EN164: Design of Computing Systems
Topic 03: Instruction Set Architecture Design

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2012
ISA is the HW/SW interface

ISA choice determines:

• program size (& memory size)
• complexity of hardware (CPI and f)
• execution time for different applications and domains
• power consumption
• die area (cost)
Stored program concept (von Neumann model)

- Instructions represented in binary numbers, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

The BIG Picture

- Processor
  - Memory
    - Accounting program (machine code)
    - Editor program (machine code)
    - C compiler (machine code)
    - Payroll data
    - Book text
    - Source code in C for editor program
Steps in execution of a program

- Fetch instruction @ PC
- Decode instruction
- Fetch Operands
- Execute instruction
- Store result
- Update PC

- What is instruction format / size?
- How is it decoded?
- Where are the operands located? What are their sizes?
- What are supported operations?
- How to determine the successor instruction?
Example of an instruction

<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>

add $t0, $s1, $s2

assembly language

special | $s1 | $s2 | $t0 | 0 | add |

| 0 | 17 | 18 | 8 | 0 | 32 |

| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

00000010001100100100000000100000₂ = 02324020₁₆

machine language
ISA design choices

• Number, size (fixed/variable) and format of instructions
• Operations supported (arithmetic, logical, string, floating point, jump, etc)
• Operands supported (bytes, words, signed, unsigned, floating, etc)
• Operand storage (accumulator, stack, registers, memory)
• Addressing modes
# Typical operations

## Data Movement
- Load (from memory)
- Store (to memory)
- Memory-to-memory move
- Register-to-register move
- Input (from I/O device)
- Output (to I/O device)
- Push, pop (to/from stack)

## Arithmetic
- Integer (binary + decimal) or FP
- Add, Subtract, Multiply, Divide

## Shift
- Shift left/right, rotate left/right

## Logical
- Not, and, or, set, clear

## Control (Jump/Branch)
- Unconditional, conditional

## Subroutine Linkage
- Call, return

## Interrupt
- Trap, return

## Synchronization
- Test & set (atomic r-m-w)
- Search, translate

[slide from M. Martin]
Classification of ISAs

These ISAs give different characteristics in terms of size of programs, number of instructions and CPI.

[Figure from D. Brooks -- Harvard]
Examples of ISA

Instruction sequence for $C = A + B$ for the four ISAs

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1, A</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C, R3</td>
</tr>
</tbody>
</table>

Some architectures (e.g. x86) support hybrid ISAs for different classes of instructions and/or for backward compatibility.
What makes a good ISA?

- Efficiency of hardware implementation
- Convenience of programming / compiling
- Matches target applications (or generality)
- Compatibility and portability

Four design principles for ISA

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

ISA design is an art!
Popular ISAs

• x86 from Intel (laptops, servers)
• ARM (mobile devices)
• MIPS (embedded devices)
• Power and PowerPC from IBM (servers, old Macs)
• and many others still spoken and dead ISAs
MIPS architecture: Example of Reduced Instruction Set Computer (RISC) architecture

• Used as the example throughout the course
• Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
• Large share of embedded core market
  – Applications in consumer electronics, network/storage equipment, cameras, printers, …
• Typical of many modern ISAs
• 32 bit and 64 bit available
• Will implement and boot a subset MIPS processor in lab
Arithmetic operations

**High-level code**

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

**MIPS assembly code**

```plaintext
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
sub $s0, $s1, $s2
```

- **add**: mnemonic indicates what operation to perform
- **$s1, $s1**: source operands on which the operation is performed
- **$s0**: destination operand to which the result is written

- **Design Principle 1**: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
Register operands

- Memory is slow.
- Most architectures have a small set of (fast) registers
- MIPS has a 32 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
  - Written with a $ before their name

- Assembler names
  - $0 always hold value 0
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations
# MIPS registers

<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>
Format of R-type instructions

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>

- **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
### R-Type examples

#### Assembly Code

<table>
<thead>
<tr>
<th>Code</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0, $s1, $s2</td>
<td>0, 17, 18, 16, 0, 32</td>
</tr>
<tr>
<td>sub $t0, $t3, $t5</td>
<td>0, 11, 13, 8, 0, 34</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>10001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>000000</td>
<td>01011</td>
<td>01101</td>
<td>01000</td>
<td>00000</td>
<td>100010</td>
</tr>
</tbody>
</table>

#### Note

The order of registers in the assembly code:

```
add rd, rs, rt
```
Memory operands

- Too much data to fit in only 32 registers
- Memory is large, but 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

- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
Word-addressable memory

- Each 32-bit data word has a unique address

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<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>
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* ($0$) 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>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000003</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>0000000002</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>0000000001</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>
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

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

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<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 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

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Data</th>
<th>Data</th>
<th>Data</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000C</td>
<td>4 0</td>
<td>F 3</td>
<td>0 7</td>
<td>8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>00000008</td>
<td>0 1</td>
<td>E E</td>
<td>2 8</td>
<td>4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>00000004</td>
<td>F 2</td>
<td>F 1</td>
<td>A C</td>
<td>0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000000</td>
<td>A B</td>
<td>C D</td>
<td>E F</td>
<td>7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>

width = 4 bytes
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 \( 2 \times 4 = 8 \)
  - the address of memory word 10 is \( 10 \times 4 = 40 \) (0x28)

- Load a word of data at memory address 4 into $s3.
- $s3$ holds the value 0xF2F1AC07 after the instruction completes.
- MIPS is byte-addressed, not word-addressed.

