

Unit 2: MIPS Assembly Basics

CSE 220: System Fundamental I

Stony Brook University

Joydeep Mitra

Why Assembly Language?

- Recall computer architecture is the programmer's view of a computer.
- One of those views involves the assembly language (human-readable representation of the computer's native language).
- Like any language, it has words (*instructions*) and a vocabulary (*instruction set*).
- Instructions are encoded in binary.
- But binary is tedious!
 - Encode instructions as easy-to-understand *mnemonics* (assembly).
- Our focus will be on MIPS assembly as this course introduces the fundamentals of MIPS architecture.

MIPS Design Principles

1. Simplicity favors regularity
 - Simple, *consistent* instructions are easier to encode and handle in hardware
2. Make the common case fast
 - MIPS architecture includes only simple, *commonly* used instructions
3. Smaller is faster
 - Smaller, simpler circuits will execute faster than large, complicated ones that implement a complex instruction set
4. Good design demands good compromises
 - We just try to minimize their number

Instructions: Addition

- Java code:
`a = b + c;`
- MIPS assembly code:
`add a, b, c`
- **add** => (mnemonic) indicates operation to perform
- **a: destination operand** (to which the result is written)
- **b, c: source operands** (on which the operation is to be performed)
- In MIPS, **a, b** and **c** are actually CPU registers. More on this soon!

Instructions: Subtraction

- Java code:
`a = b - c;`
- MIPS assembly code:
`sub a, b, c`
- **sub** => mnemonic
- **a**: destination operand
- **b, c**: source operands
- Note **add** and **sub** have similar formats
 1. **Simplicity favors regularity**
 - Consistent instruction format
 - Same number of operands (two sources and one destination)
 - Easier to encode and handle in hardware

Complex Instructions

- More complex code is handled by multiple MIPS instructions

• Java code:

`a = b + c - d;`

MIPS assembly code:

`add t, b, c # t = b + c`

`sub a, t, d # a = t - d`

- The `#` symbol denotes a comment

2. Make the common case fast

- Less common/more complex instructions (e.g., the above instruction) are expressed as multiple simple instructions
- MIPS is a **reduced instruction set computer** (RISC).

What are Operands?

- Operand location: physical location in computer
 - Registers: MIPS has thirty-two 32-bit registers
 - Faster than main memory, but much smaller
 - 3. Smaller is faster: reading data from a small set of registers is faster than from a larger one (simpler circuitry)
 - MIPS is a 32-bit architecture because it operates on 32-bit data
 - Constants (also called **immediates**)
 - Included as part of an instruction



Searching in a stack of fewer books is faster than searching in a stack with many books

MIPS Register Set

Name	Register Number	Usage
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	function return values
\$a0-\$a3	4-7	function arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	function return address

Basic Register Usage

- Registers:
 - \$ before name
 - Example: \$0, “register zero”, “dollar zero”
- Registers are used for specific purposes:
 - \$0 always holds the constant value 0.
 - The **saved registers**, \$s0–\$s7, are used to hold local variable values.
 - The **temporary registers**, \$t0–\$t9, are used to hold intermediate values during a larger computation.
 - We will discuss other registers later.
- Programming in MIPS assembly demands that you follow certain rules (called **conventions**) when using registers.

Using Registers In Instructions

- Let's revisit the **add** instruction
- Java code: MIPS assembly code:
 a = b + c; # \$s0 = a, \$s1 = b, \$s2 = c
 add \$s0, \$s1, \$s2
- General Tip => Comment your code as much as possible in MIPS to avoid confusion

Memory

- With 32 registers you can build only small programs with more than 32 variables
- Memory enhances storage capability
- Memory is larger but also slower than registers
- Efficient programming is about manipulating data in registers and programming
 - Keep commonly used variables in registers
 - When you run out of registers, store less frequent variables in memory
 - Access memory, when those variables are needed for computation

Memory Layout In MIPS

- Each location in MIPS memory can hold 32 bits (**word**)
 - A **word** (32 bits) is the unit of data used natively by a CPU
- Each location has a unique 32-bit address (**word address**)
- If the smallest unit of data we can read from memory is a word, we say that the memory is **word-addressable**
- The MIPS architecture, in contrast, is **byte-addressable**
 - Each **byte** has its own memory address

