Chapter 2

Instructions: Language of the Computer

Lecture #2

Instruction Set

- The list 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
- MIPS commercialized by MIPS Technologies (www.mips.com). Stands for Microprocessor without Interlocked Pipeline Stages
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data and Appendices B and E

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

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

- All arithmetic operations have this form
- ***Design Principle 1:*** **Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

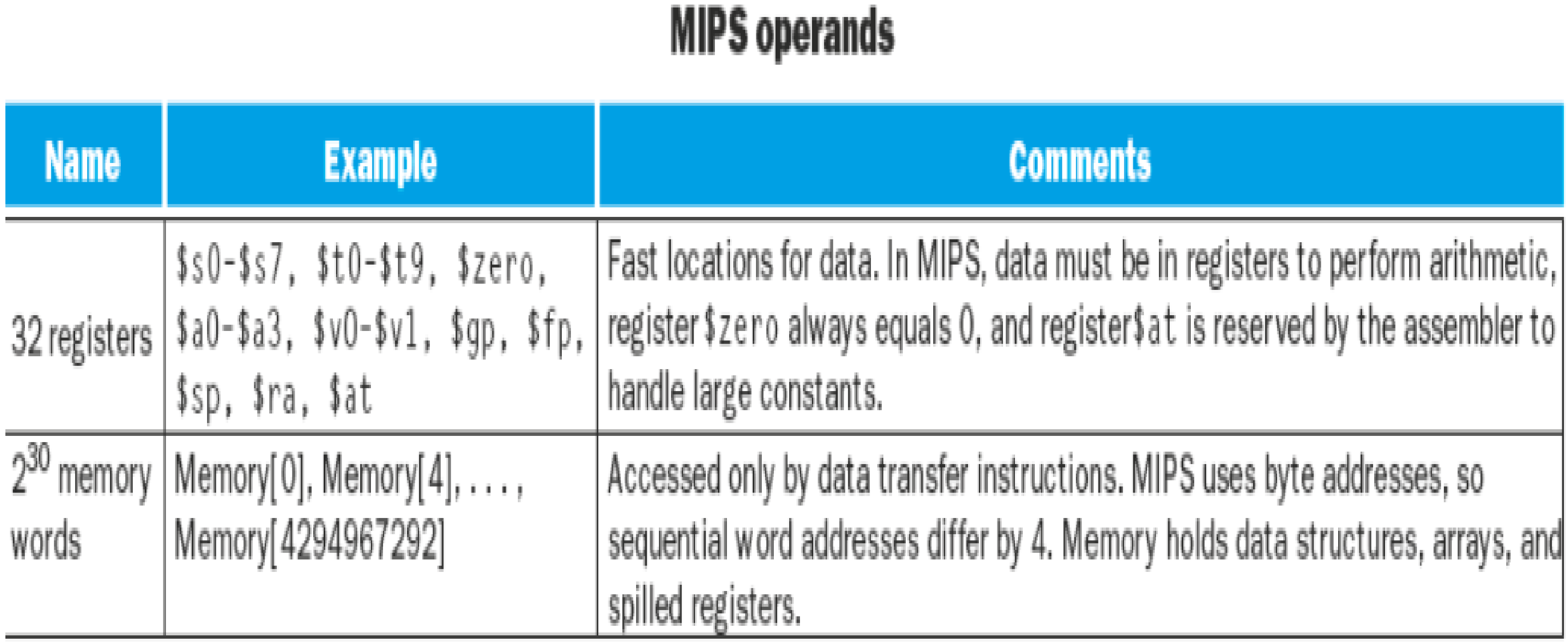
```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- ***Design Principle 2: Smaller is faster***
  - c.f. main memory: millions of locations.
  - A very large number of registers may increase the clock cycle time since it takes electronic signals longer when they must travel farther



MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register`$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Registers

- MIPS convention for naming registers: use two character names (except for $zero) following a dollar sign
- MIPS register **$zero** always maps to zero

Name	Register number (decimal)	Usage	Preserve on call?
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	procedure return values and expression evaluation	no
$a0-$a3	4-7	procedure arguments (parameters)	no
$t0-$t7	8-15	temporary registers	no
$s0-$s7	16-23	general purpose saved registers	yes
$t8-$t9	24-25	more temporary registers	no
$k0-$k1	26-27	reserved for the OS	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	procedure return address	yes

Register Operand Example

- C code:

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

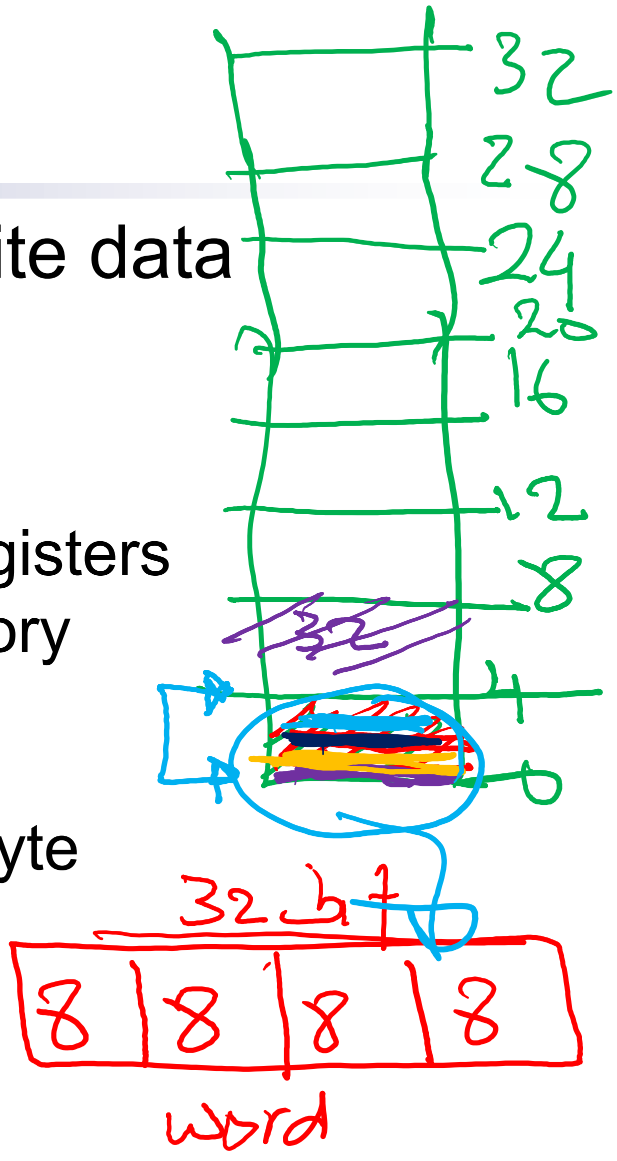
  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

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

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4



Data Transfer Instructions

- **Data transfer instructions:** transfer data between memory and registers
  - **load word** (`lw`): copies a word from memory to a register
  - **store word** (`sw`): copies a word from a register to memory

Base

