CSE 220:
Systems Fundamentals I
Unit 3:
MIPS Assembly:
Basic Instructions,
System Calls, Endianness

Computer Instructions

- Recall that a computer's architecture is the programmer's view of a computer
- The architecture is defined, in part, by its **instruction set**
- These instructions are encoded in binary as the architecture's **machine language**
- Because reading and writing machine language is tedious, instructions are represented using *mnemonics* ("neh-MAHN-icks") as **assembly language**
- You are going to be learning the fundamentals of the MIPS architecture in this course, so that means we will be covering MIPS assembly language

MIPS Architecture Design Principles

1. Simplicity favors regularity
   - Simple, similar 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

MIPS Assembly

- Why MIPS and not x86?
- Like x86, MIPS CPUs are used in real products, but mostly embedded systems like network routers
- The MIPS architecture is simpler than x86 architecture, which makes the assembly language simpler and easier to learn
- Good textbooks and educational resources exist for MIPS
- Once you learn one architecture and its assembly language, others are easy to learn
- Fun fact: MIPS was invented by John Hennessy, the former president of Stanford University and alumnus of Stony Brook's MS and PhD programs in computer science

Instructions: Addition

- Java code: `a = b + c;` MIPS assembly code: `add a, b, c`
- **`add`**: mnemonic indicates operation to perform
- **`b`**, **`c`**: **source operands** (on which the operation is to be performed)
- **`a`**: **destination operand** (to which the result is written)
- 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
- **`b`**, **`c`**: source operands
- **`a`**: destination operand
- It should come as no surprise that the code is virtually the same as for an addition
  1. Simplicity favors regularity
     - Consistent instruction format
     - Same number of operands (two sources and one destination)
     - Easier to encode and handle in hardware

Multiple 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
```

- In MIPS assembly, the **#** symbol denotes a comment

2. Make the common case fast
   - More complex instructions (that are less common) are performed using multiple simple instructions
   - MIPS is a **reduced instruction set computer** (RISC), with a small number of simple instructions
   - Other architectures, such as Intel's x86, are **complex instruction set computers** (CISC)

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
  - Memory
  - Constants (also called **immediates**)
    - Included as part of an instruction

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

Operands: Registers

- 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–$s7`**, are used to hold variables
  - The **temporary registers**, **`$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.

Instructions with Registers

- Let's revisit the **`add`** instruction
- Java code: MIPS assembly code:

```
a = b + c;      # $s0 = a, $s1 = b, $s2 = c
                add $s0, $s1, $s2
```

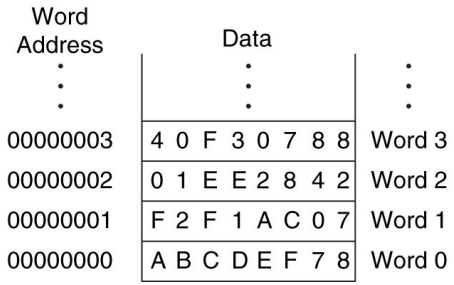
- When programming in a high-level language like Java, you generally don't (and shouldn't) comment every single line of code
- With MIPS assembly you should!
  - Assign meaning to registers and calculations
  - Even simple formulas will have to be implemented using at least several lines of assembly code
  - So comment EVERYTHING

Operands: Memory

- A typical program uses too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers
  - Less frequently used values will need to be copied from registers to memory for "safe keeping" when we run out of registers
  - Later we will need to copy the values from memory back to the register file when we need to do a calculation with them

Operands: Memory

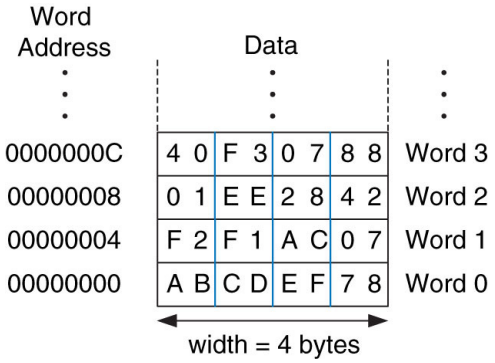
- Each 32-bit data **word** has a unique 32-bit address
- A **word** is the unit of data used natively by a CPU
- A possible logical structure of main memory (**word-addressable**), which is ***not*** how MIPS actually works:



Operands: Memory

- Most computer architectures cannot read individual bits from memory
- Rather, the architecture's instruction set can process only entire words or individual bytes
- If the smallest unit of data we can read from memory is a word, we say that the memory is **word-addressable**
  - In this case, memory addresses would be assigned sequentially, as in the previous figure
- The MIPS architecture, in contrast, is **byte-addressable**
  - Each byte has its own memory address

Operands: Memory



- Byte-addressable memory (what MIPS uses)

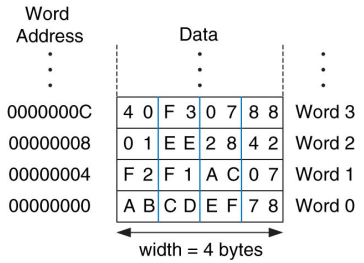
Reading Word-Addressable Memory

- A memory read is called a **load**
- Mnemonic: *load word* (`lw`)
- Format: `lw $s0, 16($t1)`
- Address calculation:
  - add **base address** (`$t1`) to the **offset** (`16`)
  - so a register first needs to have the base address that we want to add to the offset
  - **effective address** = (`$t1 + 16`)
- Result: `$s0` holds the value at address (`$t1 + 16`)
- Any register may be used to hold the base address

Reading Word-Addressable Memory

- Example: suppose we want to read a word of data at memory address 8 into **`$s3`**
  - address = (**`$0`** + 8) = 8
- So what we want is for **`$s3`** to hold 0x01EE2842
- Assembly code:

