Chapter 6 :: Topics

- Introduction
- Assembly Language
- Machine Language
- Programming
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembling, & Loading
- Odds and Ends
• Jumping up a few levels of abstraction

• **Architecture**: programmer’s view of computer
  – Defined by instructions & operand locations

• **Microarchitecture**: how to implement an architecture in hardware (covered in Chapter 7)
Assembly Language

• **Instructions:** commands in a computer’s language
  – **Assembly language:** human-readable format of instructions
  – **Machine language:** computer-readable format (1’s and 0’s)

• **MIPS architecture:**
  – Developed by John Hennessy and his colleagues at Stanford and in the 1980’s.
  – Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you’ve learned one architecture, it’s easy to learn others
John Hennessy

- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold
Underlying design principles, as articulated by Hennessy and Patterson:

1. Simplicity favors regularity
2. Make the common case fast
3. Smaller is faster
4. Good design demands good compromises
Instructions: Addition

C Code

```c
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 performed)
- **a**: destination operand (to which the result is written)
Instructions: Subtraction

• Similar to addition - only mnemonic changes

C Code
a = b - c;

MIPS assembly code
sub a, b, c

• sub: mnemonic
• b, c: source operands
• a: destination operand
Design Principle 1

Simplicity favors regularity

• Consistent instruction format
• Same number of operands (two sources and one destination)
• Easier to encode and handle in hardware
More complex code is handled by multiple MIPS instructions.

C Code

```c
a = b + c - d;
```

MIPS assembly code

```asm
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```
Make the common case fast

- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) 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
  - Memory
  - Constants (also called *immediates*)
MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called “32-bit architecture” because it operates on 32-bit data
Smaller is Faster

- MIPS includes only a small number of registers
# MIPS Register Set

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Function return values</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Function arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved variables</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>OS temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Function return address</td>
</tr>
</tbody>
</table>
• Registers:
  – $ before name
  – Example: $0, “register zero”, “dollar zero”

• Registers used for specific purposes:
  • $0 always holds the constant value 0.
  • the saved registers, $s0-$s7, used to hold variables
  • the temporary registers, $t0 - $t9, used to hold intermediate values during a larger computation
  • Discuss others later
Instructions with Registers

- Revisit add instruction

C Code

```c
a = b + c
```

MIPS assembly code

```mips
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```
• 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
### Word-Addressable Memory

- Each 32-bit data word has a unique address

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000003</td>
<td>4 0 F 3</td>
<td>0 7 8 8</td>
<td></td>
<td>Word 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000002</td>
<td>0 1 E E</td>
<td>2 8 4 2</td>
<td></td>
<td>Word 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000001</td>
<td>F 2 F 1</td>
<td>A C 0 7</td>
<td></td>
<td>Word 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000000</td>
<td>A B C D</td>
<td>E F 7 8</td>
<td></td>
<td>Word 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** MIPS uses byte-addressable memory, which we’ll talk about next.
• Memory read called **load**
• **Mnemonic:** *load word* (*lw*)
• **Format:**
  
  \[ lw \ $s0,\ 5($t1) \]

• **Address calculation:**
  – *add base address* (*$t1*) to the *offset* (*5*)
  – *address* = (*$t1 + 5*)

• **Result:**
  – *$s0* holds the value at address (*$t1 + 5*)

Any register may be used as base address
• **Example:** read a word of data at memory address 1 into $s3
  - address = ($0 + 1) = 1
  - $s3 = 0xF2F1AC07 after load

**Assembly code**

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000003</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>00000002</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>00000001</td>
<td>F 2 F 1 A C 0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000000</td>
<td>A B C D E F 7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>
• Memory write are called \textit{store}
• Mnemonic: \textit{store word} (\texttt{sw})
Example: Write (store) the value in $t4 into memory address 7
- add the base address ($0) to the offset (0x7)
- address: ($0 + 0x7) = 7

Offset can be written in decimal (default) or hexadecimal

Assembly code
sw $t4, 0x7($0) # write the value in $t4
            # to memory word 7

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>00000003</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>00000002</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>00000001</td>
<td>F 2 F 1 A C 0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000000</td>
<td>A B C D E F 7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>
Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address increments by 4

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000C</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>0000000008</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>0000000004</td>
<td>F 2 F 1 A C 0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>0000000000</td>
<td>A B C D E F 7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>

width = 4 bytes
The address of a memory word must now be multiplied by 4. For example,
- 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
**Example:** Load a word of data at memory address 4 into $s3.

$s3$ holds the value 0xF2F1AC07 after load

**MIPS assembly code**

```
lw $s3, 4($0)  # read word at address 4 into $s3
```

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Width = 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000C</td>
<td>0x40F30788</td>
<td>Word 3</td>
</tr>
<tr>
<td>00000008</td>
<td>0x01EE2842</td>
<td>Word 2</td>
</tr>
<tr>
<td>00000004</td>
<td>0xFF1AC07</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000000</td>
<td>0xADEF78</td>
<td>Word 0</td>
</tr>
</tbody>
</table>
Writing Byte-Addressable Memory

- **Example:** stores the value held in $t7 into memory address 0x2C (44)

**MIPS assembly code**

