Chapter 6 :: Architecture

Digital Design and Computer Architecture
David Money Harris and Sarah L. Harris

Chapter 6 :: Topics

• Introduction
• Assembly Language
• Machine Language
• Programming
• Addressing Modes
• Lights, Camera, Action: Compiling, Assembling, and Loading
• Odds and Ends

Introduction

- Jumping up a few levels of abstraction.
- **Architecture**: the programmer’s view of the computer
  - Defined by instructions (operations) and operand locations
- **Microarchitecture**: how to implement an architecture in hardware (covered in Chapter 7)

Assembly Language

- To command a computer, you must understand its language.
  - **Instructions**: words in a computer’s language
  - **Instruction set**: the vocabulary of a computer’s language
- Instructions indicate the operation to perform and the operands to use.
  - **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 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)
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

Architecture Design Principles

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

- **add**: mnemonic indicates what operation to perform
- **b, c**: source operands on which the operation is performed
- **a**: destination operand to which the result is written

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b + c;</td>
<td>add a, b, c</td>
</tr>
</tbody>
</table>

Instructions: Subtraction

- **sub**: mnemonic indicates what operation to perform
- **b, c**: source operands on which the operation is performed
- **a**: destination operand to which the result is written

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b - c;</td>
<td>sub a, b, c</td>
</tr>
</tbody>
</table>
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

Instructions: More Complex Code

- More complex code is handled by multiple MIPS instructions.

High-level code
\[
a = b + c - d;  
// single line comment
/* multiple line
  comment */
\]

MIPS assembly code
\[
add t, b, c \ # t = b + c
sub a, t, d \ # a = t - d
\]

Design Principle 2

Make the common case fast
- MIPS includes only simple, commonly used instructions.
- Hardware to decode and execute the instruction can be simple, small, and fast.
- More complex instructions (that are less common) can be 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 IA-32 found in many PC’s, are complex instruction set computers (CISC). They include complex instructions that are rarely used, such as the “string move” instruction that copies a string (a series of characters) from one part of memory to another.

Operands

- A computer needs a physical location from which to retrieve binary operands
- A computer retrieves operands from:
  - Registers
  - Memory
  - Constants (also called immediates)
Operands: Registers

- Memory is slow.
- Most architectures have a small set of (fast) registers.
- MIPS has thirty-two 32-bit registers.
- MIPS is called a 32-bit architecture because it operates on 32-bit data.

(A 64-bit version of MIPS also exists, but we will consider only MIPS32.)

Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers
- Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.

The 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>procedure return values</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>procedure 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>procedure return address</td>
</tr>
</tbody>
</table>

Operands: Registers

- Registers:
  - Written with a dollar sign ($$) before their name
  - For example, register 0 is written “$0”, pronounced “register zero” or “dollar zero”.
- Certain registers used for specific purposes:
  - For example,
    - $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.
- For now, we only use the temporary registers ($t0 - $t9) and the saved registers ($s0 - $s7).
- We will use the other registers in later slides.
Instructions with registers

- Revisit add instruction

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b + c )</td>
<td># $s0 = a, $s1 = b, $s2 = c</td>
</tr>
<tr>
<td></td>
<td>add $s0, $s1, $s2</td>
</tr>
</tbody>
</table>

Copyright © 2007 Elsevier

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, so it can hold a lot of data
- But it’s also slow
- Commonly used variables kept in registers
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly

Copyright © 2007 Elsevier

Word-Addressable Memory

- Each 32-bit data word has a unique address

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td></td>
</tr>
<tr>
<td>00000002</td>
<td></td>
</tr>
<tr>
<td>00000003</td>
<td>40F30788 Word 3</td>
</tr>
<tr>
<td>00000004</td>
<td>01EE2842 Word 2</td>
</tr>
<tr>
<td>00000005</td>
<td>F2F1AC07 Word 1</td>
</tr>
<tr>
<td>00000006</td>
<td>ABCDEF78 Word 0</td>
</tr>
</tbody>
</table>

Copyright © 2007 Elsevier

Reading Word-Addressable Memory

- Memory reads are called loads
- Mnemonic: load word (lw)
- Example: read a word of data at memory address 1 into $s3
- Memory address calculation:
  - add the base address ($0 + 1) to the offset (1)
  - address = ($0 + 1) = 1
- Any register may be used to store the base address.
- $s3 holds the value 0xF2F1AC07 after the instruction completes.