```
lw $s0, c ($s1)  # Memory [$s1 + c] → $s0
sw $s0, c ($s1)  # $s0 → Memory [$s1 + c]
                 # $s1 is the base register
                 # constant c is the offset
                 # $s1 + c must be divisible by 4
```

base

- **The base register** holds the starting address of the array and the **constant** selects the desired array element (**offset**)
- Traditionally, **the offset** was put in the register (called index register) and the base was supplied as the constant
- Today, the base address of the array is passed in a register since it will not fit in the constant (memories became much larger)

Data Transfer Instructions (cont.)

- Arithmetic operations read and operate on two registers and write one

- Data transfer operations read or write only one register without operating on it

- MIPS memory is only accessed through **loads** and **stores**:
  - This is why MIPS, like all other RISC machines, are called **load/store architectures**
  - MIPS arithmetic operands are registers only, not memory!
  - Registers take less time and have higher throughput than memory
  - Data in registers are both faster to access and simpler to use than memory
  - To achieve highest performance, compilers must use registers efficiently

Memory Operand Example 1

- C code:

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

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

A ($s3)

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

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

offset

base register

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

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

Registers vs. Memory

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

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

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

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

The Constant Zero

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

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

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant

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

0
1
2
3
4
$\downarrow$
7

$0 \to 2^3 - 1$
$0 \to 7$

2s-Complement Signed Integers

n=3

- Given an n-bit number

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

$\boxed{3}$
2
1
0
-1
-2
-3
$\boxed{-4}$

- Range: $\underline{-2^{n-1}}$ to $\underline{+2^{n-1} - 1}$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
- Example: negate +2
  - $+2 = 0000\ 0000 \dots 0010_2$
  - $-2 = 1111\ 1111 \dots 1101_2 + 1$
    $= 1111\ 1111 \dots 1110_2$

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
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => $\underline{\text{0000 0000}}$ 0$\underline{\text{000 0010}}$
  - –2: 1111 1110 => $\underline{\text{1111 1111 1}}$111 1110

Representing Instructions

- Instructions are encoded in binary
  - Called **machine code**
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - design principle 1: **Simplicity favors regularity!**
  - Register numbers
    - $t0 – $t7 are reg's 8 – 15
    - $t8 – $t9 are reg's 24 – 25
    - $s0 – $s7 are reg's 16 – 23

32 bit

$\boxed{\text{Int.}}$

MIPS R-format Instructions

0	s0	s1	dist	x	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
			s2		

- 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)

add s_2, s_0, s_1

dist

1st op

2nd op

R-format Example

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