```
sw $t7, 44($0)  # write $t7 into address 44
```

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Width = 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000C</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>00000008</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>00000004</td>
<td>F 2 F 1 A C 0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000000</td>
<td>A B C D E F 7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>
Big-Endian & Little-Endian Memory

- How to number 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
- **Word address** is the same for big- or little-endian

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Address</td>
<td>Word Address</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>C D E F</td>
<td>C</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0</td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
</tr>
</tbody>
</table>

- Big-Endian & Little-Endian Memory
• Jonathan Swift’s *Gulliver’s Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end

• It doesn’t really matter which addressing type used – except when the two systems need to share data!

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong></td>
<td><strong>Word</strong></td>
</tr>
<tr>
<td><strong>Address</strong></td>
<td><strong>Address</strong></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>C D E F C</td>
<td>F E D C</td>
</tr>
<tr>
<td>8 9 A B 8</td>
<td>B A 9 8</td>
</tr>
<tr>
<td>4 5 6 7 4</td>
<td>7 6 5 4</td>
</tr>
<tr>
<td>0 1 2 3 0</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td>MSB</td>
<td>MSB</td>
</tr>
<tr>
<td>LSB</td>
<td>LSB</td>
</tr>
</tbody>
</table>

Jonathan Swift’s *Gulliver’s Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end.

It doesn’t really matter which addressing type used – except when the two systems need to share data!
• Suppose $t0$ initially contains 0x23456789
• After following code runs on big-endian system, what value is $s0$?
• In a little-endian system?

  sw $t0$, 0($0)
  lb $s0$, 1($0)
• Suppose $t0$ initially contains 0x23456789
• After following code runs on big-endian system, what value is $s0$?
• In a little-endian system?
  \[\text{sw } t0, 0($0)\]
  \[\text{lb } s0, 1($0)\]
• Big-endian: 0x00000045
• Little-endian: 0x00000067
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).
• \texttt{lw} and \texttt{sw} use constants or \textit{immediates}
• \textit{immediately} available from instruction
• 16-bit two’s complement number
• \texttt{addi}: add immediate
• Subtract immediate (\texttt{subi}) necessary?

\begin{align*}
\text{C Code} & \\
\text{a = a + 4;} & \\
\text{b = a - 12;} & \\
\text{MIPS assembly code} & \\
\text{# } & \\
\text{$s0 = a, \ s1 = b$} & \\
\text{addi } & \\
\text{$s0, \ s0, \ 4$} & \\
\text{addi } & \\
\text{$s1, \ s0, \ -12$} & \\
\end{align*}
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
  - **I-Type**: immediate operand
  - **J-Type**: for jumping (discuss later)
**R-Type**

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

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>
## R-Type Examples

### Assembly Code

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0, $s1, $s2</td>
</tr>
<tr>
<td>sub $t0, $t3, $t5</td>
</tr>
</tbody>
</table>

### Field Values

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>13</td>
<td>8</td>
<td>0</td>
<td>34</td>
</tr>
</tbody>
</table>

### Machine Code

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>1001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
<td>100000 (0x0238020)</td>
</tr>
<tr>
<td>000000</td>
<td>01011</td>
<td>01101</td>
<td>01000</td>
<td>00000</td>
<td>100010 (0x016D4022)</td>
</tr>
</tbody>
</table>

### Note

The order of registers in the assembly code:

add rd, rs, rt
I-Type

- **Immediate-type**
- 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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

### I-Type Examples

#### Assembly Code

<table>
<thead>
<tr>
<th>Code</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi $s0, $s1, 5</code></td>
<td>8 17 16 5</td>
</tr>
<tr>
<td><code>addi $t0, $s3, -12</code></td>
<td>8 19 8 -12</td>
</tr>
<tr>
<td><code>lw  $t2, 32($0)</code></td>
<td>35 0 10 32</td>
</tr>
<tr>
<td><code>sw  $s1, 4($t1)</code></td>
<td>43 9 17 4</td>
</tr>
</tbody>
</table>

#### Field Values

<table>
<thead>
<tr>
<th>op (6 bits)</th>
<th>rs (5 bits)</th>
<th>rt (5 bits)</th>
<th>imm (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10001</td>
<td>10000</td>
<td>0000 0000 0000 00101</td>
</tr>
<tr>
<td>001000</td>
<td>10011</td>
<td>01000</td>
<td>1111 1111 1111 0100</td>
</tr>
<tr>
<td>100011</td>
<td>00000</td>
<td>01010</td>
<td>0000 0000 0010 0000</td>
</tr>
<tr>
<td>101011</td>
<td>01001</td>
<td>10001</td>
<td>0000 0000 0000 0100</td>
</tr>
</tbody>
</table>

#### Machine Code

<table>
<thead>
<tr>
<th>op (6 bits)</th>
<th>rs (5 bits)</th>
<th>rt (5 bits)</th>
<th>imm (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x22300005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2268FFF4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8C0A0020</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xAD310004</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- `addi rt, rs, imm`
- `lw  rt, imm(rs)`
- `sw  rt, imm(rs)`
### Machine Language: J-Type

- **Jump-type**
- **26-bit address operand** \((\text{addr})\)
- **Used for jump instructions** \((j)\)

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Review: Instruction Formats

### R-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

### I-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

### J-Type

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation
The Stored Program

Assembly Code | Machine Code
--- | ---
lw $t2, 32($0) | 0x8C0A0020
add $s0, $s1, $s2 | 0x02328020
addi $t0, $s3, -12 | 0x2268FFFF
sub $t0, $t3, $t5 | 0x016D4022

**Program Counter (PC):** keeps track of current instruction

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040000C</td>
<td>0 1 6 D 4 0 2 2</td>
</tr>
<tr>
<td>00400008</td>
<td>2 2 6 8 F F F 4</td>
</tr>
<tr>
