Revised: 3/21/2020

Assembly Language I

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Assembly Language
- Registers
- Assembly Statements
- Arithmetic Instructions
  - Addition
  - Subtraction
  - Set If Less Than
- Logical Instructions
  - AND, OR, XOR, NOT
- Shift Instructions
- Immediate Instructions
- Printing Integers
- An Example in MARS

Assembly Language

- If you're in this course, you've previously used a high-level programming language like Python, Java, etc.

- A high-level language is either compiled or interpreted.
  - The source code is converted to either:
  - Executable instructions the processor can understand. (Compiled)
  - Executable instructions that another intermediate piece of software can understand and execute. (Interpreted)

Assembly Language

- Languages like Python and Perl are interpreted.
- C and C++ source code is compiled.
- Some languages, like Java, are a mix of both.
  - Java source code is *compiled* to bytecode (Java's machine code).
  - The compiled bytecode is *interpreted* by the Java Virtual Machine.

Assembly Language

- When programming with a low-level language, there is less abstraction.
  - It's not as easily read/written like with a high-level language.
  - You're basically writing in the language of the processor.

- ***Machine Language***/***Machine Code***: Binary instructions the processor can understand.
- ***Assembly Language***: Text instructions and symbols that map to the binary machine language instructions.

Assembly Language

- Assembly programs are written in plain-text editors like Notepad or WordPad.

- We'll be using the MIPS instruction set for assembly programming.

- We'll be using a simulator in this course, MARS, that will run our assembly programs.
  - **M**IPS **A**ssembler and **R**untime **S**imulator

Registers

- Assembly instructions make use of a limited number of **registers**: small, but fast storage available to the processor.

- In the MIPS architecture, there are 32 general registers
  - Each register can store one 32-bit (4 bytes) word
  - A **word** is the term used by an instruction set that reflects both:
    - The size of a register
    - The size of an instruction
  - The word size in MIPS is 32-bits (4 bytes)

- When performing any instruction, the operation typically involves using the data currently in one or more of these registers.

Registers

- The registers are referred to by their number, preceded by a dollar sign.
  - $0 through $31

- Registers more commonly are referenced by their *name*.
  - Many registers have predefined purposes.
  - The names make their purposes easier to remember.

Registers

Register Number	Name	Used For
$0	$zero	Must always contain 0
$1	$at	Assembler Temporary Value
$2-$3	$v0-$v1	Function Return Values
$4-$7	$a0-$a3	Function Arguments
$8-$15	$t0-$t7	Temporary Values
$16-$23	$s0-$s7	Saved Temporary Values
$24-$25	$t8-$t9	Temporary Values
$26-$27	$k0-$k1	Reserved for OS Kernel
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Function Return Address

Assembly Statements

- Assembly statements have the following format:

```
[label:]   mnemonic   operand, operand, operand   [#comment]
[label:]   mnemonic   operand, operand            [#comment]
[label:]   mnemonic   operand                     [#comment]
```

- The fields in brackets are optional
  - The brackets themselves, [and], are not literally in the instruction
- Each instruction has 1, 2 or 3 operands

Assembly Statements

```
[label:]   mnemonic   operand, operand, operand   [#comment]
[label:]   mnemonic   operand, operand            [#comment]
[label:]   mnemonic   operand                     [#comment]
```

- The label is optional and used to identify a statement or group of statements
  - Will be demonstrated in the next lecture

Assembly Statements

```
[label:]   mnemonic   operand, operand, operand   [#comment]
[label:]   mnemonic   operand, operand            [#comment]
[label:]   mnemonic   operand                     [#comment]
```

- The mnemonic is the operation to be performed, such as **add** (for addition) and **sub** (for subtraction)
  - Mnemonics are easier to remember than their equivalent binary operation codes

Assembly Statements

```
[label:]   mnemonic   operand, operand, operand   [#comment]
[label:]   mnemonic   operand, operand            [#comment]
[label:]   mnemonic   operand                     [#comment]
```

