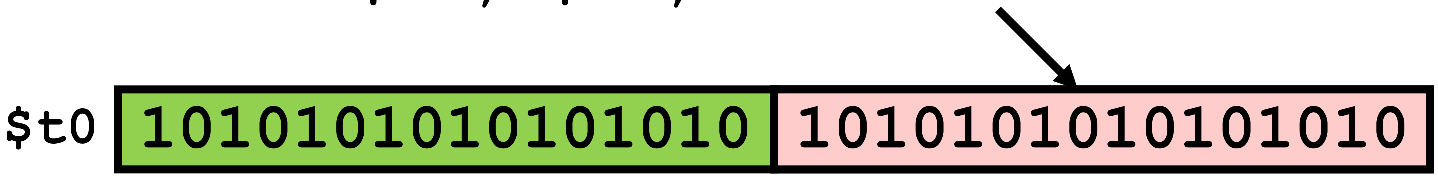
- **Immediate operations provide for 16-bit constants**
- **What about when we need larger constants?**
- **Use "load upper immediate - lui" (I-Format)**

```
lui $t0, 1010101010101010
```



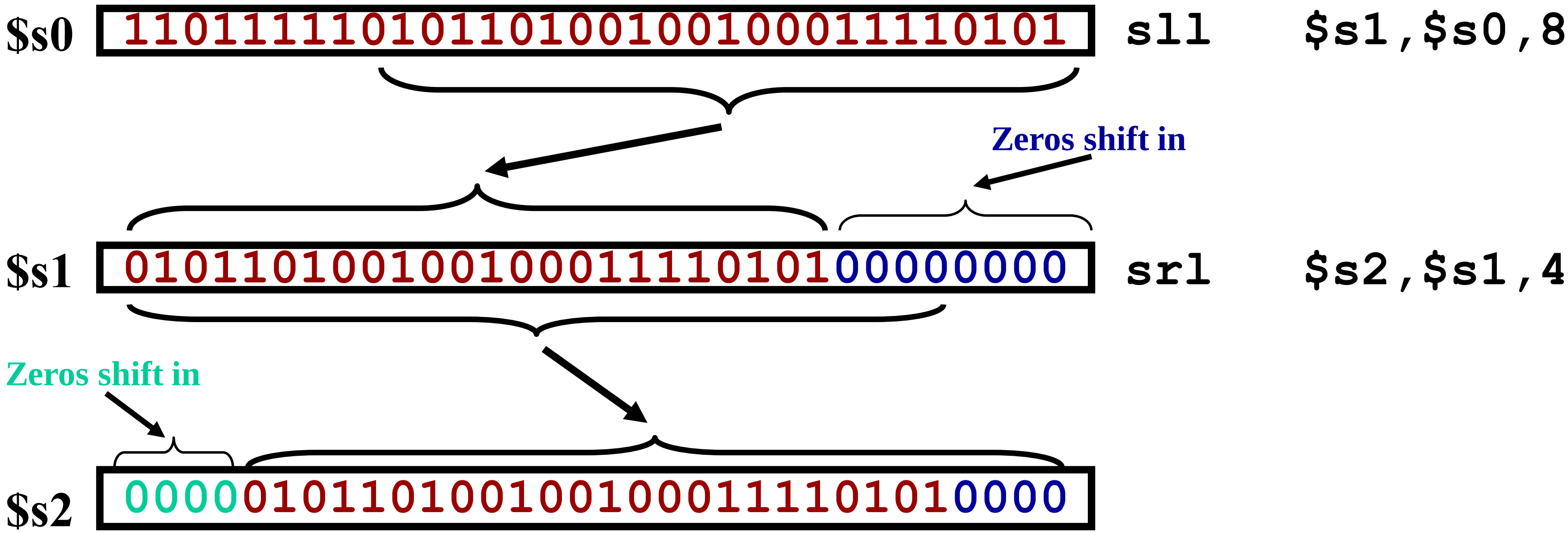
- **Then use `ori` to fill in lower 16 bits:**