Word-Addressable Memory Layout

Word Address	Data	
⋮	⋮	⋮
0x0000000C	4 0 F 3 0 7 8 8	Word 3
0x00000008	0 1 E E 2 8 4 2	Word 2
0x00000004	F 2 F 1 A C 0 7	Word 1
0x00000000	A B C D E F 7 8	Word 0

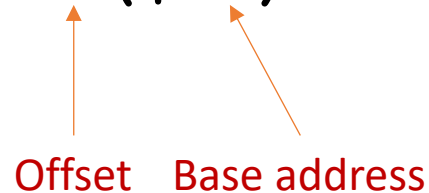
width = 4 bytes

But MIPS is **byte-addressable**

Reading MIPS Memory

- In MIPS, we use the **load** instruction read from memory
- Mnemonic: *load word* (**lw**)
- Format: **lw \$s0, 16(\$t1)**

Offset Base address



- MIPS calculates the effective address of the word we want to read
 - **Effective address = Base address + Offset**
- The value in effective address is stored in **\$s0**
- Any register can be used to hold the base address

Reading MIPS Memory

- Example: suppose we want to read a word of data, i.e., 4 bytes at memory address 8 into **\$s3**

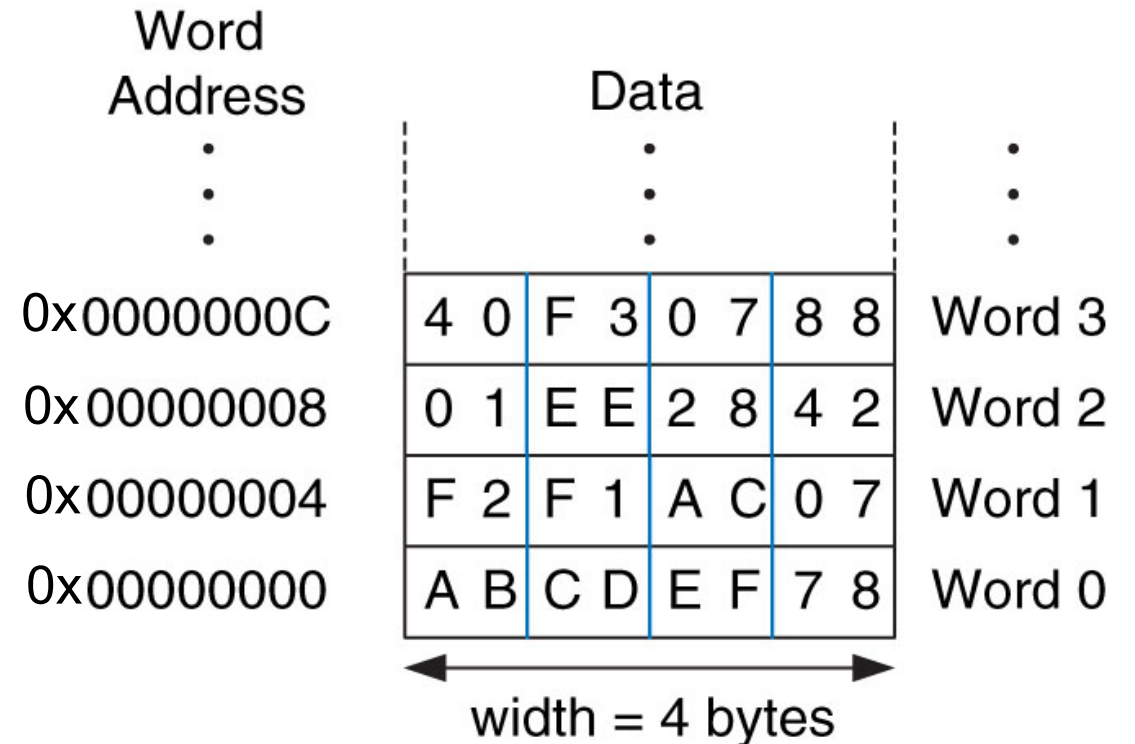
- address = (**\$0** + 8) = 8

- Assembly code:

```
# read memory word 2 into $s3
```

```
lw $s3, 8($0)
```

- Note how each word address is a **multiple of 4**



Writing MIPS Memory

- A memory write is called a **store**
- Mnemonic: *store word* (**sw**)
- Example: suppose we wanted to write (store) the value in register **\$t4** into memory address 8
 - offset for loads and stores can be written in decimal (default) or hexadecimal
 - add the base address (**\$0**) to the offset (0x8)
 - address: (**\$0** + 0x8) = 8

- Assembly code:

```
sw $t4, 0x8($0)      # write the value in $t4 to memory
                      # address 8
```


Loading/Storing Bytes

- Each data byte has a unique address
- *load byte* (**lb**) and *load byte unsigned* (**lbu**)
- When we load a byte what happens to the upper 24 bits in the destination register?