```
add $t0, $s1, $s2
     rd   rs   rt
```

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

$\boxed{00000010001100100100000000100000_2} = \boxed{02324020_{16}}$

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

MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: **offset** added to **base address** in **rs**
- ***Design Principle 4: Good design demands good compromises***
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Instruction Formats (cont.)

bits	31-26	25-21	20-16	15-11	10-6	5-0
No. of bits	6	5	5	5	5	6
R-format	op	rs	rt	rd	shamt	funct
I-format	op	rs	rt	16- bit immediate/address		
J-format	op	26-bit address				

- **Register format: R-format**
  - Used by arithmetic and logical instructions
- **Immediate format: I-format**
  - Used by data transfer instructions
  - Used by instructions that have immediate operands
  - Used by relative-address branching
- **Jump format: J-format**
  - Used by absolute-jump instructions

Instruction Formats (cont.)

- **Design principle 4: Good design demands good compromises**
  - Compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length

- We have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format

- MIPS keeps all instructions the same length, providing different kinds of instruction formats for different kinds of instructions

- **Hardware complexity can be reduced by keeping formats similar**
  - The first three fields of the R-format and I-format are the same size and have the same names
  - MIPS keep register fields in the same place in each instruction format
  - The length of the fourth field in I-format is equal to the sum of the lengths of the last three fields of R-format
  - There is a similarity between the binary representations of related instructions (e.g., `lw` and `sw`), which simplifies the hardware design