- The instruction's operands are separated by commas.
- Each operand is usually a general register
  - Later lectures will show the use of memory references as operands

Assembly Statements

```
[label:]   mnemonic   operand, operand, operand   [#comment]
[label:]   mnemonic   operand, operand             [#comment]
[label:]   mnemonic   operand                      [#comment]
```

- The end of a statement may include an optional comment.
  - Comments always begin with the # character

- Comments can exist on their own lines.
- There are **no** multi-line comments.
  - Unlike Java, for example, which uses /* and */ for multi-line comments

Addition

- The first instruction we will see is the **add** mnemonic, which adds the values of two registers and stores the result in a third register.

```
add  $rd, $rs, $rt
```

  - $rd = Destination Register
  - $rs = Source Register 1
  - $rt = Source Register 2

- This is an example of a **Register Format** Instruction (or **R-format**)
  - More details on this in a later lecture

Addition

- The following instruction:

```
add  $t0, $t1, $t2
```

will add the values of registers $t1 and $t2 together and store the result in register $t0

- One way to think of this instruction is: **`$t0 = $t1 + $t2`**
  - However, this is not valid assembly code

Addition

- Convert the following to a valid assembly instruction:

`$t1 = $t2 + $t0`

Addition

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0
```

```
add $t1, $t2, $t0
```

Addition

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 + $t3
```

- Hint: It's not `add $t1, $t2, $t0, $t3`

Addition

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 + $t3
```

- Let's assume the following values are in these registers
  - $t0 = 5
  - $t1 = 0
  - $t2 = 3
  - $t3 = 7

Addition

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 + $t3
```

- We first need to sum $t2 (3) and $t0 (5)

```
add $t1, $t2, $t0
```

- This assigns 8 to $t1

```
$t0 = 5
$t1 = 8
$t2 = 3
$t3 = 7
```

Addition

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 + $t3
```

- Next, we can add $t3 (7) to $t1 (8), and assign the result to $t1

```
add $t1, $t1, $t3
```

- This assigns 15 to $t1

```
$t0 = 5
$t1 = 15
$t2 = 3
$t3 = 7
```

Addition

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 + $t3
```

```
add $t1, $t2, $t0
add $t1, $t1, $t3
```

Subtraction

- Similar in format to the **add** mnemonic, the **sub** mnemonic finds the difference between the values of two registers and stores the result in a third register.

```
sub  $rd, $rs, $rt
```

  - $rd = Destination Register
  - $rs = Source Register 1
  - $rt = Source Register 2

Subtraction

- The following instruction:

```
sub  $t0, $t1, $t2
```

will subtract the value of registers $t2 from $t1 and store the result in register $t0

- One way to think of this instruction is: **`$t0 = $t1 - $t2`**
  - However, this is not valid assembly code

Subtraction

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 - $t0
```

Subtraction

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 - $t0
```

```
sub $t1, $t2, $t0
```

Subtraction

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 - $t3
```

Subtraction

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + $t0 - $t3
```

```
add $t1, $t2, $t0
sub $t1, $t1, $t3
```

Subtraction

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + ($t0 - $t3)
```

Subtraction

- Convert the following to a valid assembly instruction:

```
$t1 = $t2 + ($t0 - $t3)
```

```
sub $t1, $t0, $t3
add $t1, $t1, $t2
```

Set If Less Than

- The **`slt`** mnemonic compares the numeric values of two registers and stores the either a 1 or 0 in a third register.