- Fill the upper 24 bits with the 0s

lbu \$s1, 2(\$0)

- Fill the upper 24 bits with the sign bit

lb \$s2, 2(\$0)

Byte Address	3	2	1	0
Data	F7	8C	42	03

- Similarly, we can store bytes using **sb**

\$s3

XX	XX	XX	9B
----	----	----	----

sb \$s3, 3(\$0)

- Replaces 0xF7 with 0x9B in memory byte 3

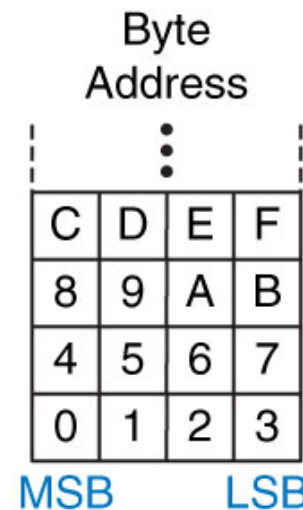
Registers

\$s1	00	00	00	8C	lb<u>u</u> \$s1, 2(\$0)
\$s2	FF	FF	FF	8C	lb \$s2, 2(\$0)

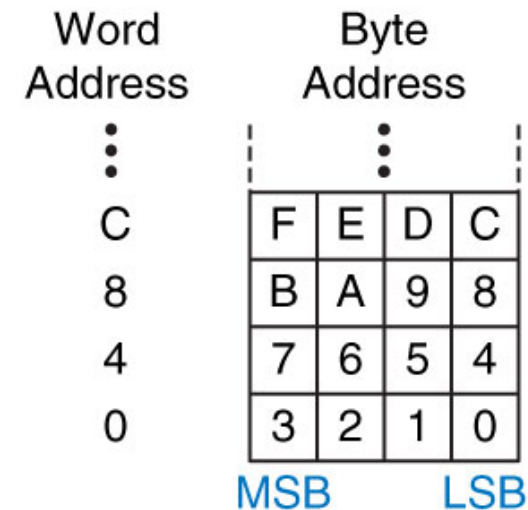
Endianness

- Each 32-bit word has 4 bytes. How should we number the bytes within a word?
- **Little-endian**: byte numbers start at the little (least significant) end
- **Big-endian**: byte numbers start at the big (most significant) end
- **LSB** = least significant byte
- **MSB** = most significant byte
- The word address is the same in either case