Assembly code

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

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td></td>
</tr>
<tr>
<td>00000002</td>
<td></td>
</tr>
<tr>
<td>00000003</td>
<td>40F30788 Word 3</td>
</tr>
<tr>
<td>00000004</td>
<td>01EE2842 Word 2</td>
</tr>
<tr>
<td>00000005</td>
<td>F2F1AC07 Word 1</td>
</tr>
<tr>
<td>00000006</td>
<td>ABCDEF78 Word 0</td>
</tr>
</tbody>
</table>
Writing Word-Addressable Memory

- Memory writes are called stores.
- Mnemonic: store word (sw)
- Example: Write (store) the value held in $t4 into memory address 7
- Offset can be written in decimal (default) or hexadecimal
- Memory address calculation:
  - add the base address ($0) to the offset (0x7)
  - address: ($0 + 0x7) = 7
- Any register may be used to store the base address

Assembly code

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

Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)
- Load a word of data at memory address 4 into $s3.
- MIPS is byte-addressed, not word-addressed

MIPS assembly code

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

Writing Byte-Addressable Memory

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

MIPS assembly code

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

Byte-Addressable Memory

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

- How to number bytes within a word?
- Word address is the same for big- or little-endian
- Little-endian: byte numbers start at the little (least significant) end
- Big-endian: byte numbers start at the big (most significant) end

<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>C D E F</td>
<td>C F E D C</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8 B A 9 8</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4 7 6 5 4</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 3 2 1 0</td>
</tr>
</tbody>
</table>

Big- and Little-Endian Example

- Suppose $t0$ initially contains 0x23456789. After the following program is run on a big-endian system, what value does $s0$ contain? In a little-endian system?
  - sw $t0$, 0($0)
  - lb $s0$, 1($0)

  - Big-endian: 0x00000045
  - Little-endian: 0x00000067
Design Principle 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).

Operands: Constants/Immediates
- `lw` and `sw` illustrate the use of constants or immediates
- Called immediates because they are immediately available from the instruction
- Immediates don’t require a register or memory access.
- The add immediate (`addi`) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two’s complement number.
- Is subtract immediate (`subi`) necessary?

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = a + 4;</code></td>
<td><code>add $s0, $s0, 4</code></td>
</tr>
<tr>
<td><code>b = a - 12;</code></td>
<td><code>addi $s1, $s0, -12</code></td>
</tr>
</tbody>
</table>

Machine Language
- Computers only understand 1’s and 0’s
- Machine language: binary representation of instructions
- 32-bit instructions
  - Again, simplicity favors regularity: 32-bit data and instructions
- Three instruction formats:
  - `R-Type`: register operands
  - `I-Type`: immediate operand
  - `J-Type`: for jumping (we’ll 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
  - together, the opcode and function tell the computer what operation to perform
  - `shamt`: the shift amount for shift instructions, otherwise it’s 0

<table>
<thead>
<tr>
<th>R-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
</tr>
<tr>
<td>6 bits</td>
</tr>
</tbody>
</table>
### R-Type Examples

#### Assembly Code

- `add $s0, $s1, $s2`
- `sub $t0, $t3, $t5`

#### Field Values

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>0</td>
</tr>
<tr>
<td>rs</td>
<td>17</td>
</tr>
<tr>
<td>rt</td>
<td>18</td>
</tr>
<tr>
<td>rd</td>
<td>16</td>
</tr>
<tr>
<td>shamt</td>
<td>0</td>
</tr>
<tr>
<td>funct</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>0</td>
</tr>
<tr>
<td>rs</td>
<td>11</td>
</tr>
<tr>
<td>rt</td>
<td>13</td>
</tr>
<tr>
<td>rd</td>
<td>8</td>
</tr>
<tr>
<td>shamt</td>
<td>0</td>
</tr>
<tr>
<td>funct</td>
<td>34</td>
</tr>
</tbody>
</table>

#### Machine Code

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>000000</td>
</tr>
<tr>
<td>rs</td>
<td>100001</td>
</tr>
<tr>
<td>rt</td>
<td>100101</td>
</tr>
<tr>
<td>rd</td>
<td>100000</td>
</tr>
<tr>
<td>shamt</td>
<td>000000</td>
</tr>
<tr>
<td>funct</td>
<td>100010</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 the opcode