MIPS Instruction Encoding

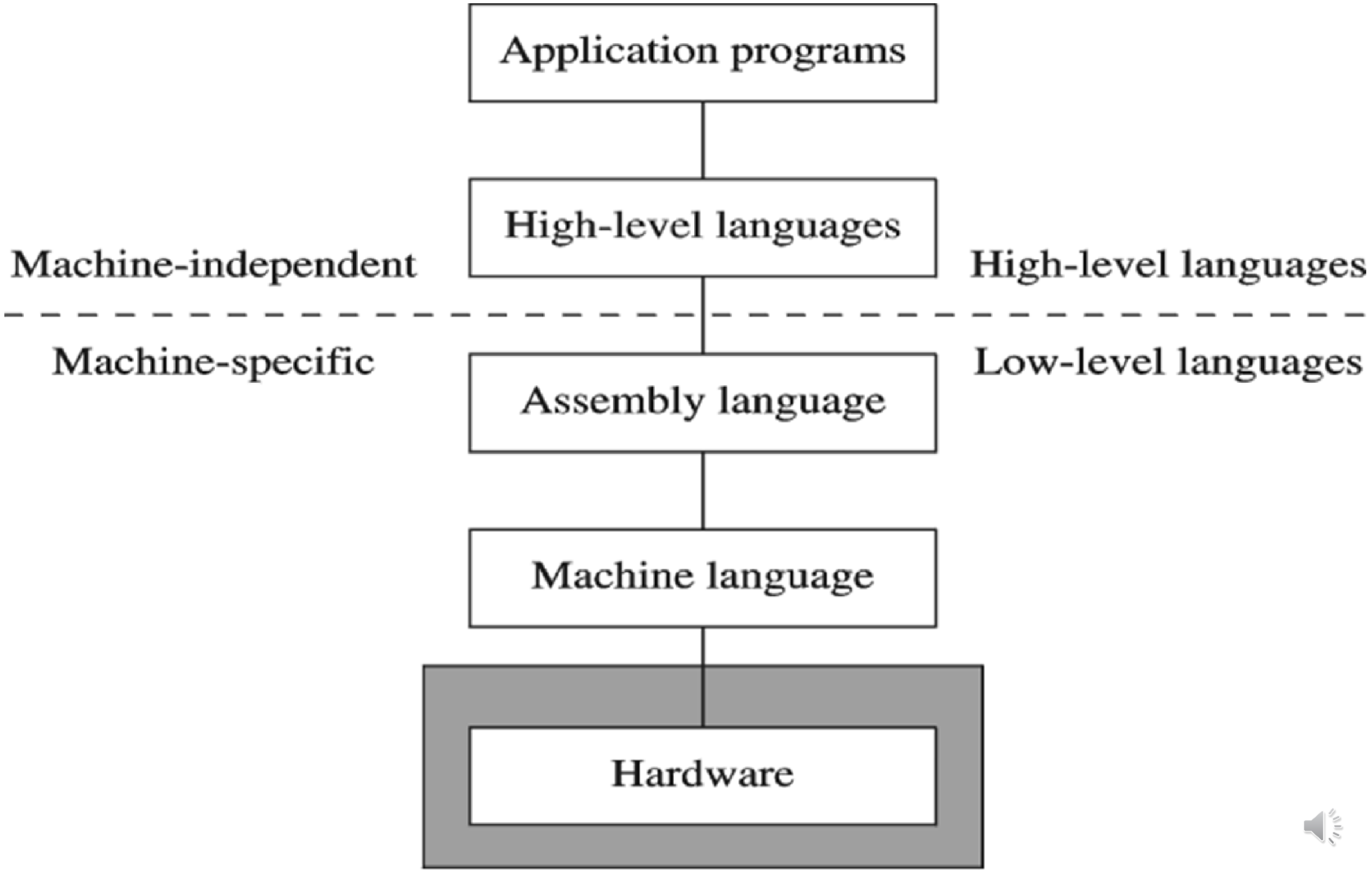
6 5 5 5 5 6

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

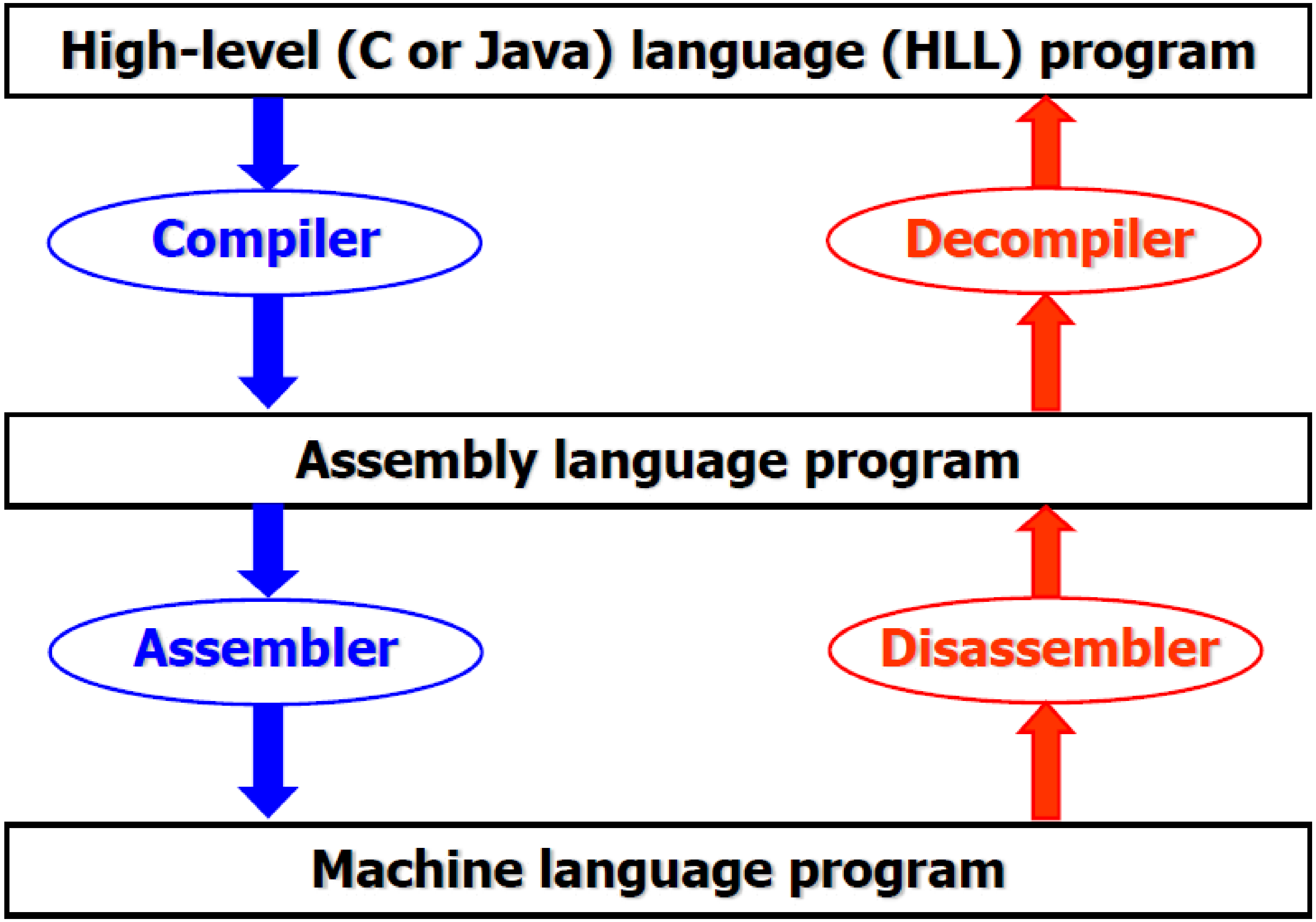
base

In the table above, "reg" means a register number between 0 and 31, "address" means a 16-bit address, and "n.a." (not applicable) means this field does not appear in this format.

A Hierarchy of Languages

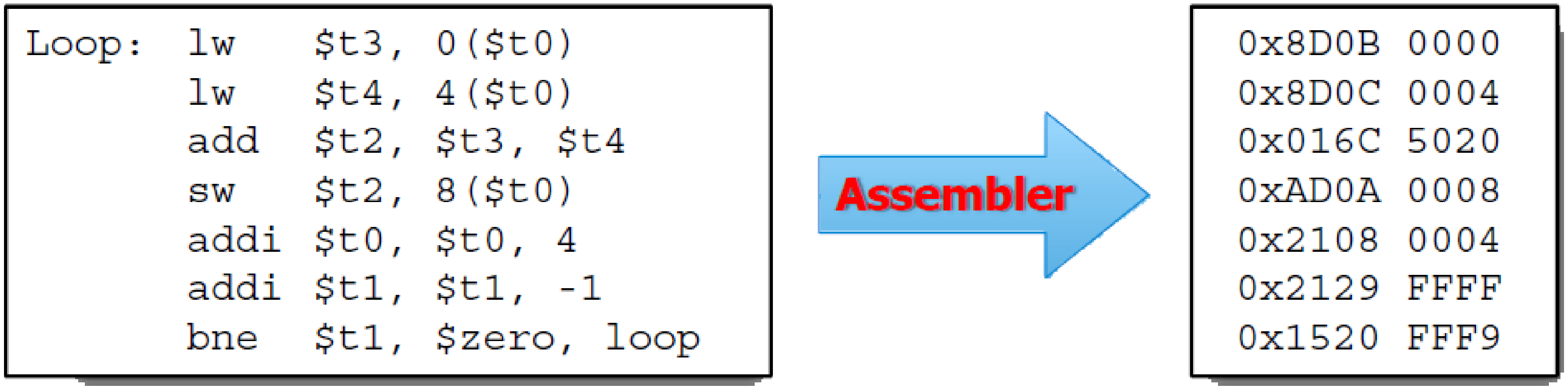


Program Translation Hierarchy


High-level (C or Java) language (HLL) program
Compiler
Decompiler
Assembly language program
Assembler
Disassembler
Machine language program

Assemblers

- The assembler is a computer program, which translates assembly language to machine code, saved in an object file
- Machine code is the interface between software and hardware
- There is a one-to-one correspondence between assembly instructions and machine code