<td>00400004</td>
<td>0 2 3 2 8 0 2 0</td>
</tr>
<tr>
<td>00400000</td>
<td>8 C 0 A 0 0 2 0</td>
</tr>
</tbody>
</table>

Main Memory
Interpreting Machine Code

- Start with opcode: tells how to parse rest
- If opcode all 0’s
  - R-type instruction
  - Function bits tell operation
- Otherwise
  - opcode tells operation

### Machine Code

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10001</td>
<td>101111111111110001</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

### Field Values

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>17</td>
<td>23</td>
<td>-15</td>
</tr>
</tbody>
</table>

### Assembly Code

- `addi $s7, $s1, -15`
- `sub $t0, $s7, $s3`
• High-level languages:
  – e.g., C, Java, Python
  – Written at higher level of abstraction

• Common high-level software constructs:
  – if/else statements
  – for loops
  – while loops
  – arrays
  – function calls
Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage’s Analytical Engine
- She was the daughter of the poet Lord Byron
Logical Instructions

• **and, or, xor, nor**
  - **and**: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      \[ 0xF234012F \text{ AND } 0x000000FF = 0x0000002F \]
  - **or**: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      \[ 0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC \]
  - **nor**: useful for **inverting** bits:
    - A NOR $0 = \text{ NOT A}$

• **andi, ori, xori**
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - nori not needed
Logical Instructions Example 1

<table>
<thead>
<tr>
<th>Source Registers</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s1 1111 1111 1111 1111 0000 0000 0000 0000</td>
<td>and $s3, $s1, $s2</td>
</tr>
<tr>
<td>$s2 0100 0110 1010 0001 1111 0000 1011 0111</td>
<td>or $s4, $s1, $s2</td>
</tr>
<tr>
<td></td>
<td>xor $s5, $s1, $s2</td>
</tr>
<tr>
<td></td>
<td>nor $s6, $s1, $s2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s3</td>
</tr>
<tr>
<td>$s4</td>
</tr>
<tr>
<td>$s5</td>
</tr>
<tr>
<td>$s6</td>
</tr>
</tbody>
</table>
### Logical Instructions Example 1

<table>
<thead>
<tr>
<th>Source Registers</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s1 0111 1111 1111 1111 0000 0000 0000 0000</td>
<td>$s3 0100 0110 1010 0001 1111 0000 1011 0111</td>
</tr>
<tr>
<td>$s2 0100 0110 1010 0001 0000 0000 0000 0000</td>
<td>$s4 1111 1111 1111 1111 1111 0000 1011 0111</td>
</tr>
<tr>
<td>$s3 1111 1111 1111 1111 1111 0000 1011 0111</td>
<td>$s5 1011 1001 0101 1110 1111 0000 1011 0111</td>
</tr>
<tr>
<td>$s4 0000 0000 0000 0000 0000 1111 0100 1000</td>
<td>$s6 0000 0000 0000 0000 0000 1111 0100 1000</td>
</tr>
</tbody>
</table>

**Assembly Code**

- `and $s3, $s1, $s2`
- `or  $s4, $s1, $s2`
- `xor $s5, $s1, $s2`
- `nor $s6, $s1, $s2`
### Logical Instructions Example 2

#### Source Values

<table>
<thead>
<tr>
<th>$s1</th>
<th>0000 0000 0000 0000 0000 0000 1111 1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm</td>
<td>0000 0000 0000 0000 1111 1010 0011 0100</td>
</tr>
</tbody>
</table>

#### Assembly Code

- `andi $s2, $s1, 0xFA34`
- `ori  $s3, $s1, 0xFA34`
- `xori $s4, $s1, 0xFA34`

#### Result

<table>
<thead>
<tr>
<th>$s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s3</td>
</tr>
<tr>
<td>$s4</td>
</tr>
</tbody>
</table>

The `imm` value is zero-extended from the original value of 0x1234 to accommodate the source register(s).
### Logical Instructions Example 2

<table>
<thead>
<tr>
<th>Source Values</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s1</td>
<td>0000 0000 0000 0000 0000 0000 1111 1111</td>
</tr>
<tr>
<td>imm</td>
<td>0000 0000 0000 0000 1111 1010 0011 0100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Source Values</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>andi $s2, $s1, 0xFA34</td>
<td>$s2</td>
<td>0000 0000 0000 0000 0000 0000 0011 0100</td>
</tr>
<tr>
<td>ori $s3, $s1, 0xFA34</td>
<td>$s3</td>
<td>0000 0000 0000 0000 1111 1010 1111 1111</td>
</tr>
<tr>
<td>xori $s4, $s1, 0xFA34</td>
<td>$s4</td>
<td>0000 0000 0000 0000 1111 1010 1100 1011</td>
</tr>
</tbody>
</table>
Shift Instructions

• **sll**: shift left logical
  – **Example**: `sll $t0, $t1, 5  # $t0 <= $t1 << 5`

• **srl**: shift right logical
  – **Example**: `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`

• **sra**: shift right arithmetic
  – **Example**: `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`
- **sllv**: shift left logical variable
  - 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
Shift Instructions

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t0, $s1, 2</td>
<td>op</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>000000</td>
</tr>
<tr>
<td>000000</td>
</tr>
<tr>
<td>000000</td>
</tr>
</tbody>
</table>
Generating Constants

• 16-bit constants using `addi`:

  C Code
  ```c
  int a = 0xFEDC8765;
  ```

  MIPS assembly code
  ```
  # $s0 = a
  addi $s0, $0, 0x8765
  ```

• 32-bit constants using load upper immediate (lui) and `ori`:

  C Code
  ```c
  int a = 0x4f3c;
  ```

  MIPS assembly code
  ```
  # $s0 = a
  addi $s0, $0, 0x4f3c
  ```
Multiplication, Division

- Special registers: lo, hi
- $32 \times 32$ multiplication, 64 bit result
  - mult $s0$, $s1$
  - Result in \{hi, lo\}
- 32-bit division, 32-bit quotient, remainder
  - div $s0$, $s1$
  - Quotient in lo
  - Remainder in hi
- Moves from lo/hi special registers
  - mflo $s2$
  - mfhi $s3$
Branching

• Execute instructions out of sequence

• Types of branches:
  
  – Conditional
    • branch if equal ($beq$)
    • branch if not equal ($bne$)

  – Unconditional
    • jump ($j$)
    • jump register ($jr$)
    • jump and link ($jal$)
Review: The Stored Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040000C</td>
<td>0 1 6 D 4 0 2 2</td>
</tr>
<tr>
<td>00400008</td>
<td>2 2 6 8 F F F 4</td>
</tr>
<tr>
<td>00400004</td>
<td>0 2 3 2 8 0 2 0</td>
</tr>
<tr>
<td>00400000</td>
<td>8 C 0 A 0 0 2 0</td>
</tr>
</tbody>
</table>

Machine Code

lw    $t2, 32($0)    0x8C0A0020
add   $s0, $s1, $s2  0x02328020
addi  $t0, $s3, -12  0x2268FFF4
sub   $t0, $t3, $t5  0x016D4022
Conditional Branching \((\text{beq})\)

# MIPS assembly