```
# read memory
#  word 2 into $s3
lw $s3, 8($0)
```



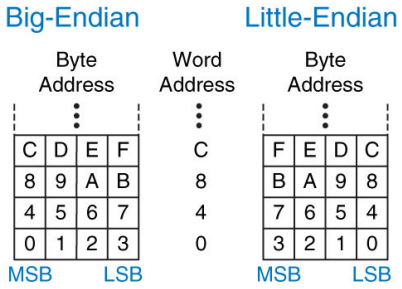
Writing Word-Addressable Memory

- 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 addr 8
```


Big-Endian & Little-Endian Memory

- 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
- Word address is the same in either case

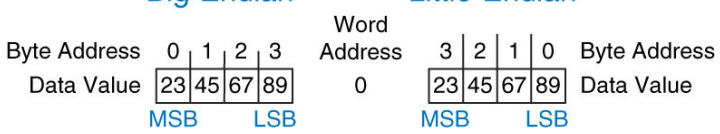


Big-Endian & Little-Endian Memory

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

```
sw $t0, 0($0)
lb $s0, 1($0)
```

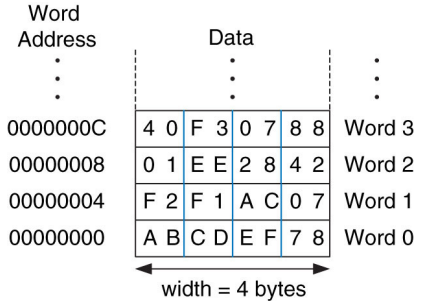
- Big-endian: 0x00000045
- Little-endian: 0x00000067



- The MIPS simulator we will use is little-endian

Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: *load byte* (**`lb`**) and *store byte* (**`sb`**)
- 32-bit word = 4 bytes, so word addresses increment by 4
- So when performing a **`lw`** or **`sw`**, the effective address must be a multiple of 4



Byte-Addressable Memory

- When loading a byte, what do we do with the other 24 bits in the 32-bit register?
- **`lb`** sign-extends to fill the upper 24 bits
- Suppose the byte loaded is **`zxxx xxxx`** ← 8 bits
  - The bit **`z`** is copied into the upper 24 bits
- Normally with characters do not want to sign-extend the byte, but rather prepend zeroes
  - This is called **zero-extension**
- MIPS instruction that does zero-extension when loading bytes:
  - *load byte unsigned*: **`lbu`**