```
slt  $rd, $rs, $rt
```

  - $rd = Destination Register
  - $rs = Source Register 1
  - $rt = Source Register 2

- The slt mnemonic stores a 1 in $rd if $rs < $rt
- The slt mnemonic stores a 0 in $rd if $rs >= $rt

Set If Less Than

- For the following instruction:

```
slt  $t0, $t1, $t2
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 5
  - $t2 = 13

- **The slt mnemonic stores a 1 in $t0 if $t1 < $t2**
- The slt mnemonic stores a 0 in $t0 if $t1 >= $t2

Set If Less Than

- For the following instruction:

```
slt  $t0, $t1, $t2
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 13
  - $t2 = 11

- The slt mnemonic stores a 1 in $t0 if $t1 < $t2
- **The slt mnemonic stores a 0 in $t0 if $t1 >= $t2**

Logical Instructions

- Logical instructions return a true or false value.
- The operands of a logical instruction must be either true or false.
- For the rest of the course:
  - 1 means true
  - 0 means false

Logical Instructions

- An **AND** logical operation results in true (1) only when both operands are themselves true (1)

x	y	$x\ AND\ y$
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

- May be expressed as:
  - $x\ AND\ y$
  - $x \wedge y$
  - $x \cdot y$

Logical Instructions

- The **and** mnemonic performs an and operation on two registers, and assigns the result in a third.

```
and  $rd, $rs, $rt
```

  - $rd = Destination Register
  - $rs = Source Register 1
  - $rt = Source Register 2

Logical Instructions

- For the following instruction:

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

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1
  - $t2 = 0

- This instruction will assign 0 to $t0

x	y	$x\ AND\ y$
0	0	0
0	1	0
1	**0**	**0**
1	1	1

Logical Instructions

- For the following instruction:

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

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1
  - $t2 = 1

- This instruction will assign 1 to $t0

x	y	$x\ AND\ y$
0	0	0
0	1	0
1	0	0
1	**1**	**1**

Logical Instructions

- Logical instructions perform *bitwise* operations.

- Let's assume the following values are in these registers
  - $t0 = 0000000
  - $t1 = 1001001
  - $t2 = 1100111

- We perform the following instruction:

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

Logical Instructions

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

$t0 = 0000000
$t1 = 1001001
$t2 = 1100111

1001001
AND 1100111
1000001

$t0 = **1000001**
$t1 = 1001001
$t2 = 1100111

- Since registers are 32-bits in length, each logical instruction will perform 32 bitwise operations

Logical Instructions

- An **OR** logical operation results in true (1) only when at least one (or both) of the operands are themselves true (1)

x	y	$x\ OR\ y$
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

- May be expressed as:
  - $x\ OR\ y$
  - $x \vee y$
  - $x + y$

Logical Instructions

- The **`or`** mnemonic performs an or operation on two registers, and assigns the result in a third.

```
or   $rd, $rs, $rt
```

  - $rd = Destination Register
  - $rs = Source Register 1
  - $rt = Source Register 2

Logical Instructions

- For the following instruction:

```
or   $t0, $t1, $t2
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1
  - $t2 = 0

- This instruction will assign 1 to $t0

x	y	$x\ OR\ y$
0	0	0
0	1	1
1	**0**	**1**
1	1	1

Logical Instructions

- For the following instruction:

```
or   $t0, $t1, $t2
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1
  - $t2 = 1

- This instruction will assign 1 to $t0

x	y	$x\ OR\ y$
0	0	0
0	1	1
1	0	1
1	**1**	**1**

Logical Instructions

```
or   $t0, $t1, $t2
```

$t0 = 0000000
$t1 = 1001001
$t2 = 1100111

1001001
OR 1100111
1101111

$t0 = **1101111**
$t1 = 1001001
$t2 = 1100111

Logical Instructions

- An **XOR** (e**x**clusive **or**, "*x or*"; "*zor*") logical operation results in true (1) only when **one** of the operands are true (1)

x	y	$x\ XOR\ y$
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

- May be expressed as:
  - $x\ XOR\ y$
  - $x \oplus y$

Logical Instructions

- The **`xor`** mnemonic performs an xor operation on two registers, and assigns the result in a third.