```plaintext
addi $s0, $0, 4  # $s0 = 0 + 4 = 4
addi $s1, $0, 1  # $s1 = 0 + 1 = 1
sll $s1, $s1, 2  # $s1 = 1 << 2 = 4
beq $s0, $s1, target  # branch is taken
addi $s1, $s1, 1  # not executed
sub $s1, $s1, $s0  # not executed

target:
add $s1, $s1, $s0  # label
```

Labels indicate instruction location. They can’t be reserved words and must be followed by colon (:).
The Branch Not Taken (bne)

# MIPS assembly

addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
bne   $s0, $s1, target    # branch not taken
addi  $s1, $s1, 1         # $s1 = 4 + 1 = 5
sub   $s1, $s1, $s0       # $s1 = 5 - 4 = 1

target:
add    $s1, $s1, $s0      # $s1 = 1 + 4 = 5
Unconditional Branching (j)

# MIPS assembly

```
addi $s0, $0, 4  # $s0 = 4
addi $s1, $0, 1  # $s1 = 1
j              target  # jump to target
sra $s1, $s1, 2  # not executed
addi $s1, $s1, 1  # not executed
sub $s1, $s1, $s0  # not executed

target:
add $s1, $s1, $s0  # $s1 = 1 + 4 = 5
```
# MIPS assembly

```assembly
0x00002000  addi $s0, $0, 0x2010
0x00002004  jr  $s0
0x00002008  addi $s1, $0, 1
0x0000200C  sra $s1, $s1, 2
0x00002010  lw  $s3, 44($s1)
```

jr is an **R-type** instruction.
High-Level Code Constructs

- if statements
- if/else statements
- while loops
- for loops
C Code

if (i == j)
    f = g + h;

f = f - i;

MIPS assembly code

# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

If Statement
If Statement

C Code

```c
if (i == j)
    f = g + h;

f = f - i;
```

MIPS assembly code

```mips
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
L1: sub $s0, $s0, $s3
```

Assembly tests opposite case (i != j) of high-level code (i == j)
If/Else Statement

C Code

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

MIPS assembly code
If/Else Statement

C Code

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

MIPS assembly code

```assembly
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
j done
L1:   sub $s0, $s0, $s3
done:
```
C Code

// determines the power
// of x such that $2^x = 128$
int pow = 1;
int x   = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}

MIPS assembly code

Assembly tests for the opposite case ($\text{pow} == 128$) of the C code ($\text{pow} != 128$).
While Loops

C Code

// determines the power
// of x such that 2^x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}

MIPS assembly code

# $s0 = pow, $s1 = x
addi $s0, $0, 1
add $s1, $0, $0
addi $t0, $0, 128
while: beq $s0, $t0, done
sll $s0, $s0, 1
addi $s1, $s1, 1
j while
done:

Assembly tests for the opposite case (pow == 128) of the C code (pow != 128).
For Loops

for (initialization; condition; loop operation)
statement

- **initialization**: executes before the loop begins
- **condition**: is tested at the beginning of each iteration
- **loop operation**: executes at the end of each iteration
- **statement**: executes each time the condition is met
For Loops

High-level code

```c
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```

Chapter 6 <70>
C Code

// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}

MIPS assembly code
For Loops

C Code

```c
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

MIPS assembly code

```mips
# $s0 = i, $s1 = sum
addi $s1, $0, 0
add  $s0, $0, $0
addi $t0, $0, 10
for:  beq  $s0, $t0, done
      add  $s1, $s1, $s0
      addi $s0, $s0, 1
      j     for
done:
```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
C Code

// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}

MIPS assembly code

# $s0 = i, $s1 = sum
addi $s1, $0, 0
addi $s0, $0, 1
addi $t0, $0, 101

loop:  slt  $t1, $s0, $t0
beq   $t1, $0, done
add   $s1, $s1, $s0
sll   $s0, $s0, 1
j     loop

done:

$\textbf{t1} = 1 \textit{ if } i < 101
Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements
Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register

<table>
<thead>
<tr>
<th>Address</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12340010</td>
<td>array[4]</td>
</tr>
<tr>
<td>0x1234800C</td>
<td>array[3]</td>
</tr>
<tr>
<td>0x12348008</td>
<td>array[2]</td>
</tr>
<tr>
<td>0x12348004</td>
<td>array[1]</td>
</tr>
<tr>
<td>0x12348000</td>
<td>array[0]</td>
</tr>
</tbody>
</table>
Accessing Arrays

// C Code

int array[5];
array[0] = array[0] * 2;
// C Code
int array[5];
array[0] = array[0] * 2;

# MIPS assembly code
# $s0 = array base address
lui $s0, 0x1234 # 0x1234 in upper half of $s0
ori $s0, $s0, 0x8000 # 0x8000 in lower half of $s0

lw $t1, 0($s0) # $t1 = array[0]
sll $t1, $t1, 1 # $t1 = $t1 * 2
sw $t1, 0($s0) # array[0] = $t1

lw $t1, 4($s0) # $t1 = array[1]
sll $t1, $t1, 1 # $t1 = $t1 * 2
sw $t1, 4($s0) # array[1] = $t1
// C Code
int array[1000];
int i;

for (i=0; i < 1000; i = i + 1)
    array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
# MIPS assembly code

# $s0 = array base address, $s1 = i

# initialization code

lui $s0, 0x23B8     # $s0 = 0x23B80000
ori $s0, $s0, 0xF000 # $s0 = 0x23B8F000
addi $s1, $0, 0     # i = 0
addi $t2, $0, 1000   # $t2 = 1000

loop:

slt $t0, $s1, $t2    # i < 1000?
b eq $t0, $0, done   # if not then done
sll $t0, $s1, 2     # $t0 = i * 4 (byte offset)
add $t0, $t0, $s0    # address of array[i]
lw $t1, 0($t0)       # $t1 = array[i]
sll $t1, $t1, 3     # $t1 = array[i] * 8
sw $t1, 0($t0)       # array[i] = array[i] * 8
addi $s1, $s1, 1    # i = i + 1
j loop               # repeat

done:
American Standard Code for Information Interchange

Each text character has unique byte value

- For example, S = 0x53, a = 0x61, A = 0x41
- Lower-case and upper-case differ by 0x20 (32)
# Cast of Characters

<table>
<thead>
<tr>
<th>#</th>
<th>Char</th>
<th>#</th>
<th>Char</th>
<th>#</th>
<th>Char</th>
<th>#</th>
<th>Char</th>
<th>#</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>space</td>
<td>30</td>
<td>0</td>
<td>40</td>
<td>@</td>
<td>50</td>
<td>P</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>21</td>
<td>!</td>
<td>31</td>
<td>1</td>
<td>41</td>
<td>A</td>
<td>51</td>
<td>Q</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>22</td>
<td>&quot;</td>
<td>32</td>
<td>2</td>
<td>42</td>
<td>B</td>
<td>52</td>
<td>R</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>23</td>
<td>#</td>
<td>33</td>
<td>3</td>
<td>43</td>
<td>C</td>
<td>53</td>
<td>S</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>24</td>
<td>$</td>
<td>34</td>
<td>4</td>
<td>44</td>
<td>D</td>
<td>54</td>
<td>T</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>25</td>
<td>%</td>
<td>35</td>
<td>5</td>
<td>45</td>
<td>E</td>
<td>55</td>
<td>U</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>26</td>
<td>&amp;</td>
<td>36</td>
<td>6</td>
<td>46</td>
<td>F</td>
<td>56</td>
<td>V</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>27</td>
<td>'</td>
<td>37</td>
<td>7</td>
<td>47</td>
<td>G</td>
<td>57</td>
<td>W</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>28</td>
<td>(</td>
<td>38</td>
<td>8</td>
<td>48</td>
<td>H</td>
<td>58</td>
<td>X</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>29</td>
<td>)</td>
<td>39</td>
<td>9</td>
<td>49</td>
<td>I</td>
<td>59</td>
<td>Y</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>2A</td>
<td>*</td>
<td>3A</td>
<td>:</td>
<td>4A</td>
<td>J</td>
<td>5A</td>
<td>Z</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>2B</td>
<td>+</td>
<td>3B</td>
<td>:</td>
<td>4B</td>
<td>K</td>
<td>5B</td>
<td>[</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>2C</td>
<td>,</td>
<td>3C</td>
<td>&lt;</td>
<td>4C</td>
<td>L</td>
<td>5C</td>
<td>\</td>
<td>6C</td>
<td>1</td>
</tr>
<tr>
<td>2D</td>
<td>-</td>
<td>3D</td>
<td>=</td>
<td>4D</td>
<td>M</td>
<td>5D</td>
<td>]</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>2E</td>
<td>.</td>
<td>3E</td>
<td>&gt;</td>
<td>4E</td>
<td>N</td>
<td>5E</td>
<td>^</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>2F</td>
<td>/</td>
<td>3F</td>
<td>?</td>
<td>4F</td>
<td>0</td>
<td>5F</td>
<td>_</td>
<td>6F</td>
<td>o</td>
</tr>
</tbody>
</table>
Function Calls

• **Caller:** calling function (in this case, `main`)
• **Callee:** called function (in this case, `sum`)

**C Code**