```
Loop:  lw    $t3, 0($t0)
       lw    $t4, 4($t0)
       add   $t2, $t3, $t4
       sw    $t2, 8($t0)
       addi  $t0, $t0, 4
       addi  $t1, $t1, -1
       bne   $t1, $zero, loop
```

```
0x8D0B 0000
0x8D0C 0004
0x016C 5020
0xAD0A 0008
0x2108 0004
0x2129 FFFF
0x1520 FFF9
```

Assembly program
(text file)
source code

Machine code
(binary)
object code

How to Assemble Assembly Instructions

1. Decide the format (R, I, or J) of the instruction
2. Determine the value of each instruction field (component)
3. Convert each field value to binary
4. Put together the full binary code of the instruction
5. Convert the instruction binary code to hexadecimal

Example on Representation

If $t1 has the base of the array A and $s2 corresponds to h, the assignment statement

A[300] = h + A[300]; is compiled into

t1 base

lw $t0,1200($t1) I

add $t0,$s2,$t0 R

sw $t0,1200($t1) I

Op	rs	rt	rd	address/ shamt	funct
35	9 ✓	8		1200	✓
0 ✓	18 ✓	8 ✓	8 ✓	0	32 ✓
43 ✓	9	8 ✓		1200	✓

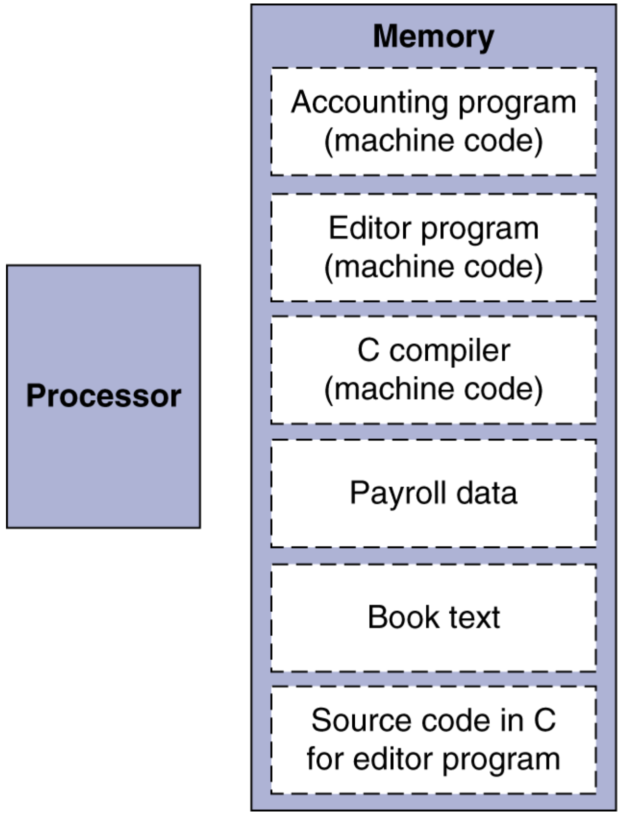
Example on Representation cont.

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

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

Example on Shift Format

sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits

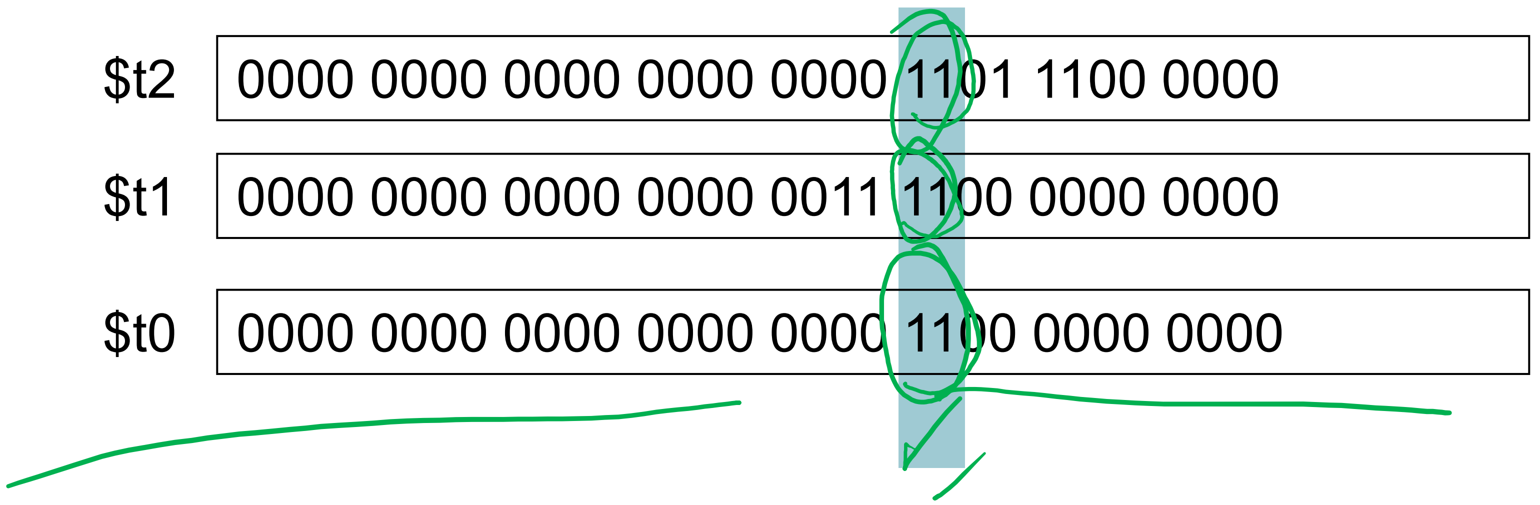
op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

dist

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

Not(A or 0) = (A)

nor

- 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