```
xor  $rd, $rs, $rt
```

  - $rd = Destination Register
  - $rs = Source Register 1
  - $rt = Source Register 2

Logical Instructions

- For the following instruction:

```
xor  $t0, $t1, $t2
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1
  - $t2 = 0

- This instruction will assign 1 to $t0

x	y	$x\ XOR\ y$
0	0	0
0	1	1
1	**0**	**1**
1	1	0

Logical Instructions

- For the following instruction:

```
xor  $t0, $t1, $t2
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1
  - $t2 = 1

- This instruction will assign 0 to $t0

x	y	$x\ XOR\ y$
0	0	0
0	1	1
1	0	1
1	**1**	**0**

Logical Instructions

```
xor  $t0, $t1, $t2
```

$t0 = 0000000
$t1 = 1001001
$t2 = 1100111

1001001
XOR 1100111
0101110

$t0 = **0101110**
$t1 = 1001001
$t2 = 1100111

Logical Instructions

- A **NOT** logical operation negates/inverts a single operand

x	$NOT\ x$
0	**1**
1	**0**

- May be expressed as:
  - $NOT\ x$
  - $\overline{x}$
  - $\neg x$
  - x'
  - $\sim x$

Logical Instructions

- The **`not`** mnemonic performs an not operation on one register, and assigns the result in a second.

```
not  $rd, $rs
```

  - $rd = Destination Register
  - $rs = Source Register

Logical Instructions

- For the following instruction:

```
not  $t0, $t1
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 1

x	$NOT\ x$
0	1
1	**0**

- This instruction will assign 0 to $t0

Logical Instructions

- For the following instruction:

```
not  $t0, $t1
```

- Let's assume the following values are in these registers
  - $t0 = 0
  - $t1 = 0

x	$NOT\ x$
0	**1**
1	0

- This instruction will assign 1 to $t0

Logical Instructions

```
not  $t0, $t1
```

$t0 = 0000000
$t1 = 1001001

NOT 1001001
0110110

$t0 = **0110110**
$t1 = 1001001

Shift Instructions

- Shifting moves the bits of a word to the left or right
- Shift instructions have a destination register, source register, and the shift amount
  - The shift instruction does not alter the data in the source register
- The first type of shifting process we will see is *logical shift*

Shift Instructions

- The **`sll`** mnemonic performs a logical left shift operation on one register, and assigns the result in a second.

```
sll  $rd, $rs, shamt
```

  - $rd = Destination Register
  - $rs = Source Register
  - shamt = Shift Amount

Shift Instructions

- For the following instruction:

```
sll  $t0, $t1, 2
```

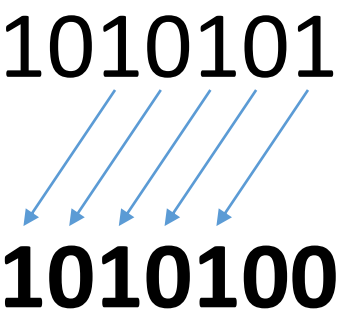
- Let's assume the following values are in these registers
  - $t0 = 0000000
  - $t1 = 1010101

- This operation will shift the bits in $t1 to the left by two bits
  - Inserts 0's to the right

Shift Instructions

```
sll  $t0, $t1, 2
```

$t0 = 0000000
$t1 = 1010101



$t0 = **1010100**
$t1 = 1010101

Shift Instructions

- The **`srl`** mnemonic performs a logical right shift operation on one register, and assigns the result in a second.

```
srl  $rd, $rs, shamt
```

  - $rd = Destination Register
  - $rs = Source Register
  - shamt = Shift Amount

Shift Instructions

- For the following instruction:

```
srl  $t0, $t1, 2
```

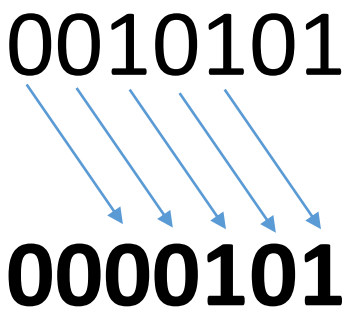
- Let's assume the following values are in these registers
  - $t0 = 0000000
  - $t1 = 0010101