```c
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```
Function Conventions

• **Caller:**
  – passes **arguments** to callee
  – jumps to callee

• **Callee:**
  – **performs** the function
  – **returns** result to caller
  – **returns** to point of call
  – **must not overwrite** registers or memory needed by caller
MIPS Function Conventions

- **Call Function**: jump and link (`jal`)
- **Return from function**: jump register (`jr`)
- **Arguments**: `$a0 - $a3`
- **Return value**: `$v0`
C Code

```c
int main() {
    simple();
    a = b + c;
}

void simple() {
    return;
}
```

MIPS assembly code

```assembly
0x00400200 main: jal simple
0x00400204 add $s0, $s1, $s2
...
0x00401020 simple: jr $ra
```

**void means that simple doesn’t return a value**
Function Calls

C Code
int main() {
    simple();
    a = b + c;
}

void simple() {
    return;
}

MIPS assembly code
0x00400200 main: jal simple
0x00400204 add $s0, $s1, $s2
...

0x00401020 simple: jr $ra

jal: jumps to simple
$ra = PC + 4 = 0x00400204

jr $ra: jumps to address in $ra (0x00400204)
MIPS conventions:

- Argument values: $a0 - $a3
- Return value: $v0
C Code

```c
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5);  // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;               // return value
}
```
MIPS assembly code

# $s0 = y

main:

...  
  addi $a0, $0, 2    # argument 0 = 2  
  addi $a1, $0, 3    # argument 1 = 3  
  addi $a2, $0, 4    # argument 2 = 4  
  addi $a3, $0, 5    # argument 3 = 5  
  jal diffofsums    # call Function  
  add $s0, $v0, $0  # y = returned value  
...

# $s0 = result  

diffofsums:  
  add $t0, $a0, $a1    # $t0 = f + g  
  add $t1, $a2, $a3    # $t1 = h + i  
  sub $s0, $t0, $t1    # result = (f + g) - (h + i)  
  add $v0, $s0, $0    # put return value in $v0  
  jr $ra               # return to caller
MIPS assembly code

# $s0 = result
diffofsums:
    add $t0, $a0, $a1  # $t0 = f + g
    add $t1, $a2, $a3  # $t1 = h + i
    sub $s0, $t0, $t1  # result = (f + g) - (h + i)
    add $v0, $s0, $0   # put return value in $v0
    jr $ra              # return to caller

• diffofsums overwrote 3 registers: $t0, $t1, $s0
• diffofsums can use stack to temporarily store registers
The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- *Expands*: uses more memory when more space needed
- *Contracts*: uses less memory when the space is no longer needed
The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: $sp$ points to top of the stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFFC</td>
<td>12345678</td>
</tr>
<tr>
<td>7FFFFFFF8</td>
<td></td>
</tr>
<tr>
<td>7FFFFFFF4</td>
<td></td>
</tr>
<tr>
<td>7FFFFFFF0</td>
<td></td>
</tr>
<tr>
<td>$sp$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFFC</td>
<td>12345678</td>
</tr>
<tr>
<td>7FFFFFFF8</td>
<td>AABBCDD</td>
</tr>
<tr>
<td>7FFFFFFF4</td>
<td>11223344</td>
</tr>
<tr>
<td>7FFFFFFF0</td>
<td>$sp$</td>
</tr>
</tbody>
</table>
How Functions use the Stack

- Called functions must have no unintended side effects
- **But** `diffofsums` overwrites 3 registers: `$t0$, `$t1$, `$s0$

```mips
# MIPS assembly
# $s0 = result
diffofsums:
    add $t0, $a0, $a1  # $t0 = f + g
    add $t1, $a2, $a3  # $t1 = h + i
    sub $s0, $t0, $t1  # result = (f + g) - (h + i)
    add $v0, $s0, $0   # put return value in $v0
    jr $ra             # return to caller
```
# $s0 = result
diffofsums:

```
addi $sp, $sp, -12  # make space on stack
    # to store 3 registers
sw  $s0, 8($sp)     # save $s0 on stack
sw  $t0, 4($sp)     # save $t0 on stack
sw  $t1, 0($sp)     # save $t1 on stack
add  $t0, $a0, $a1  # $t0 = f + g
add  $t1, $a2, $a3  # $t1 = h + i
sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
add  $v0, $s0, $0   # put return value in $v0
lw  $t1, 0($sp)     # restore $t1 from stack
lw  $t0, 4($sp)     # restore $t0 from stack
lw  $s0, 8($sp)     # restore $s0 from stack
addi $sp, $sp, 12   # deallocate stack space
jr  $ra             # return to caller
```
The stack during \texttt{diffofsums} Call