### I-Type Examples

#### Assembly Code

- `addi $s0, $s1, 5`
- `addi $t0, $s3, -12`
- `lw $t2, 32($0)`
- `sw $s1, 4($t1)`

#### Field Values

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>8</td>
</tr>
<tr>
<td>rs</td>
<td>17</td>
</tr>
<tr>
<td>rt</td>
<td>16</td>
</tr>
<tr>
<td>imm</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>8</td>
</tr>
<tr>
<td>rs</td>
<td>19</td>
</tr>
<tr>
<td>rt</td>
<td>8</td>
</tr>
<tr>
<td>imm</td>
<td>-12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>35</td>
</tr>
<tr>
<td>rs</td>
<td>6</td>
</tr>
<tr>
<td>rt</td>
<td>10</td>
</tr>
<tr>
<td>imm</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>43</td>
</tr>
<tr>
<td>rs</td>
<td>9</td>
</tr>
<tr>
<td>rt</td>
<td>17</td>
</tr>
<tr>
<td>imm</td>
<td>4</td>
</tr>
</tbody>
</table>

#### Machine Code

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>001000</td>
</tr>
<tr>
<td>rs</td>
<td>100000</td>
</tr>
<tr>
<td>rt</td>
<td>100000</td>
</tr>
<tr>
<td>imm</td>
<td>000000</td>
</tr>
</tbody>
</table>

Note: the differing order of registers in the 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 (addr)
- Used for jump instructions (j)

#### J-Type

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

#### R-Type

- **op**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **rd**: 5 bits
- **shamt**: 5 bits
- **funct**: 6 bits

#### I-Type

- **op**: 6 bits
- **rs**: 5 bits
- **rt**: 5 bits
- **imm**: 16 bits

#### J-Type

- **op**: 6 bits
- **addr**: 26 bits

### The Power of the Stored Program

- 32-bit instructions and data stored in memory
- Sequence of instructions: only difference between two applications (for example, a text editor and a video game)
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- The processor hardware executes the program:
  - *fetches* (reads) the instructions from memory in sequence
  - performs the specified operation
- The program counter (PC) keeps track of the current instruction
- In MIPS, programs typically start at memory address 0x00400000

### The Stored Program

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $t2, 32($0)</code></td>
<td>0x8C0A0020</td>
</tr>
<tr>
<td><code>add $s0, $s1, $s2</code></td>
<td>0x02328020</td>
</tr>
<tr>
<td><code>addi $t0, $s3, -12</code></td>
<td>0x2268FF4</td>
</tr>
<tr>
<td><code>sub $t0, $t3, $t5</code></td>
<td>0x016D4022</td>
</tr>
</tbody>
</table>

### Interpreting Machine Language Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is

### Field Values

- `0x2F34022`
**Programming**

- High-level languages:
  - e.g., C, Java, Python
  - Written at more abstract level
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - array accesses
  - procedure calls
- Other useful instructions:
  - Arithmetic/logical instructions
  - Branching

---

**Ada Lovelace, 1815 - 1852**

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage’s Analytical Engine
- She was the only legitimate child 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:
      \[ \text{0xF234012F AND 0x000000FF = 0x0000002F} \]
  - **or**: useful for combining bit fields
    - Combine 0xF2340000 with 0x000012BC:
      \[ \text{0xF2340000 OR 0x000012BC = 0xF23412BC} \]
  - **nor**: useful for inverting bits:
    - A NOR \( S_0 \) = NOT \( A \)
- **andi, ori, xori**
  - 16-bit immediate is zero-extended (not sign-extended)
  - nori not needed

---

**Logical Instruction Examples**

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

### Source Values

<table>
<thead>
<tr>
<th>$s1</th>
<th>$s2</th>
<th>$s3</th>
<th>$s4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 0000 0000 0000 0000 1111 1111</td>
<td>0000 0000 0000 0000 1111 1010 0011 0100</td>
<td>0000 0000 0000 0000 0000 0011 0100</td>
<td>0000 0000 0000 0000 1111 1010 1111 1111</td>
</tr>
</tbody>
</table>

### Assembly Code

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

### Result

- zero-extended
- `ori $s3, $s1, 0xFA34`
- `xori $s4, $s1, 0xFA34`

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

### Variable shift instructions:

- **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

Generating Constants

- 16-bit constants using `addi`:
  - **High-level code**
    ```c
    int a = 0x4f3c;
    addi $s0, $0, 0x4f3c
    ```
  - **MIPS assembly code**
    ```asm
    # $s0 = a
    addi $s0, $0, 0x4f3c
    ```