Big-Endian



Little-Endian



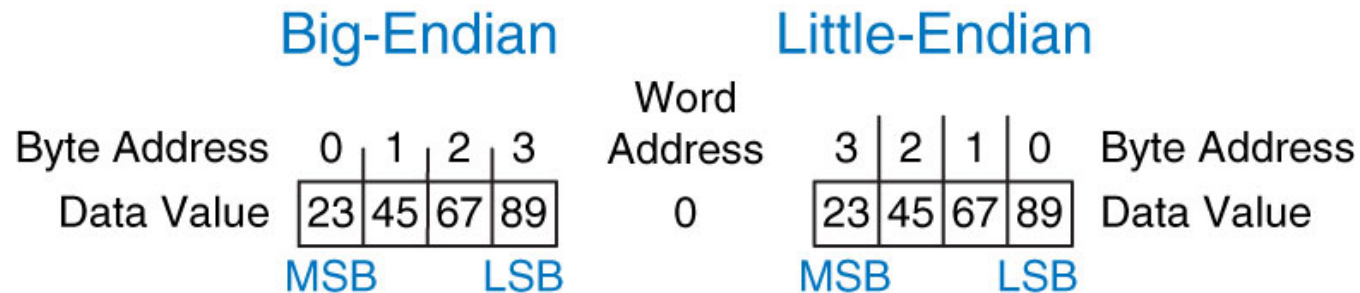
Endianness

- Suppose `$t0` initially contains `0x23456789`
- After following code runs, what value is in `$s0`?

```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

- Big-endian: `0x00000045`
- Little-endian: `0x00000067`



- The MIPS simulator we will use is little-endian

Load/Store Half-Words

- MIPS also supports instructions for loading and storing **half-words** (16 bits)
- **lh** `$s0, 6($t1)`
 - As with **lb**, **lh** sign-extends the loaded value
- **lhu** performs zero-extension of the loaded value
- **sh** stores the *lower* 16 bits of a register into the specified memory address
 - Example: **sh** `$s0, 6($t1)`

Points To Note

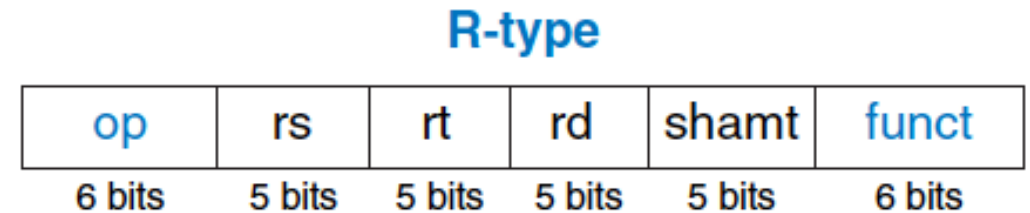
- The address of a *memory word* must be a **multiple of 4**
- Examples:
 - the address of memory word #2 is $2 \times 4 = 8$
 - the address of memory word #10 is $10 \times 4 = 40$ (0x28)
- MIPS is byte-addressed, not word-addressed!
 - You can read load/store individual bytes
 - *Instructions: lb, lbu, sb*
 - If you load/store half-words, memory addresses should be half-word aligned (multiple of 2)
 - *Instructions: lh, lhu, sh*
 - If you load/store words, memory addresses should be word aligned (multiple of 4)
 - *Instructions: lw, sw*

Instruction Formats

- Notice how load/store and arithmetic instructions have different formats.
- But doesn't this violate the design principles?
- Yes! But *good design demands good compromises (Design Principle 4)*.
 - Having only one format is inflexible
 - Trade-off between hardware simplicity and programmer productivity
- MIPS supports 3 instruction formats
 - R-Type : register type
 - I-Type : immediate type
 - J-Type : jump type

R-Type Instruction

- R-type instruction uses 3 operands
 - 2 source operands
 - 1 destination operand
 - All operands are register operands
- Encoding in binary is made of 6 fields
 - Opcode (**op**) and function (**funct**)
=> the operation to perform
 - Source operands **rs** and **rt**
 - Destination operand **rd**
 - Optional field **shamt** to indicate *shift amount*
 - Used for shift operators
 - Set to 0 for non-shift operators



R-Type Instruction

- Examples of R-Type instruction
 - add and sub

Assembly Code

	op	rs	rt	rd	shamt	funct
add \$s0, \$s1, \$s2	0	17	18	16	0	32
sub \$t0, \$t3, \$t5	0	11	13	8	0	34
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

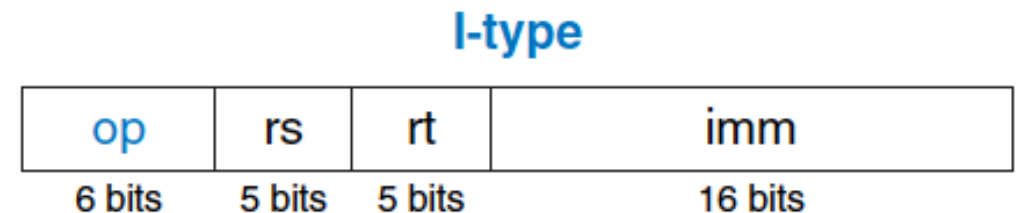
Field Values

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

I-Type Instruction

- I-type instruction uses 3 operands
 - 2 register operands
 - 1 immediate operand
- Encoding in binary is made of 4 fields
 - Opcode (**op**) => operation to perform
 - Source operands **rs** and **imm**
 - **rs** is register operand; **imm** is immediate
 - Register operand **rt**
 - used as destination (e.g., `addi` and `lw`) and source (e.g., `sw`)



I-Type Instruction

- Examples of I-Type
 - addi, lw, sw instructions

Assembly Code

Field Values

Machine Code

	op	rs	rt	imm
addi \$s0, \$s1, 5	8	17	16	5
addi \$t0, \$s3, -12	8	19	8	-12
lw \$t2, 32(\$0)	35	0	10	32
sw \$s1, 4(\$t1)	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

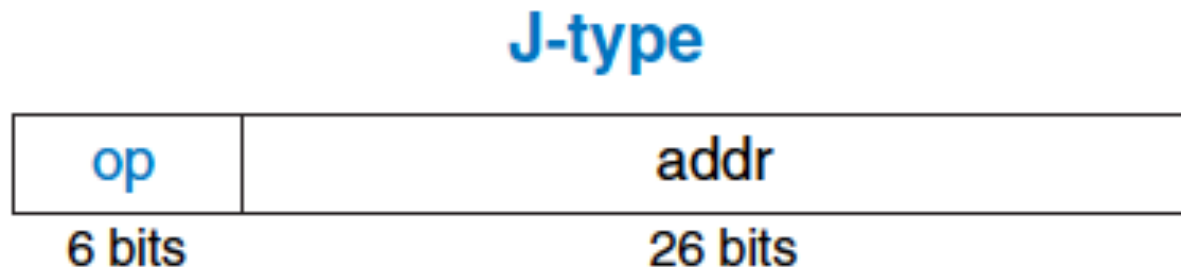
op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

I-Type Immediate

- Note the *Immediate* in an I-Type instruction is 16 bits, but used in 32-bit operations
- What should be in the upper 16 bits then?
 - *Sign extend* the immediate (all 0s for +ve and all 1s for -ve)
- Most MIPS instructions sign extend the immediate (e.g., `addi`, `lw`, `sw`)
- Exceptions include logical instructions (e.g., `andi`, `ori`, `xori`)

J-Type Instruction

- J-type instruction uses one operand.
- The operand indicates the address to jump to (**addr**).
- It is only used for *jump instructions* (e.g., conditionals, loops, and functions).
- The Opcode (**op**) encodes the jump instruction.



Interpret Machine Language Code

- Instructions are encoded in 1s and 0s.
- How does the processor decode instruction formats?
 - Looks at the opcode.
- But all instructions begin with opcode!
- However, R-type instructions have 0 opcode!
 - *If opcode 0 then R-type; otherwise, I-type or J-type.*

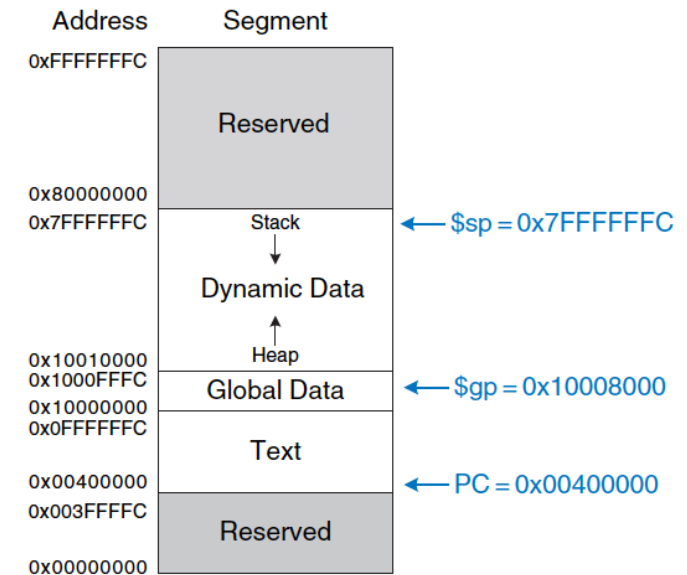
Programming in MIPS

- A MIPS program is a sequence of instructions
- The sequence of instructions can be broken up into blocks of instructions identified by **labels**
 - A **label** is a sequence of alphanumeric characters, underscores and dots. Cannot begin with a number. Ends with a colon.
 - A label denotes the address in memory where the instruction/data is stored