Reading Byte-Addressable Memory

- The address of a memory word must be a multiple of 4
- For example,
  - the address of memory word #2 is 2 × 4 = 8
  - the address of memory word #10 is 10 × 4 = 40 (0x28)
- So do not forget this: MIPS is byte-addressed, not word-addressed!
  - To read/write a word from/to memory, your **`lw/sw`** instruction must provide an effective address that is **word-aligned**

Instruction Formats

4. Good design demands good compromises

- Multiple instruction formats allow flexibility
  - **`add`**, **`sub`**: use 3 register operands
  - **`lw`**, **`sw`**: use 2 register operands and a constant
- Number of instruction formats kept small…
  - …to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster)

Instruction Formats

- **`lw`** and **`sw`** use constants or **immediates**
- *Immediately* available from instruction
- The immediate value is stored in the instruction as a 16-bit two's complement number
- **`addi`**: add immediate
- Is subtract immediate (**`subi`**) necessary?
- Java code:

```
a = a + 4;
b = a - 12;
```

MIPS assembly code:

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```


Machine Language

- Binary representation of instructions
- Computers only "understand" 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- 3 instruction formats:
  - **R-Type**: register operands (**register-type** instruction)
  - **I-Type**: immediate operand (**immediate-type** instruction)
  - **J-Type**: for jumping (**jump-type** instruction) – more on this later

R-Type Instructions

- 3 register operands:
  - **`rs`**, **`rt`**: source registers
  - **`rd`**: destination register
- Other fields:
  - **`op`**: the **operation code** or **opcode** (0 for R-type)
  - **`funct`**: the **function**; with opcode, tells CPU what operation to perform
  - **`shamt`**: the **shift amount** for shift instructions; otherwise it's 0

R-type

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

R-Type Examples

Assembly Code / Field Values

	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

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	

- Note the order of registers in the assembly code:
  `add rd, rs, rt`

I-Type Instructions

- 3 operands:
  - **`rs`**, **`rt`**: register operands
  - **`imm`**: 16-bit two's complement immediate
- Other fields:
  - **`op`**: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by opcode

I-type

I-Type Examples

Assembly 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

Field Values

Machine Code

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	

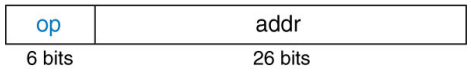
Note the differing order of registers in assembly and machine codes:

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```


J-Type Instructions

- 26-bit address operand (**`addr`**)
- Used for jump instructions (**`j`**)
  - if-statements, loops, functions

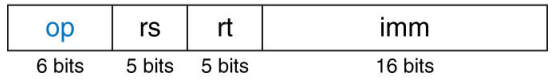
J-type



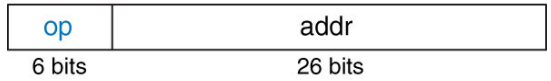
Review: Instruction Formats

R-type

I-type



J-type

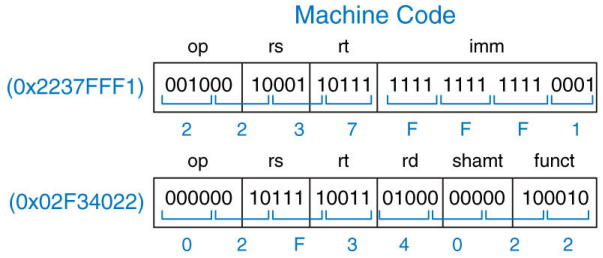


Power of the Stored Program

- 32-bit instructions and data are stored in memory
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Affords **general purpose computing**
- Program execution:
  - Processor **fetches** (reads) instructions from memory in sequence
  - Processor performs the specified operation and fetches the next instruction

Interpreting Machine Code

- Start with opcode: tells how to parse the rest
- If opcode all 0's we have an R-type instruction
  - Function bits (**`funct`**) indicate operation
- Otherwise, opcode tells operation