- 32-bit constants using load upper immediate (`lui`) and `ori`:
  - `(lui loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.)`

  - **High-level code**
    ```c
    int a = 0xFEDC8765;
    ```
  - **MIPS assembly code**
    ```asm
    lui $s0, 0xFEDC
    ori $s0, $s0, 0x8765
    ```
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, 32-bit remainder
  - div $s0$, $s1$
  - Quotient in lo
  - Remainder in hi
- Moves from lo/hi special registers
  - mflo $s2$
  - mfhi $s3$

Branching

- Allows a program to execute instructions out of sequence.
- Types of branches:
  - Conditional branches
    - branch if equal (beq)
    - branch if not equal (bne)
  - Unconditional branches
    - jump (j)
    - jump register (jr)
    - jump and link (jal)

Review: The Stored Program

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t2, 32(s0)</td>
<td>0880A000D</td>
</tr>
<tr>
<td>add $s0, $s1, $s2</td>
<td>000238024</td>
</tr>
<tr>
<td>addi $t0, $s3, -12</td>
<td>0002268FF4</td>
</tr>
<tr>
<td>sub $t0, $t3, $t5</td>
<td>000016D4022</td>
</tr>
</tbody>
</table>

# MIPS assembly

```
addi $s0, $0, 4  # $s0 = 0 + 4 = 4
addi $s1, $0, 1  # $s1 = 0 + 1 = 1
sl $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:  # label

```
add $s1, $s1, $s0  # $s1 = 4 + 4 = 8
```

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).
The Branch Not Taken (bne)

```mips
# 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 / Jumping (j)

```mips
# 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
```

Unconditional Branching (jr)

```mips
# MIPS assembly
0x00002000 addi $s0, $0, 0x2010
0x00002004 jr $s0
0x00002008 addi $s1, $0, 1
0x0000200c sra $s1, $s1, 2
0x00002010 lw $s3, 44($s1)

Note: jr is an R-type instruction.
```

High-Level Code Constructs

- if statements
- if/else statements
- while loops
- for loops
If Statement

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (i == j)</td>
<td># $s0 = f, $s1 = g, $s2 = h</td>
</tr>
<tr>
<td></td>
<td># $s3 = i, $s4 = j</td>
</tr>
<tr>
<td>f = g + h;</td>
<td></td>
</tr>
<tr>
<td>f = f - i;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1:</td>
</tr>
</tbody>
</table>

Notice that the assembly tests for the opposite case \((i \neq j)\) than the test in the high-level code \((i == j)\).

If / Else Statement

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (i == j)</td>
<td># $s0 = f, $s1 = g, $s2 = h</td>
</tr>
<tr>
<td></td>
<td># $s3 = i, $s4 = j</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>f = f - i;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1:</td>
</tr>
<tr>
<td></td>
<td>done:</td>
</tr>
</tbody>
</table>

While Loops

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>// determines the power</td>
<td># $s0 = pow, $s1 = x</td>
</tr>
<tr>
<td>// of x such that (2^x = 128)</td>
<td></td>
</tr>
<tr>
<td>int pow = 1;</td>
<td></td>
</tr>
<tr>
<td>int x = 0;</td>
<td></td>
</tr>
<tr>
<td>while (pow != 128) {</td>
<td></td>
</tr>
<tr>
<td>pow = pow * 2;</td>
<td></td>
</tr>
<tr>
<td>x = x + 1;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>done:</td>
</tr>
</tbody>
</table>

Notice that the assembly tests for the opposite case \((pow == 128)\) than the test in the high-level code \((pow != 128)\).

For Loops

The general form of a for loop is:

for (initialization; condition; loop operation)  
loop body

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

High-level code

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

MIPS assembly code

```mips
addi $s1, $0, 0
addi $s0, $0, 0
addi $t0, $0, 10
for:   beq  $s0, $t0, done
    add  $s1, $s1, $s0
    addi $s0, $s0, 1
    sll  $s0, $s0, 1
    j    loop
done:
```

Notice that the assembly tests for the opposite case (i == 128) than the test in the high-level code (i != 10).