```
label_1:  
    <instr1>  
    <instr2>  
label_2:  
    <instr3>  
    <instr4>
```

The MIPS Memory Map

- Total memory 4 GB (2^{32}); used to store program and data
- *Text Segment*
 - Stores the program code; 256 MB (2^{28})size
- *Global Data Segment*
 - Stores global variables
 - Defined at start-up
 - 64 KB (2^{16}) size
- *Dynamic Data Segment* (2 GB)
 - Holds the *stack* and *heap*
 - *Stack* stores local variables and function-related variable
 - *Heap* stores data allocated at runtime
- *Reserved Segment*
 - Used by the OS; cannot be accessed by programmer



Assembling A MIPS Program

- An assembler takes assembly code and translates to machine code.
- Assembler looks for *assembly directives* to guide translation.
- Assembly directives are not instructions but meant for the assembler. A few key ones:
 - **.text**: beginning of the text segment (code)
 - **.data** : beginning of data segment (global variables)
 - **.asciiz**: declares an ASCII string terminated by NULL
 - **.ascii**: an ASCII string, not terminated by NULL
 - **.word**: allocates space for one or more 32-bit words
 - **.globl**: the name that follows is visible outside the file

MIPS System Calls

- System calls help programmers access OS-level features (e.g., I/O and program termination).
- MIPS has a pre-defined list of system calls.
- Use the instruction **syscall** to make a system call
 - Load service number into register \$v0
 - Load additional arguments into other registers (depends on service type)

MIPS System Calls

Service	Service # (in \$v0)	Arguments	Result
<code>print_int</code>	1	<code>\$a0</code> = integer	
<code>print_float</code>	2	<code>\$f12</code> = float	
<code>print_double</code>	3	<code>\$f12</code> = double	
<code>print_string</code>	4	<code>\$a0</code> = string	
<code>read_int</code>	5		integer (in <code>\$v0</code>)
<code>read_float</code>	6		float (in <code>\$f0</code>)
<code>read_double</code>	7		double (in <code>\$f0</code>)
<code>read_string</code>	8	<code>\$a0</code> = buffer, <code>\$a1</code> = length	
<code>sbrk</code>	9	<code>\$a0</code> = amount	<code>\$v0</code> contains address of allocated memory
<code>exit</code>	10		
<code>print_char</code>	11	<code>\$a0</code> = character	

MIPS Pseudoinstructions

- Remember MIPS is a *reduced instruction set computer (RISC)*?
- Complex instructions are made with combination of simple instructions
- Some complex instructions are very common; programmers have made *pseudoinstructions* for them
- Some useful *pseudoinstructions*
 - The **li** pseudoinstruction loads a *immediate* (32 and 16 bits) into a register
 - **li \$v0, 4** # loads 4 into \$v0; same as lui and then ori
 - The **la** pseudoinstruction loads an address (or label) into a register
 - **la \$a0, str** # loads address of str into \$a0
 - The **move** pseudoinstruction copies the contents of one register to another
 - **move \$t1, \$t2** # same as add \$t1, \$t2, \$0

Hello World Program in MIPS