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>?</td>
</tr>
<tr>
<td>F8</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td></td>
</tr>
</tbody>
</table>

$\texttt{sp}$

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>?</td>
</tr>
<tr>
<td>F8</td>
<td>$s0$</td>
</tr>
<tr>
<td>F4</td>
<td>$t0$</td>
</tr>
<tr>
<td>F0</td>
<td>$t1$</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>stack frame</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td></td>
</tr>
</tbody>
</table>

$\texttt{sp}$

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>?</td>
</tr>
<tr>
<td>F8</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
</tr>
</tbody>
</table>

$\texttt{sp}$
## Registers

<table>
<thead>
<tr>
<th>Preserved</th>
<th>Nonpreserved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Callee-Saved</strong></td>
<td><strong>Caller-Saved</strong></td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>$t0-$t9</td>
</tr>
<tr>
<td>$ra</td>
<td>$a0-$a3</td>
</tr>
<tr>
<td>$sp</td>
<td>$v0-$v1</td>
</tr>
<tr>
<td><strong>stack above $sp</strong></td>
<td><strong>stack below $sp</strong></td>
</tr>
</tbody>
</table>
Multiple Function Calls

procl:

```
addi $sp, $sp, -4  # make space on stack
sw $ra, 0($sp)    # save $ra on stack
jal proc2
...      
lw $ra, 0($sp)   # restore $s0 from stack
addi $sp, $sp, 4 # deallocate stack space
jr  $ra          # return to caller
```
# $s0 = result
diffofsums:

\[
\text{addi } \$s0, \$sp, -4 \quad \# \text{make space on stack to store one register}
\]

\[
\text{sw } \$s0, 0(\$sp) \quad \# \text{save } \$s0 \text{ on stack}
\]

\[
\text{add } \$t0, \$a0, \$a1 \quad \# \text{no need to save } \$t0 \text{ or } \$t1
\]

\[
\text{add } \$t1, \$a2, \$a3 \quad \# \text{result} = (f + g) - (h + i)
\]

\[
\text{sub } \$s0, \$t0, \$t1 \quad \# \text{put return value in } \$v0
\]

\[
\text{add } \$v0, \$s0, \$0 \quad \# \text{restore } \$s0 \text{ from stack}
\]

\[
\text{lw } \$s0, 0(\$sp) \quad \# \text{deallocate stack space}
\]

\[
\text{addi } \$sp, \$sp, 4 \quad \# \text{return to caller}
\]

\[
\text{jr } \$ra
\]
High-level code

```c
int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n-1));
}
```
Recursive Function Call

MIPS assembly code
MIPS assembly code

0x90 factorial: addi $sp, $sp, -8  # make room
0x94          sw  $a0, 4($sp)     # store $a0
0x98          sw  $ra, 0($sp)     # store $ra
0x9C          addi $t0, $0, 2
0xA0          slt $t0, $a0, $t0  # a <= 1 ?
0xA4          beq $t0, $0, else  # no: go to else
0xA8          addi $v0, $0, 1   # yes: return 1
0xAC          addi $sp, $sp, 8   # restore $sp
0xB0          jr  $ra           # return
0xB4 else: addi $a0, $a0, -1  # n = n - 1
0xB8          jal factorial     # recursive call
0xBC          lw  $ra, 0($sp)    # restore $ra
0xC0          lw  $a0, 4($sp)    # restore $a0
0xC4          addi $sp, $sp, 8   # restore $sp
0xC8          mul $v0, $a0, $v0 # n * factorial(n-1)
0xCC          jr  $ra           # return

Recursive Function Call
### Stack During Recursive Call

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FC</td>
<td></td>
<td>F8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>F8</td>
<td>$a0 (0x3)</td>
<td>F4</td>
<td>$ra</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F4</td>
<td>$ra</td>
<td>F0</td>
<td>$a0 (0x2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EC</td>
<td>$ra (0xBC)</td>
<td>E8</td>
<td>$a0 (0x1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E8</td>
<td>$a0 (0x1)</td>
<td>E4</td>
<td>$ra (0xBC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E4</td>
<td>$ra (0xBC)</td>
<td>E0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>E0</td>
<td></td>
<td>DC</td>
<td></td>
</tr>
</tbody>
</table>

- $sp$ points to the current stack pointer.
- $v0$ is updated with the product of $a0$ and $v0$ at each recursive call.

Starting values:
- $a0 = 1$
- $v0 = 1$

Recursive calls:
1. $a0 = 2$, $v0 = 2 \times 1$
2. $a0 = 3$, $v0 = 3 \times 2$
3. $a0 = 4$, $v0 = 4 \times 3$

Final values:
- $a0 = 1$
- $v0 = 6$
Function Call Summary

• **Caller**
  
  – Put arguments in $a0-$a3
  – Save any needed registers ($ra, maybe $t0-t9)
  – jal callee
  – Restore registers
  – Look for result in $v0

• **Callee**

  – Save registers that might be disturbed ($s0-$s7)
  – Perform function
  – Put result in $v0
  – Restore registers
  – jr $ra
How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct
Register Only

- Operands found in registers
  - Example: `add $s0, $t2, $t3`
  - Example: `sub $t8, $s1, $0`

Immediate

- 16-bit immediate used as an operand
  - Example: `addi $s4, $t5, -73`
  - Example: `ori $t3, $t7, 0xFF`
Base Addressing

- Address of operand is:
  \[ \text{base address + sign-extended immediate} \]

- **Example:** `lw  $s4, 72($0)`
  - `address = $0 + 72`

- **Example:** `sw  $t2, -25($t1)`
  - `address = $t1 - 25`
Addressing Modes

PC-Relative Addressing

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>beq $t0, $0, else</td>
</tr>
<tr>
<td>0x14</td>
<td>addi $v0, $0, 1</td>
</tr>
<tr>
<td>0x18</td>
<td>addi $sp, $sp, i</td>
</tr>
<tr>
<td>0x1C</td>
<td>jr $ra</td>
</tr>
<tr>
<td>0x20</td>
<td>else: addi $a0, $a0, -1</td>
</tr>
<tr>
<td>0x24</td>
<td>jal factorial</td>
</tr>
</tbody>
</table>