Less Than Comparisons

High-level code

```c
int sum = 0;
for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

MIPS assembly code

```mips
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:

```$t1 = 1 if i < 101.

Arrays

- Useful for accessing large amounts of similar data
- Array element: accessed by index
- Array size: number of elements in the array

Arrays

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

```
0x12348000  array[0]
0x12348004  array[1]
0x12348008  array[2]
0x1234800C  array[3]
0x12348010  array[4]
```
Arrays

// high-level code
int array[5];
array[0] = array[0] * 2;

# MIPS assembly code
# array base address = $s0

Arrays Using For Loops

// high-level 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

Arrays

// high-level code
int array[5];
array[0] = array[0] * 2;

# MIPS assembly code
# array base address = $s0

Arrays Using For Loops

// high-level 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
# initialization code
lui $s0, 0x1234        # put 0x1234 in upper half of $s0
ori $s0, $s0, 0x8000   # put 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

# loop 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?
beq $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
add $s1, $s1, 1        # i = i + 1
}
loop     # repeat
done:
ASCII Codes

- **American Standard Code for Information Interchange**
  - assigns each text character a unique byte value
- For example, S = 0x53, a = 0x61, A = 0x41
- Lower-case and upper-case letters differ by 0x20 (32).

**Procedure Calls**

**Definitions**
- **Caller:** calling procedure (in this case, `main`)
- **Callee:** called procedure (in this case, `sum`)

**High-level code**
```c
void main()
{
    int y;
    y = sum(42, 7);
    ...
}
```

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

**Procedure calling conventions:**
- **Caller:**
  - passes arguments to callee.
  - jumps to the callee
- **Callee:**
  - performs the procedure
  - returns the result to caller
  - returns to the point of call
  - must not overwrite registers or memory needed by the caller

**MIPS conventions:**
- Call procedure: `jump and link (jal)`
  - Return from procedure: `jump register (jr)`
- Argument values: $s0 = s3$
- Return value: $v0$
Procedure Calls

**High-level 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.

---

Input Arguments and Return Values

**MIPS conventions:**
- Argument values: $a0 - $a3
- Return value: $v0

---

Procedure Calls

**High-level 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**
```assembly
0x00400200 main: jal simple
0x00400204 add $s0, $s1, $s2
... 0x00400210 simple: jr $ra
```

*jal*: jumps to *simple* and saves PC+4 in the return address register ($ra).

In this case, $ra = 0x00400204 after jal executes.

*jr $ra*: jumps to address in $ra, in this case 0x00400204.

---

Input Arguments and Return Values
Input Arguments and Return Values

MIPS assembly code

```assembly
# $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 diffoofts     # call procedure
add $s0, $v0, $0  # y = returned value...
```

```assembly
# $s0 = result
diffoofts:
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
```

- `diffoofts` overwrote 3 registers: $t0, $t1, and $s0
- `diffoofts` can use the `stack` to temporarily store registers

The Stack

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

- 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>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFFC</td>
<td>12345678</td>
<td>7FFFFFFC</td>
<td>12345678</td>
</tr>
<tr>
<td>7FFFFFFF8</td>
<td></td>
<td>7FFFFFFF8</td>
<td>AABBCDDE</td>
</tr>
<tr>
<td>7FFFFFFF4</td>
<td></td>
<td>7FFFFFFF4</td>
<td>11223344</td>
</tr>
<tr>
<td>7FFFFFFF0</td>
<td></td>
<td>7FFFFFFF0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How Procedures use the Stack

- Called procedures must have no other unintended side effects.
- But `diffosums` overwrites 3 registers: $t0, t1, s0

```mips
# $s0 = result
diffosums:
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
```

Storing Register Values on the Stack

```mips
# $s0 = result
diffosums:
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 `diffosums` 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>
</tbody>
</table>

Registers

Preserved

<table>
<thead>
<tr>
<th>Called-Saved</th>
<th>Nonpreserved</th>
<th>Caller-Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0 - $s7</td>
<td>$t0 - $t9</td>
<td>$ra</td>
</tr>
<tr>
<td>$ra</td>
<td>$a0 - $a3</td>
<td>$sp</td>
</tr>
<tr>
<td>$sp</td>
<td>$v0 - $v1</td>
<td>stack above $sp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stack below $sp</td>
</tr>
</tbody>
</table>