- This operation will shift the bits in $t1 to the right by two bits
  - Inserts 0's to the left

Shift Instructions

```
srl  $t0, $t1, 2
```

$t0 = 0000000
$t1 = 0010101



$t0 = **0000101**
$t1 = 0010101

Shift Instructions

- Right shifting can be either
  - *Logical Shift* (The previous example)
  - *Arithmetic Shift*

- Arithmetic Shift ensures a negative number (begins with 1) remains negative.
  - It fills the left bits with 1's instead of 0's for negative numbers

- The **`sra`** mnemonic is used for arithmetic right shift.
  - There is no arithmetic left shift

Shift Instructions

- For the following instruction:

```
sra  $t0, $t1, 2
```

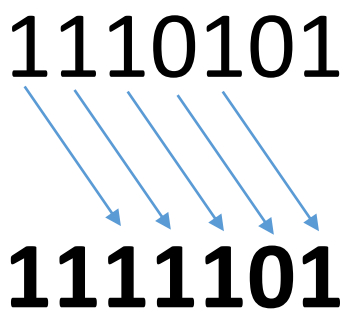
- Let's assume the following values are in these registers
  - $t0 = 0000000
  - $t1 = 1110101

- This operation will shift the bits in $t1 to the right by two bits
  - Inserts 1's to the left since the left-most bit is a 1
  - If the number in $t1 started with 0, 0's would be inserted

Shift Instructions

```
sra  $t0, $t1, 2
```

$t0 = 0000000
$t1 = 1110101



$t0 = **1111101**
$t1 = 1110101

Shift Instructions

- A useful application of left shifting is to perform multiplication by powers of 2

- $00001100_2 = 12_{10}$
  - Left shifting by **2** yields $\mathbf{00110000}_2 = 48_{10}$
  - $12_{10} \times 2^{\mathbf{2}}{}_{10}$

- $00001111_2 = 15_{10}$
  - Left shifting by **3** yields $\mathbf{01111000}_2 = 120_{10}$
  - $15_{10} \times 2^{\mathbf{3}}{}_{10}$

Shift Instructions

- A useful application of right shifting is to perform division by powers of 2

- $01001100_2 = 76_{10}$
  - Right shifting by **2** yields $\mathbf{00010011}_2 = 19_{10}$
  - $76_{10} \div 2^{\mathbf{2}}{}_{10}$

- $11110000_2 = 240_{10}$
  - Right shifting by **3** yields $\mathbf{00011110}_2 = 30_{10}$
  - $240_{10} \div 2^{\mathbf{3}}{}_{10}$

Immediate Instructions

- The R-Format instructions seen so far lack the ability to:
  - Assign a constant value (other than $zero) to a register
  - Perform arithmetic instructions with a constant value
  - Perform logical instructions with a constant value
  - Transfer data to registers from main memory and vice versa

- **Immediate Format** Instructions (**I-Format**) allow the use of immediate/constants
  - More details on this in a later lecture
- The maximum size of an immediate constant is 16-bits (half-word)
  - -32,768 to 32,767

Immediate Instructions

- The first immediate instruction we will see is the **`addi`** mnemonic, which adds the value of a register with a constant and stores the result in a destination register.

```
addi   $rd, $rs, constant
```

  - $rd = Destination Register
  - $rs = Source Register
  - constant = A literal/constant value

Immediate Instructions

- The following instruction:

```
addi   $t0, $t1, 5
```

will add the values of register $t1 and the number 5 together and store the result in register $t0

- One way to think of this instruction is: **`$t0 = $t1 + 5`**
  - However, this is not valid assembly code

Immediate Instructions

- The **`subi`** mnemonic subtracts the value of a register with a constant and stores the result in a destination register.

```
subi   $rd, $rs, constant
```

  - $rd = Destination Register
  - $rs = Source Register
  - constant = A literal/constant value