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



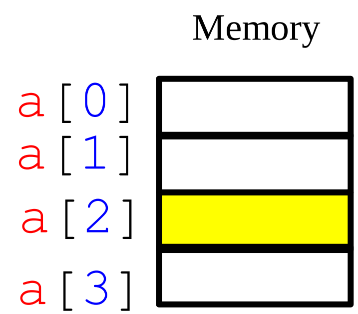
MIPS Shift Instructions

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



Shift Instruction Encodings

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	op	rs	rt	rd	shamt	funct
sll	0	rs	rt	rd	shamt	0
srl	0	rs	rt	rd	shamt	6
		unused				



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

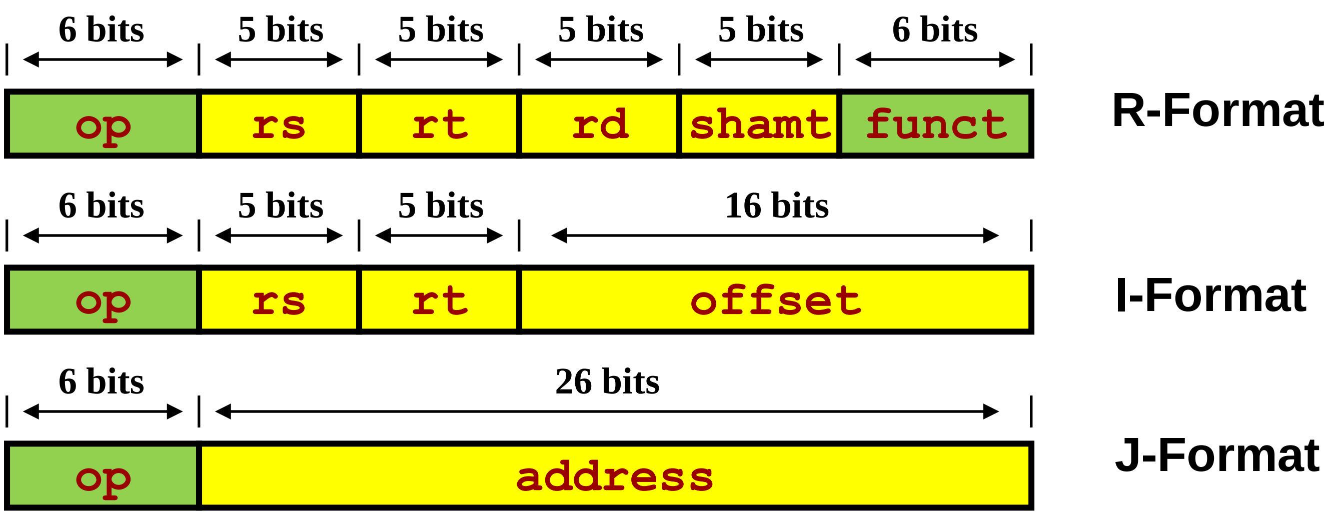
i = 2

lw $t1,i
la $t3, a

```
y = a[i];      sll $t0, $t1, 2      # $t0=$t1*4
               add $t3, $t0, $t3    # t3 = addr of a[i]
               lw  $t3,  0($t3)     # y = a[i]
```

Summary - MIPS Instruction Set

- Three instruction formats
- Similarities in formats ease implementation



Instruction Sets - Overview

- Instruction Set Overview
- MIPS Instruction Set
  - Overview
  - Registers and Memory
  - Review: Unsigned & Signed Binary Numbers
  - Instructions
- Software Concerns
  - Compiling C Constructs
  - Procedures (Subroutines) and Stacks
  - Example: Internet Worms
  - Compiling, Linking, and Loading
  - Example: String Processing / SPIM Demo

코드테이블 보는법

OP(31:26)

Machine code

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0																										

28~26

31~29

	000	001	010	100	101	110	111
000	R-format		jump				
001							
010							
100							
101							
110							
111							

op(31:26)

28–26 / 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format 000 000	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word 100 011	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=000000 (R-format), funct(5:0)

2–0 / 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)	100 000		set l.t.	set l.t. unsigned				
6(110)								
7(111)								