Copyright © 2007 Elsevier
Multiple Procedure Calls

proc1:
    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

Storing Saved Registers on the Stack

# $s0 = result
diffosums:
    addi $sp, $sp, -4  # make space on stack to
    sw $s0, 0($sp)    # store $s0 on stack
    # no need to save $t0 or $t1
    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 $s0, 0($sp)    # restore $s0 from stack
    addi $sp, $sp, 4  # deallocate stack space
    jr $ra            # return to caller

Recursive Procedure Call

High-level code

int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n-1));
}

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 add $t0, $0, 2
0xA0 slt $t0, $a0, $t0  # a <= 1 ?
0xA4 beq $t0, $0, else # no: go to else
0xA8 add $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
0xCB mul $v0, $a0, $v0 # n * factorial(n-1)
0xCC jr $ra # return
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>FC</td>
<td>$sp</td>
<td>FC</td>
<td>$sp</td>
<td>$sp</td>
<td>$s0 = 6</td>
</tr>
<tr>
<td>F8</td>
<td>$a0 (lo)</td>
<td>F8</td>
<td>$a0 (lo)</td>
<td>$sp</td>
<td>$s0 + 3</td>
</tr>
<tr>
<td>F4</td>
<td>$a</td>
<td>F4</td>
<td>$a</td>
<td>$sp</td>
<td>$s0 + 3 x 2</td>
</tr>
<tr>
<td>F0</td>
<td>$a0 (hi)</td>
<td>F0</td>
<td>$a0 (hi)</td>
<td>$sp</td>
<td>$s0 + 2</td>
</tr>
<tr>
<td>EC</td>
<td>$a (lo4)</td>
<td>EC</td>
<td>$a (lo4)</td>
<td>$sp</td>
<td>$s0 + 2 x 1</td>
</tr>
<tr>
<td>EB</td>
<td>$a0 (hi)</td>
<td>EB</td>
<td>$a0 (hi)</td>
<td>$sp</td>
<td>$s0 + 1</td>
</tr>
<tr>
<td>E4</td>
<td>$a (lo4)</td>
<td>E4</td>
<td>$a (lo4)</td>
<td>$sp</td>
<td>$s0 + 1 x 1</td>
</tr>
<tr>
<td>E0</td>
<td>$a</td>
<td>E0</td>
<td>$a</td>
<td>$sp</td>
<td>$a0</td>
</tr>
<tr>
<td>DC</td>
<td>DC</td>
<td>DC</td>
<td>DC</td>
<td>$a0 (0x3)</td>
<td>$ra (0xBC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$a0 (0x2)</td>
<td>$ra (0xBC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$a0 (0x1)</td>
<td>$ra (0xBC)</td>
</tr>
</tbody>
</table>

Copyright © 2007 Elsevier

Procedure Call Summary

- **Caller**
  - Put arguments in $a0-$a3
  - Save any registers that are needed ($ra, maybe $t0-$t9)
  - `jal callee`
  - Restore registers
  - Look for result in $v0
- **Callee**
  - Save registers that might be disturbed ($s0-$s7)
  - Perform procedure
  - Put result in $v0
  - Restore registers
  - `jr $ra`

Addressing Modes

How do we address the operands?

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

Register Only Addressing

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

Immediate Addressing

- 16-bit immediate used as an operand
  - Example: `andi $t4, $t5, -73`
  - Example: `ori $t3, $t7, 0xFF`
### Addressing Modes

#### Base Addressing
- Address of operand is:
  base address + sign-extended immediate
- Example: `lw $s4, 72($0)`
  - Address = $0 + 72
- Example: `sw $t2, -25($t1)`
  - Address = $t1 - 25

#### PC-Relative Addressing
- Example: `beq $t0, $0, else` `addi $v0, $0, 1` `addi $sp, $sp, i` `jr $ra` `addi $a0, $a0, -1` `jal factorial`

#### Pseudo-direct Addressing
- Example: `jal sum`
- `sum: add $v0, $a0, $a1`
Grace Hopper, 1906 - 1992

- Graduated from Yale University with PhD 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 needs to be 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

The MIPS Memory Map

Example Program: C Code

```c
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);
}
```