Immediate Instructions

- The following instruction:

```
subi   $t0, $t1, 5
```

will subtract 5 from the values of register $t1 and store the result in register $t0

- One way to think of this instruction is: **`$t0 = $t1 - 5`**
  - However, this is not valid assembly code

Immediate Instructions

- There are immediate instruction mnemonics for the logical instructions.
  - No need for an immediate instruction for NOT operations

```
andi   $rd, $rs, constant
ori    $rd, $rs, constant
xori   $rd, $rs, constant
```

  - $rd = Destination Register
  - $rs = Source Register
  - constant = A literal/constant value

Immediate Instructions

- **Loading** describes the process of placing data into a register
  - Either as a constant value or data loaded from main memory

- The `li` (**l**oad **i**mmediate) mnemonic loads a constant to a register
  - `li` is a pseudo-operation; it is not included as part of the MIPS instruction set
  - The assembler converts an `li` instruction to appropriate instructions

Immediate Instructions

```
li    $rd, constant
```

- $rd = Destination Register
- constant = A literal/constant value

```
li    $t0, 17
```

- This instruction would load the value 17 to register $t0
  - Essentially, $t0 = 17

Immediate Instructions

- **Moving** describes the process of copying data from one register to another

- The **`move`** mnemonic copies data from one register to another
  - **`move`** is a pseudo-operation; it is not included as part of the MIPS instruction set
  - The assembler converts a **`move`** instruction to appropriate instructions

Immediate Instructions

```
move   $rd, $rs
```

- $rd = Destination Register
- $rs = Source Register

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

- This instruction would copy the value in $t0 to register $t1

Immediate Instructions

```
li      $t0, 17
move    $t1, $t0
```

$t0 = 00000000 $t1 = 00000000	**`li $t0, 17`**	$t0 = **00010001** $t1 = 00000000
$t0 = 00010001 $t1 = 00000000	**`move $t1, $t0`**	$t0 = 00010001 $t1 = **00010001**

Immediate Instructions

- As previously stated, the constants can only be 16 bits in size
  - In two's complement this gives us a range of -32,768 through 32,767
  - Or 0 through 65,535 for unsigned numbers

- When using the `li` pseudo-operation, it assigns the constant to the *lower-order half* of the register's 32 bits.
- Another instruction is used to assign a constant to the *higher-order half* of the register's 32 bits.

Immediate Instructions

- To demonstrate, we'll use hexadecimal constants as this will easily show the higher and lower halves of the register.

- Let's say we want to load the number 75,000 (a number that requires at least 17 bits) in register $t0

- $75{,}000_{10} = 00000000000000010010010011111000_2 = 000124F8_{16}$

Immediate Instructions

$$000124F8_{16}$$

Upper Half Lower Half

- To load a constant to the higher-order half of the register, the **`lui`** (**l**oad **u**pper **i**mmediate) pseudo-operation is used.

```
lui    $rd, constant
```

  - $rd = Destination Register
  - constant = A literal/constant value

Immediate Instructions

- The **`li`** pseudo-instruction assigns data to the lower half and clears/zeros the upper half

- The **`lui`** instruction assigns data to the upper half and clears/zeros the lower half

Immediate Instructions

- This demonstrates using hexadecimal literals instead of the decimal literals used in previous slides
  - A 0x prefix indicates a hexadecimal literal/constant

$t0 = 0x00000000	**`lui`**	**`$t0, 0x0001`**	$t0 = 0x**00010000**
$t0 = 0x00000000	**`li`**	**`$t0, 0x24f8`**	$t0 = 0x**000024F8**

Immediate Instructions

- Both examples below will yield the **wrong** value assigned to $t0

$t0 = 0x00000000	**`lui`**	**`$t0, 0x0001`**	$t0 = 0x**00010000**
$t0 = 0x00010000	**`li`**	**`$t0, 0x24f8`**	$t0 = 0x**000024F8** ✖