Assembly Code

```
beq $t0, $0, else
```

Field Values

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>4</td>
</tr>
<tr>
<td>rs</td>
<td>8</td>
</tr>
<tr>
<td>rt</td>
<td>0</td>
</tr>
<tr>
<td>imm</td>
<td>3</td>
</tr>
</tbody>
</table>

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits
Addressing Modes

Pseudo-direct Addressing

0x0040005C  jal  sum
...
0x004000A0  sum:  add $v0, $a0, $a1

JTA  0000 0000 0100 0000 0000 0000 1010 0000  (0x004000A0)
26-bit addr  0000 0000 0100 0000 0000 0000 0000 0000  1010 0000  (0x0100028)

Field Values

<table>
<thead>
<tr>
<th>op</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0x0100028</td>
</tr>
</tbody>
</table>
6 bits 26 bits

Machine Code

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>000011</td>
<td>00 0001 0000 0000 0000 0010 1000</td>
</tr>
</tbody>
</table>
(0xC100028)
6 bits 26 bits
How to Compile & Run a Program

High Level Code
  ↓
  Compiler
  ↓
Assembly Code
  ↓
Assembler
  ↓
Object File
  ↓
Linker
  ↓
Executable
  ↓
Loader
  ↓
Memory

Object Files

Library Files
Grace Hopper, 1906-1992

• Graduated from Yale University with a Ph.D. in mathematics
• Developed first compiler
• Helped develop the COBOL programming language
• Highly awarded naval officer
• Received World War II Victory Medal and National Defense Service Medal, among others
What is Stored in Memory?

- Instructions (also called text)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program
- How big is memory?
  - At most $2^{32} = 4$ gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF
int f, g, y; // global variables

int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);

    return y;
}

int sum(int a, int b) {
    return (a + b);
}
int f, g, y; // global

int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);
    return y;
}

int sum(int a, int b) {
    return (a + b);
}
Example Program: Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Program: Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>0x10000000</td>
</tr>
<tr>
<td>g</td>
<td>0x10000004</td>
</tr>
<tr>
<td>y</td>
<td>0x10000008</td>
</tr>
<tr>
<td>main</td>
<td>0x00400000</td>
</tr>
<tr>
<td>sum</td>
<td>0x0040002C</td>
</tr>
</tbody>
</table>
## Example Program: Executable

<table>
<thead>
<tr>
<th>Executable file header</th>
<th>Text Size</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x34 (52 bytes)</td>
<td>0xC (12 bytes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>0x23BDFFFC</td>
<td>addi $sp, $sp, -4</td>
</tr>
<tr>
<td>0x00400004</td>
<td>0xAFBF0000</td>
<td>sw $ra, 0 ($sp)</td>
</tr>
<tr>
<td>0x00400008</td>
<td>0x20040002</td>
<td>addi $a0, $0, 2</td>
</tr>
<tr>
<td>0x0040000C</td>
<td>0xAF848000</td>
<td>sw $a0, 0x8000 ($gp)</td>
</tr>
<tr>
<td>0x00400010</td>
<td>0x20050003</td>
<td>addi $a1, $0, 3</td>
</tr>
<tr>
<td>0x00400014</td>
<td>0xAF858004</td>
<td>sw $a1, 0x8004 ($gp)</td>
</tr>
<tr>
<td>0x00400018</td>
<td>0xC10000B</td>
<td>jal 0x0040002C</td>
</tr>
<tr>
<td>0x0040001C</td>
<td>0xAF828008</td>
<td>sw $v0, 0x8008 ($gp)</td>
</tr>
<tr>
<td>0x00400020</td>
<td>0x8FBF0000</td>
<td>lw $ra, 0 ($sp)</td>
</tr>
<tr>
<td>0x00400024</td>
<td>0x23BD0004</td>
<td>addi $sp, $sp, -4</td>
</tr>
<tr>
<td>0x00400028</td>
<td>0x03E00008</td>
<td>jr $ra</td>
</tr>
<tr>
<td>0x0040002C</td>
<td>0x00851020</td>
<td>add $v0, $a0, $a1</td>
</tr>
<tr>
<td>0x00400030</td>
<td>0x03E00008</td>
<td>jr $ra</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data segment</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x10000000</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>0x10000004</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>0x10000008</td>
<td>y</td>
</tr>
</tbody>
</table>
### Example Program: In Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFC</td>
<td>Stack</td>
</tr>
<tr>
<td>0x10010000</td>
<td>Heap</td>
</tr>
<tr>
<td>0x10000000</td>
<td></td>
</tr>
<tr>
<td>0x03E00008</td>
<td></td>
</tr>
<tr>
<td>0x00851020</td>
<td></td>
</tr>
<tr>
<td>0x03E00008</td>
<td></td>
</tr>
<tr>
<td>0x23BD0004</td>
<td></td>
</tr>
<tr>
<td>0x8FBF0000</td>
<td></td>
</tr>
<tr>
<td>0xAF828008</td>
<td></td>
</tr>
<tr>
<td>0x10008000</td>
<td></td>
</tr>
<tr>
<td>0x0C1000B</td>
<td></td>
</tr>
<tr>
<td>0xAF858004</td>
<td></td>
</tr>
<tr>
<td>0x02050003</td>
<td></td>
</tr>
<tr>
<td>0xAF848000</td>
<td></td>
</tr>
<tr>
<td>0x02040002</td>
<td></td>
</tr>
<tr>
<td>0xAFBF0000</td>
<td></td>
</tr>
<tr>
<td>0x00400000</td>
<td></td>
</tr>
<tr>
<td>0x03E00008</td>
<td></td>
</tr>
<tr>
<td>0x00851020</td>
<td></td>
</tr>
<tr>
<td>0x03E00008</td>
<td></td>
</tr>
<tr>
<td>0x23BD0004</td>
<td></td>
</tr>
<tr>
<td>0x8FBF0000</td>
<td></td>
</tr>
<tr>
<td>0xAF828008</td>
<td></td>
</tr>
<tr>
<td>0x10008000</td>
<td></td>
</tr>
<tr>
<td>0x0C1000B</td>
<td></td>
</tr>
<tr>
<td>0xAF858004</td>
<td></td>
</tr>
<tr>
<td>0x02050003</td>
<td></td>
</tr>
<tr>
<td>0xAF848000</td>
<td></td>
</tr>
<tr>
<td>0x02040002</td>
<td></td>
</tr>
<tr>
<td>0xAFBF0000</td>
<td></td>
</tr>
<tr>
<td>0x00400000</td>
<td></td>
</tr>
</tbody>
</table>