```
# Hello world program
.data
    Greeting: .asciiz "Hello world!\n"

.text
main:
    # Prints a string on the screen
    li $v0, 4
    la $a0, Greeting
    syscall

    # Terminates the program
    li $v0, 10
    syscall
```

Multiplication And Division Instructions

- Multiplying two 32-bit numbers results in a 64-bit number
- Dividing two 32-bit numbers results in a 32-bit quotient and a 32-bit remainder
- How does MIPS store the results?
 - Two special registers `hi` and `lo`
- `mult $s0, $s1`
 - stores upper 32 bits of $\$s0 * \$s1$ in `hi` and 32 lower bits in `lo`
- `div $s0, $s1`
 - Computes $\$s0 / \$s1$; quotient in `lo` and remainder in `hi`
- Instructions to move values from `lo/hi` special registers
 - `mflo $s2`
 - `mfhi $s3`
- You can use instruction `mul` if product is 32-bit number
 - `mul $s1, $s2, $s3`

MIPS Program To Convert Celsius To Fahrenheit

```
.data
prompt: .asciiz "Enter temperature (Celsius): "
ans1:   .asciiz "The temperature in Fahrenheit is "
endl:   .asciiz "\n"

.text
main:

    la $a0,prompt          #print prompt on terminal
    li $v0,4
    syscall

    li $v0,5                #syscall 5 reads an integer
    syscall
```

MIPS Program To Convert Celsius To Fahrenheit