$t0 = 0x00000000	**`li`**	**`$t0, 0x24f8`**	$t0 = 0x**000024F8**
$t0 = 0x00010000	**`lui`**	**`$t0, 0x0001`**	$t0 = 0x**00010000** ✖

Immediate Instructions

- Start by loading the upper half

$t0 = 0x00000000 **`lui $t0, 0x0001`** $t0 = 0x**0001**0000

Immediate Instructions

- Then, perform an or operation with the lower half

$t0 = 0x00000000	**`lui $t0, 0x0001`**	$t0 = 0x**0001**0000
$t0 = 0x00010000	**`ori $t0, 0x24f8`**	$t0 = 0x**000124F8**

000000000000000100000000000000000

OR 00000000000000000**0010010011111000**

00000000000000010010010011111000 = **000124F8**$_{16}$

Printing Integers

- MIPS has special ***system calls*** to provide services like reading input and printing output.

- We'll focus only on printing integers and show examples of printing other data types and reading keyboard input in later lectures.

Printing Integers

- To print an integer, the number **must** be placed in register $a0
- The system call code for printing an integer is 1
  - 1 **must** be placed in register $v0
- When the above two steps are complete, the **`syscall`** instruction is used to perform the function indicated by the number in $v0

Printing Integers

```
move        $a0, $t0     #Copies the value in $t0 to $a0
li          $v0, 1       #Sets the syscall code for printing an integer
syscall                  #Prints the integer in register $a0
```

An Example in MARS

- Open MARS
  - See MARS Download document in Canvas

- Click the New button to create a new assembly source code file

An Example in MARS

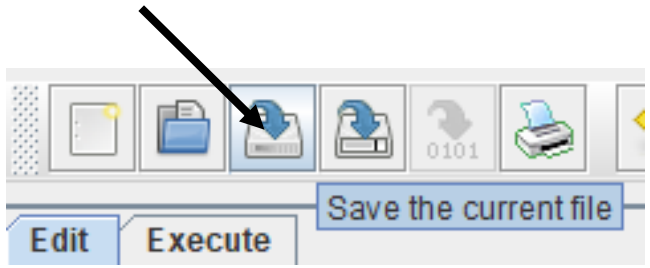
- Enter the following:

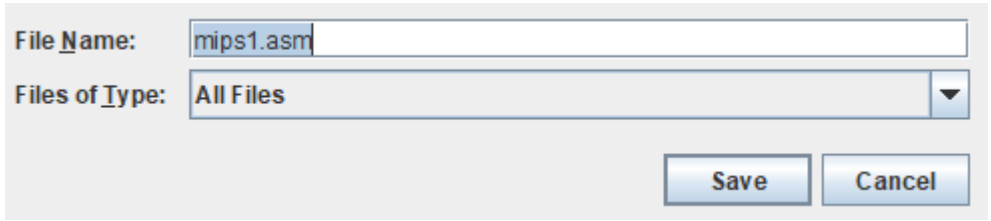
mips1.asm*

```
1  #Prints the number 15
2
3  li $t1, 15       #Loads the constant 15 to $t1
4
5  move $a0, $t1    #Copies $t1 (15) to $a0
6
7  li $v0, 1        #Sets the syscall code for printing an integer
8
9  syscall          #Prints the integer ($v0 = 1) in $a0 (15)
10
```

An Example in MARS

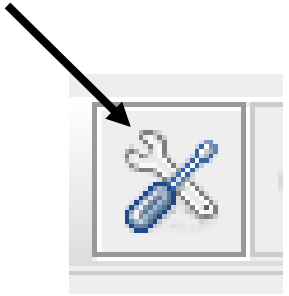
- Save the .asm file
  - The default name “mips1.asm” is fine