Interpreting Machine Code

Field Values Assembly Code

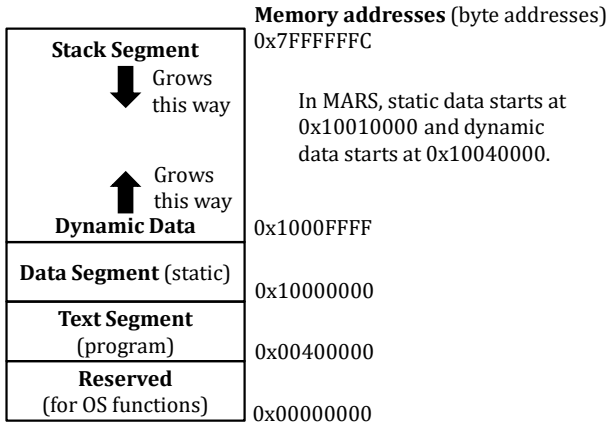
op	rs	rt	imm	
8	17	23	–15	`addi $s7, $s1, -15`

op	rs	rt	rd	shamt	funct	
0	23	19	8	0	34	`sub $t0, $s7, $s3`

MIPS Assembly Programming

- There's a lot more to the MIPS instruction set still to cover, but we (almost) know enough now to write some simple programs that do computations
- Every statement is divided into fields:
  - [Label:] operation [operands] [#comment]
  - Parts in square brackets are optional
- A **label** is a sequence of alphanumeric characters, underscores and dots. Cannot begin with a number. Ends with a colon.
  - After the assembler has **assembled** (processed) your code, the label refers to the address of where the line of MIPS code is stored in memory

MIPS Memory Layout



MIPS Assembly Programming

- The **`main`** label indicates the start of a program
- Labels are also used to give names to locations in memory where we want to store data (we will see this shortly)
- Assembly programs also include **assembler directives**, which start with a dot and give commands to the assembler, but are not assembly language instructions
  - **`.text`**: beginning of the text segment
  - **`.data`** : beginning of data segment
  - **`.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 Assembly Programming

- Strings (**`.asciiz`** and **`.ascii`**) are enclosed in quotes
  - They recognize escape sequences: **`\n`**, **`\t`**, **`\0`**, **`\r`**, etc.
- Finally, we need some way of doing basic input and output
  - The computer's architecture does not handle those responsibilities, but relies on the operating system
- A **system call** is a request made by the program for the OS to perform some service, such as to read input, print output, quit the program, etc.
- In our MIPS assembly programs we write **`syscall`** to perform a system call
- We have to include a numerical code (loaded into **`$v0`**) to indicate the service requested

MIPS System Calls

Service	System Call Code	Arguments	Result
`print_int`	1	$a0=integer	
`print_float`	2	$f12=float	
`print_double`	3	$f12=double	
`print_string`	4	$a0=string	
`read_int`	5		integer (in $v0)
`read_float`	6		float (in $f0)
`read_double`	7		double (in $f0)
`read_string`	8	$a0=buffer, $a1=length	
`sbrk`	9	$a0=amount	
`exit`	10		

MIPS System Calls

- **`sbrk`** allocates memory in the heap (e.g., large chunks of memory)
- These are the original MIPS system calls
- The SBU MARS simulator has a few custom ones you will learn about later. These system calls are not available in the "vanilla" version of MARS publicly available on the web.

Generating Constants

- 16-bit constants using **`addi`**:
- Java code:

```
// int is a 32-bit
// signed word
int a = 0x4f3c;
```

MIPS assembly code:

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants use load upper immediate (**`lui`**) and **`ori`** (more on **`ori`** in a few minutes):
- Java code:

```
int a = 0xFEDC8765;
```

MIPS assembly code:

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```


MIPS Assembly Pseudoinstructions

- MIPS implements the RISC concept
  - Relatively few, simple instructions
- But there are some operations that assembly language programmers need to do frequently that are not so natural to write in "native" MIPS assembly instructions
  - These instructions can instead be written as a single **pseudoinstruction**
- Example: to load a 32-bit integer into a register requires **`lui`** and **`ori`**
  - Instead we can use the **`li`** (*load immediate*) pseudoinstruction
  - Example: **`li $v0, 4  # loads 4 into $v0`**