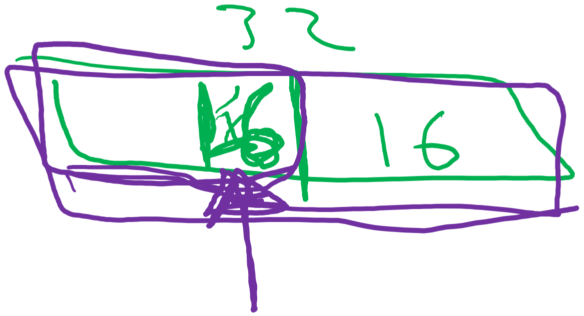
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

32-bit Constants

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

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rt, constant`
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0



`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0011 1101	0000 1001 0000 0000

- To handle larger constants:

– Use the instruction **load upper immediate** (lui) to set the upper 16 bits of a constant in a register

• filling the lower 16 bits with 0s

– Then, use ori to specify the lower 16 bits

- **What is the MIPS assembly code to load the following**

32-bit constant into register $s0?

- 0000 0000 0011 1101 0000 1001 0000 0000

32-bit Constants

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

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rt, constant`
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lui` `$s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0011 1101	0000 1001 0000 0000

- What is the MIPS assembly code to load the following 32-bit constant into register $s0?

61	2304
0000 0000 0011 1101	0000 1001 0000 0000

```
lui  $s0, 61              # 61 is 0000 0000 0011 1101
```

  - The value of register $s0 afterward is

0000 0000 0011 1101	0000 0000 0000 0000

```
ori  $s0, $s0, 2304       # 2304 is 0000 1001 0000 0000
```

  - The value of register $s0 now is

0000 0000 0011 1101	0000 1001 0000 0000