- $sp = 0x7FFFFFFC
- $gp = 0x10008000
- PC = 0x00400000
Odds & Ends

- Pseudoinstructions
- Exceptions
- Signed and unsigned instructions
- Floating-point instructions
## Pseudoinstructions

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>MIPS Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>li $s0, 0x1234AA77</td>
<td>lui $s0, 0x1234</td>
</tr>
<tr>
<td></td>
<td>ori $s0, 0xAA77</td>
</tr>
<tr>
<td>clear $t0</td>
<td>add $t0, $0, $0</td>
</tr>
<tr>
<td>move $s1, $s2</td>
<td>add $s2, $s1, $0</td>
</tr>
<tr>
<td>nop</td>
<td>sll $0, $0, 0</td>
</tr>
</tbody>
</table>
Exceptions

• Unscheduled function call to *exception handler*

• Caused by:
  – Hardware, also called an *interrupt*, e.g., keyboard
  – Software, also called *traps*, e.g., undefined instruction

• When exception occurs, the processor:
  – Records the cause of the exception
  – Jumps to exception handler (at instruction address 0x80000180)
  – Returns to program
Exception Registers

• Not part of register file
  – **Cause**: Records cause of exception
  – **EPC** (Exception PC): Records PC where exception occurred

• **EPC and Cause**: part of Coprocessor 0

• Move from Coprocessor 0
  – `mfc0 $k0, EPC`
  – Moves contents of **EPC** into **$k0**
## Exception Causes

<table>
<thead>
<tr>
<th>Exception</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Interrupt</td>
<td>0x00000000</td>
</tr>
<tr>
<td>System Call</td>
<td>0x00000020</td>
</tr>
<tr>
<td>Breakpoint / Divide by 0</td>
<td>0x00000024</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>0x00000028</td>
</tr>
<tr>
<td>Arithmetic Overflow</td>
<td>0x00000030</td>
</tr>
</tbody>
</table>
Exception Flow

- Processor saves cause and exception PC in Cause and EPC
- Processor jumps to exception handler (0x80000180)
- Exception handler:
  - Saves registers on stack
  - Reads Cause register
    \[
    \text{mfc0 $k0, Cause}
    \]
  - Handles exception
  - Restores registers
  - Returns to program
    \[
    \text{mfc0 $k0, EPC}
    \]
    \[
    \text{jr $k0}
    \]
Signed & Unsigned Instructions

- Addition and subtraction
- Multiplication and division
- Set less than
Addition & Subtraction

- **Signed**: add, addi, sub
  - Same operation as unsigned versions
  - But processor takes exception on overflow
- **Unsigned**: addu, addiu, subu
  - Doesn’t take exception on overflow

**Note**: addiu sign-extends the immediate
• **Signed**: `mult, div`
• **Unsigned**: `multu, divu`
Set Less Than

- **Signed**: `slt, slti`
- **Unsigned**: `sltu, sltiu`

**Note**: `sltiu` sign-extends the immediate before comparing it to the register
Loads

• **Signed:**
  - Sign-extends to create 32-bit value to load into register
  - Load halfword: \( lh \)
  - Load byte: \( lb \)

• **Unsigned:**
  - Zero-extends to create 32-bit value
  - Load halfword unsigned: \( lhu \)
  - Load byte: \( lbu \)
Floating-Point Instructions

- Floating-point coprocessor (Coprocessor 1)
- 32 32-bit floating-point registers ($f0$-$f31$)
- Double-precision values held in two floating point registers
  - e.g., $f0$ and $f1$, $f2$ and $f3$, etc.
  - Double-precision floating point registers: $f0$, $f2$, $f4$, etc.
### Floating-Point Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fv0 - $fv1</td>
<td>0, 2</td>
<td>return values</td>
</tr>
<tr>
<td>$ft0 - $ft3</td>
<td>4, 6, 8, 10</td>
<td>temporary variables</td>
</tr>
<tr>
<td>$fa0 - $fa1</td>
<td>12, 14</td>
<td>Function arguments</td>
</tr>
<tr>
<td>$ft4 - $ft8</td>
<td>16, 18</td>
<td>temporary variables</td>
</tr>
<tr>
<td>$fs0 - $fs5</td>
<td>20, 22, 24, 26, 28, 30</td>
<td>saved variables</td>
</tr>
</tbody>
</table>
F-Type Instruction Format

- **Op code** = \(17\ (010001_2)\)

- **Single-precision:**
  - \(\text{cop} = 16\ (010000_2)\)
  - \(\text{add.s}, \text{sub.s}, \text{div.s}, \text{neg.s}, \text{abs.s}, \text{etc.}\)

- **Double-precision:**
  - \(\text{cop} = 17\ (010001_2)\)
  - \(\text{add.d}, \text{sub.d}, \text{div.d}, \text{neg.d}, \text{abs.d}, \text{etc.}\)

- **3 register operands:**
  - \(\text{fs, ft: source operands}\)
  - \(\text{fd: destination operands}\)

<table>
<thead>
<tr>
<th>op</th>
<th>cop</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

\[\text{F-type}_{-\text{Type}}\]
Floating-Point Branches

• Set/clear condition flag: `fpcond`
  – Equality: `c.seq.s, c.seq.d`
  – Less than: `c.lt.s, c.lt.d`
  – Less than or equal: `c.le.s, c.le.d`

• Conditional branch
  – `bclf`: branches if `fpcond` is FALSE
  – `bclt`: branches if `fpcond` is TRUE

• Loads and stores
  – `lwcl1:lwcl1 $ft1, 42($s1)`
  – `swcl1:swcl1 $fs2, 17($sp)`
Looking Ahead

Microarchitecture – building MIPS processor in hardware

Bring colored pencils