MIPS Assembly Pseudoinstructions

- Another useful pseudoinstruction is **`la`** (*load address*)
- Example: assume that **`str`** is a label (i.e., a memory address)
  - **`la $a0, str # loads addr of str into $a0`**
- The **`move`** pseudoinstruction copies the contents of one register to another
  - **`move $1, $2 # equiv to add $1, $2, $0`**

MIPS Program: Hello World

- No introduction to a new programming language would be complete without the obligatory "hello world" program
- Let's see how this is done in MIPS

Multiplication and Division

- Special registers: **`lo`**, **`hi`**
- 32-bit × 32-bit multiplication produces a 64-bit result
  - **`mult $s0, $s1`**
  - Result in {**`hi`**, **`lo`**}
- 32-bit division produces a 32-bit quotient and a 32-bit remainder
  - **`div $s0, $s1`**
  - Quotient in **`lo`**
  - Remainder in **`hi`**
- Instructions to move values from lo/hi special registers
  - **`mflo $s2`**
  - **`mfhi $s3`**

MIPS Assembly Pseudoinstructions

- Another useful pseudoinstruction that will help us write up a program:
  - **`mul d, s, t  # d = s * t`**
- **`mul d, s, t`** is equivalent to:

```
mult s, t
mflo d
```

- Similar pseudoinstruction for **`div`**

MIPS Program: Compute $Ax^2 + Bx + C$

- For the first version of this program we will hard-code the values for the three coefficients and x
- Major steps of our program:
  1. Load values of A, B, C and x from memory into registers
  2. Compute $Ax^2 + Bx + C$
     - Requires 5 total arithmetical operations
  3. Print the result with an appropriate message
     - Requires several system calls
- I am going to comment nearly every single line of MIPS assembly code I write. You should do the same on your homework!

MIPS Program: Compute $Ax^2 + Bx + C$

- For version 2 of the program we will add prompts to ask the user to enter the values for *x*, *A*, *B* and *C*
- Each prompt requires two system calls:
  - One to print the prompt message on the screen (if one is desired/required)
  - Another to read the input

Logical Instructions

- **`and`**, **`or`**, **`xor`**, **`nor`**
- **`and`**: useful for **masking** bits
  - Masking out (excluding) all but the least significant byte of a value: 0xF234012E AND 0x000000FF = 0x0000002E
  - Why? Let's see:

```
0xF234012E AND 0x000000FF
1111 0010 0011 0100 0000 0001 0010 1110
0000 0000 0000 0000 0000 0000 1111 1111
0000 0000 0000 0000 0000 0000 0010 1110
   0    0    0    0    0    0    2    E
```


Logical Instructions

- **`or`**: useful for **combining** bit fields
  - Combine 0xF2340000 with 0x000012BC:
  - 0xF2340000 OR 0x000012BC = 0xF23412BC
  - Written as bits:

```
0xF2340000 OR 0x000012BC
1111 0010 0011 0100 0000 0000 0000 0000
0000 0000 0000 0000 0001 0010 1011 1100
1111 0010 0011 0100 0001 0010 1011 1100
   F    2    3    4    1    2    B    C
```


Logical Instructions

- **`nor`**: useful for **inverting** bits
  - A NOR $0 = NOT A
- **`andi`**, **`ori`**, **`xori`**
  - 16-bit immediate is zero-extended (not sign-extended)
  - **`nori`** not needed (can use **`ori`** and **`nor`**)
  - Examples in a moment…

Logical Instructions Examples

	Source Registers							
$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code		Result							
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or  $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instructions Examples

	Source Values							
$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code		Result							
`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

Shift Instructions

- Allow you to shift the value in a register left or right by up to 31 bits
- **`sll`**: shift left logical
  - Example: **`sll $t0, $t1, 5  # $t0 = $t1 << 5`**
  - Shifts bits to the left, filling least significant bits with zeroes
- **`srl`**: shift right logical
  - Example: **`srl $t0, $t1, 5  # $t0 = $t1 >>> 5`**
  - Shifts zeroes into most significant bits
- **`sra`**: shift right arithmetic
  - Example: **`sra $t0, $t1, 5  # $t0 = $t1 >> 5`**
  - Shifts sign bit into most significant bits