An Example in MARS

- Click the Assemble button



An Example in MARS

- You should see something similar to what is shown below

Edit | Execute

Text Segment

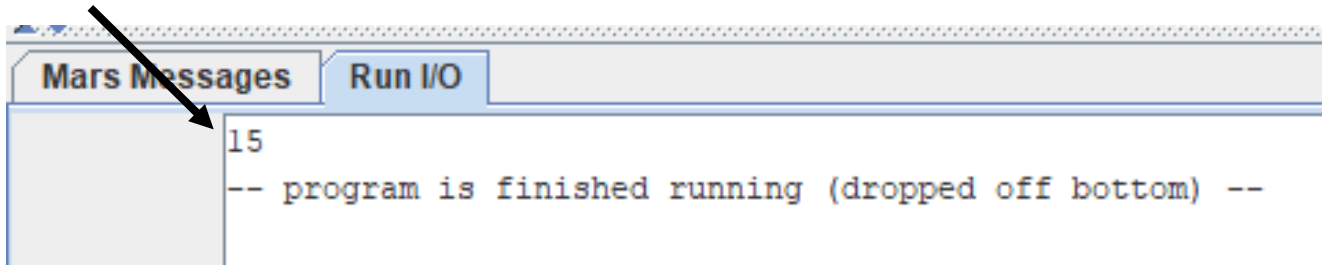
Bkpt	Address	Code	Basic	Source
	0x00400000	0x2409000f	addiu $9,$0,0x0000000f	3: li $t1, 15 #Loads the constant 15 to $t1
	0x00400004	0x00092021	addu $4,$0,$9	5: move $a0, $t1 #Copies $t1 (15) to $a0
	0x00400008	0x24020001	addiu $2,$0,0x00000001	7: li $v0, 1 #Sets the syscall code for printing an integer
	0x0040000c	0x0000000c	syscall	9: syscall #Prints the integer ($v0 = 1) in $a0 (15)

An Example in MARS

- Click the Run Current Program button

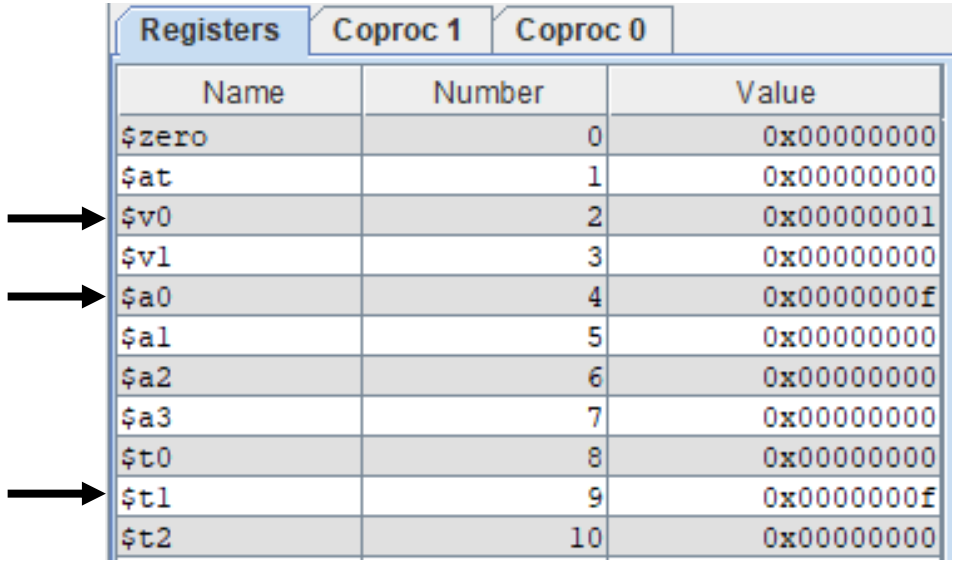
An Example in MARS

- 15 should be displayed in the Run I/O pane at the bottom of MARS



An Example in MARS

- Note that you can see the data in the registers at the conclusion of the program
  - Values are displayed in hexadecimal



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000001
$v1	3	0x00000000
$a0	4	0x0000000f
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x0000000f
$t2	10	0x00000000

An Example in MARS

- To re-run a program, click the Reset MARS button

- Then click the Run Current Program button again