Copyright © 2007 Elsevier
Example Program: Assembly Code

```assembly
int f, g, y; // global .data
f: 
g: 
y: 
int main(void) .text
{
    main:
        addi $sp, $sp, -4  # stack frame
        sw $ra, 0($sp)    # store $ra
        f = 2;
        addi $a0, $0, 2   # $a0 = 2
        sw $a0, f         # $a0 = f
        g = 3;
        addi $a1, $0, 3   # $a1 = 3
        sw $a1, g         # $a1 = g
        y = sum(f, g);   # call sum
        sw $v0, y         # y = sum()
    return y;   # return to OS
}
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>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>0x20 (12 bytes)</td>
<td>0x20 (12 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000500</td>
<td>add $sp, $sp, -4  # stack frame</td>
<td></td>
</tr>
<tr>
<td>0x00000504</td>
<td>sw $ra, 0($sp)    # store $ra</td>
<td></td>
</tr>
<tr>
<td>0x00000508</td>
<td>f = 2;</td>
<td></td>
</tr>
<tr>
<td>0x0000050C</td>
<td>addi $a0, $0, 2   # $a0 = 2</td>
<td></td>
</tr>
<tr>
<td>0x00000510</td>
<td>sw $a0, f         # $a0 = f</td>
<td></td>
</tr>
<tr>
<td>0x00000514</td>
<td>g = 3;</td>
<td></td>
</tr>
<tr>
<td>0x00000518</td>
<td>addi $a1, $0, 3   # $a1 = 3</td>
<td></td>
</tr>
<tr>
<td>0x0000051C</td>
<td>sw $a1, g         # $a1 = g</td>
<td></td>
</tr>
<tr>
<td>0x00000520</td>
<td>y = sum(f, g);   # call sum</td>
<td></td>
</tr>
<tr>
<td>0x00000524</td>
<td>sw $v0, y         # y = sum()</td>
<td></td>
</tr>
<tr>
<td>0x00000528</td>
<td>return y;   # return to OS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data segment</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000000</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>0x10000004</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>0x10000008</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

Example Program: In Memory

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFC</td>
<td>Stack</td>
</tr>
<tr>
<td>0x23BDFFFC</td>
<td>Heap</td>
</tr>
<tr>
<td>0x10000000</td>
<td>Stack</td>
</tr>
<tr>
<td>0x10008000</td>
<td>Heap</td>
</tr>
<tr>
<td>0x00400000</td>
<td>PC = 0x00400000</td>
</tr>
</tbody>
</table>

Copyright © 2007 Elsevier
### Odds and Ends

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

### Pseudoinstruction Examples

<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>mul $s0, $s1, $s2</td>
<td>mult $s1, $s2</td>
</tr>
<tr>
<td></td>
<td>mflo $s0</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 procedure call to the 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 the exception handler at instruction address 0x80000180
  - Returns to program

### Exception Registers

- Not part of the register file.
  - Cause
    - Records the cause of the exception
  - EPC (Exception PC)
    - Records the PC where the exception occurred
- EPC and Cause: part of Coprocessor 0
- Move from Coprocessor 0
  - mfc0 $t0, EPC
  - Moves the contents of EPC into $t0
### 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>

### Exceptions

- Processor saves cause and exception PC in `Cause` and `EPC`
- Processor jumps to exception handler (0x80000180)
- Exception handler:
  - Saves registers on stack
  - Reads the `Cause` register
    ```
    mfc0 $t0, Cause
    ```
  - Handles the exception
  - Restores registers
  - Returns to program
    ```
    mfc0 $k0, EPC
    jr $k0
    ```

### Signed and Unsigned Instructions

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

### Addition and 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
Multiplication and Division

- **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:** addu, addiu, subu
  - 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.
  - So, 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 - $fa1</td>
<td>12, 14</td>
<td>procedure arguments</td>
</tr>
<tr>
<td>$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

- Opcode = 17 (0100012)
- Single-precision:
  - cop = 16 (0100002)
  - add.s, sub.s, div.s, neg.s, abs.s, etc.
- Double-precision:
  - cop = 17 (0100012)
  - add.d, sub.d, div.d, neg.d, abs.d, etc.
- 3 register operands:
  - fs, ft: source operands
  - fd: destination operands

<table>
<thead>
<tr>
<th>F-Type</th>
<th>op</th>
<th>cop</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>functl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>

Floating-Point Branches

- Set/clear condition flag: fpcond
  - Equality: c.eq.s, c.eq.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
  - lwcl: lwcl $ft1, 42($si)
  - swcl: swcl $fs2, 17($sp)

Looking Ahead

- Microarchitecture – building MIPS processor in hardware!
  - Bring colored pencils