```
main:
```

```
...
```

```
li $t1, 9
```

```
mul $t0,$v0, $t1      #to convert, multiply by 9,
```

```
li $t1, 5
```

```
div $t0,$t1           #divide by 5, then
```

```
mflo $t0
```

```
addi $t0,$t0,32       #add 32
```

```
la $a0,ans1           #print string before result
```

```
li $v0,4
```

```
syscall
```

MIPS Program To Convert Celsius To Fahrenheit

```
main:
```

```
...
```

```
    move $a0,$t0        #print result
```

```
    li $v0,1
```

```
    syscall
```

```
    la $a0,end1         #system call to print
```

```
    li $v0,4            #out a newline
```

```
    syscall
```

```
    li $v0,10
```

```
    syscall
```


Logical Instructions

- R-type instructions `and`, `or`, `xor`, and `nor`.
- Take two source registers, operate bit-by-bit and store in destination register.
- The `and` instruction can be used for *bit masking* (i.e., make unwanted bits 0).
- The `or` instruction can be used to combine bits from source registers.
- The `xor` instruction is used for bit parity checking.
- The `nor` instruction can be used to substitute NOT
 - $A \text{ NOR } B = \text{NOT } (A \text{ OR } B)$

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111

Result

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111
\$s5	1011	1001	0101	1110	1111	0000	1011	0111
\$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instructions

- I-type instructions `andi`, `ori`, and `xori`
- Same as R-type logical instructions except one of the source is an *immediate* (*zero-extended*)
- There is no `nor`. Why?
 - can be implemented with `ori` and `nor`

Source Values

\$s1

0000	0000	0000	0000	0000	0000	1111	1111
------	------	------	------	------	------	------	------

imm

0000	0000	0000	0000	1111	1010	0011	0100
------	------	------	------	------	------	------	------

←

zero-extended

→

Assembly Code

andi \$s2, \$s1, 0xFA34

\$s2

0000	0000	0000	0000	0000	0000	0011	0100
------	------	------	------	------	------	------	------

ori \$s3, \$s1, 0xFA34

\$s3

0000	0000	0000	0000	1111	1010	1111	1111
------	------	------	------	------	------	------	------

xori \$s4, \$s1, 0xFA34

\$s4

0000	0000	0000	0000	1111	1010	1100	1011
------	------	------	------	------	------	------	------

Result

Shift Instructions

- R-type instruction with
 - a source register to hold value to shift (**rt**)
 - no. of bits to shift (**shamt**)
 - destination register to hold result (**rd**)
- Shift left logical (**sll**)
 - Fill least significant bits with 0s
- Shift right logical (**srl**)
 - Fill most significant bits with 0s
- Shift right arithmetic (**sra**)
 - Fill most significant bits with **sign bit**
- Shift operators can be used for multiplication and division
 - *Shifting left* by N is same as *multiplying* 2^N
 - *Arithmetic right shift* by N is same as *dividing* by 2^N

Source Values							
\$s1	1111	0011	0000	0000	0000	0010	1010 1000
shamt							
							00100
Assembly Code				Result			
sll \$t0, \$s1, 4	\$t0	0011	0000	0000	0000	0010	1010 0000
srl \$s2, \$s1, 4	\$s2	0000	1111	0011	0000	0000	0010 1010
sra \$s3, \$s1, 4	\$s3	1111	1111	0011	0000	0000	0010 1010

Variable Shift Instructions

- R-type instructions with
 - a source register to hold value to shift (**rt**)
 - No. of bits to shift (5 least significant bits of **rs**)
 - destination register to hold result (**rd**)

		Source Values							
		\$s1	1111	0011	0000	0100	0000	0010	1010 1000
		\$s2	0000	0000	0000	0000	0000	0000	0000 1000
Assembly Code		Result							
sllv \$s3, \$s1, \$s2	\$s3	0000	0100	0000	0010	1010	1000	0000	0000
srlv \$s4, \$s1, \$s2	\$s4	0000	0000	1111	0011	0000	0100	0000	0010
srav \$s5, \$s1, \$s2	\$s5	1111	1111	1111	0011	0000	0100	0000	0010

Generating Constants

- Use the `addi` instruction to generate 16-bit constants

High-Level Code	MIPS Assembly Code
<pre>int a = 0x4f3c;</pre>	<pre># \$s0 = a addi \$s0, \$0, 0x4f3c # a = 0x4f3c</pre>

- Use `lui` followed by `ori` instruction to generate 32-bit constants

High-Level Code	MIPS Assembly Code
<pre>int a = 0x6d5e4f3c;</pre>	<pre># \$s0 = a lui \$s0, 0x6d5e # a = 0x6d5e0000 ori \$s0, \$s0, 0x4f3c # a = 0x6d5e4f3c</pre>