Shift Instructions Examples

	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	1000	0000
`srl $s2, $s1, 4`	$s2	0000	1111	0011	0000	0000	0000	0010	1010
`sra $s3, $s1, 4`	$s3	1111	1111	0011	0000	0000	0000	0010	1010

Variable-Shift Instructions

- These R-type instructions shift bits by number in a register
- **`sllv`**: shift left logical variable
  - **`sllv rd, rt, rs`** (note: **`rs`** and **`rt`** reversed)
  - **`rt`** has value to be shifted
  - 5 least significant bits of **`rs`** give amount to shift (0-31)
  - Example: **`sllv $t0, $t1, $t2 # $t0 = $t1 << $t2`**
- **`srlv`**: shift right logical variable
  - Example: **`srlv $t0, $t1, $t2 # $t0 = $t1 >>> $t2`**
- **`srav`**: shift right arithmetic variable
  - Example: **`srav $t0, $t1, $t2 # $t0 = $t1 >> $t2`**
- **`shamt`** field is ignored

Variable-Shift Instructions Examples

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

Rotate or Circular Shift

- Bits are not lost when we **rotate** them (i.e., do a "circular shift")
- They wrap around and enter the register from the other end
- These are pseudo-instructions:
  - **`rol`**: *rotate left*
  - **`ror`**: *rotate right*
- Example: **`rol $t2, $t2, 4`**
  - Rotate left bits of **`$t2`** by 4 positions:
  - **`1101 0010 0011 0100 0101 0110 0111 1000`**
  - **`0010 0011 0100 0101 0110 0111 1000 1101`**

Applications of Bitwise Operators

- The bitwise operations are useful in situations where we have a set of Yes/No properties and using many Boolean variables would waste memory
  - Example: file access flags in Unix/Linux
- Network protocols: packets have very specific formats, which may include many bits that need to be extracted to determine how to process a packet
- Compression algorithms sometime work on a bit-by-bit basis
- Implementing a mathematical set of values: item a_i is present in the set if bit i is 1; not present if the bit is 0

Bitwise Operator Examples

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in **`$t0`**. Simply use this:

```
andi $t0, $t0, 0xFF
0001 0010 0011 0100 0101 0110 0111 1000
0000 0000 0000 0000 0000 0000 0111 1000
```


Bitwise Operator Examples

- Suppose we want to isolate byte 1 (bits 8 to 15) of a word in **`$t0`**. (Bits are numbered right-to-left.) We can use:

```
andi $t0,$t0,0xFF00
```

  but then we still need to do a logical shift to the right by 8 bits. Why? To move the byte we have isolated into byte 0 and also to set bytes 1, 2 and 3 to all zeroes.
- Could use instead:
- `sll $t0,$t0,16` *
- `srl $t0,$t0,24` **

```
0001 0010 0011 0100 0101 0110 0111 1000
0101 0110 0111 1000 0000 0000 0000 0000 *
0000 0000 0000 0000 0000 0000 0101 0110 **
```


Bitwise Operator Examples

- In binary, multiplying by 2 is same as shifting left by 1 bit:
  - $11_2 \times 10_2 = 110_2$
- Multiplying by 4 is same as shifting left by 2 bits:
  - $11_2 \times 100_2 = 1100_2$
- Multiplying by 2^n is same as shifting left by *n* bits
- Since shifting may be faster than multiplication, a good compiler (e.g., C or Java) will recognize a multiplication by a power of 2 and compile it as a shift:
  - **`a = a*8;`** would compile as **`sll $s0,$s0,3`**
- Likewise, shift right to do *integer division* by powers of 2.
- Remember to use **`sra`**, not **`srl`**. Why?

MIPS Programming Tips

- Initialize all your variables as needed (e.g., use **`li`**)
  - The MARS simulator fills the memory with zeroes, but this is merely a convenience and luxury
  - When we test your homework programs, *we may fill the registers and main memory with garbage to make sure you initialize registers with values!*
- Use the MARS debugger to fix problems with your code
  - The Registers view (on right) is especially useful
- Use **`$s0–$s7`** for local variables, and **`$t0–$t9`** for temporary values, such as intermediate results
  - We will see just how important this distinction is when we study functions!