**MIPS assembly code**

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

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000C</td>
<td>4 0 F 3 0 7 8 8</td>
</tr>
<tr>
<td>00000008</td>
<td>0 1 E E 2 8 4 2</td>
</tr>
<tr>
<td>00000004</td>
<td>F 2 F 1 A C 0 7</td>
</tr>
<tr>
<td>00000000</td>
<td>A B C D E F 7 8</td>
</tr>
</tbody>
</table>

width = 4 bytes
Writing byte-addressed 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>Word</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>

width = 4 bytes
Big-Endian and little-Endian

- 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>Byte Address</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

MSB   LSB

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

MSB   LSB
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 in action

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).
Constant/Immediate operands

- 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?

### High-level code

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

### MIPS assembly code

```plaintext
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```
I-Type format instructions

- **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

<table>
<thead>
<tr>
<th>I-Type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
### I-Type examples

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

#### Machine Code

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10001</td>
<td>10000</td>
<td>0000 0000 0000 0101</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>

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

- addi rt, rs, imm
- lw  rt, imm(rs)
- sw  rt, imm(rs)
Logical operations

Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

Useful for extracting and inserting groups of bits in a word
Shift operations

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - **sll** by $i$ bits multiplies by $2^i$
- **Shift right logical**
  - Shift right and fill with 0 bits
  - **srl** by $i$ bits divides by $2^i$ (unsigned only)
AND operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0
and $t0$, $t1$, $t2$

```
$t2$  0000 0000 0000 0000 0000 1101 1100 0000
$t1$  0000 0000 0000 0000 0011 1100 0000 0000
$t0$  0000 0000 0000 0000 0000 1100 0000 0000
```
OR operations

- Useful to include bits in a word
- Set some bits to 1, leave others unchanged

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

<table>
<thead>
<tr>
<th>$t2$</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1$</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0$</td>
<td>0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
NOT operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - \( a \text{ NOR } b = \text{ NOT}(a \text{ OR } b) \)

\[
\text{nor } \$t0, \$t1, \$zero
\]

| $t1$ | 0000 0000 0000 0000 0011 1100 0000 0000 |
| $t0$ | 1111 1111 1111 1111 1100 0011 1111 1111 |

Register 0: always read as zero
Conditional operators

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- \texttt{beq rs, rt, L1}
  - if (rs == rt) branch to instruction labeled L1;
- \texttt{bne rs, rt, L1}
  - if (rs != rt) branch to instruction labeled L1;
- \texttt{j L1}
  - unconditional jump to instruction labeled L1
Compiling if statement

- **C code:**
  
  ```
  if (i==j) f = g+h;
  else f = g-h;
  ```

  - f, g, ... in $s0, $s1, ...

- **Compiled MIPS code:**
  
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```
Compiling loop statements

- **C code**:

  ```c
  while (save[i] == k) i += 1;
  ```

  - `i` in `$s3`, `k` in `$s5`, address of `save` in `$s6`

- **Compiled MIPS code**:

  ```mips
  Loop:    sll  $t1, $s3, 2
           add  $t1, $t1, $s6
           lw   $t0, 0($t1)
           bne  $t0, $s5, Exit
           addi $s3, $s3, 1
           j    Loop
  Exit:    ...
  ```
More conditional operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq, bne`
  - `slt $t0, $s1, $s2`  # if ($s1 < $s2)
  - `bne $t0, $zero, L`  # branch to L
Why not `blt`, `bge`, etc?

Hardware for `<`, `≥`, … slower than `=`, ≠

- Combining with branch involves more work per instruction, requiring a slower clock
- All instructions penalized!

`beq` and `bne` are the common case

This is a good design compromise
### Summary of 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>
Procedure calls

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: $a0 - $a3
• Return value: $v0 -$v1
Procedure calls

High-level code
```c
int main() {
    simple();
    a = b + c;
}
```

```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.**

**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.
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
}
```
Input arguments and return values

MIPS assembly code

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 procedure
   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 conventions:

- Argument values: $a0 - $a3
- Return value: $v0
Potential problems in procedural calling

MIPS assembly code

```assembly
# $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
```

- `diffosums` overwrote 3 registers: `$t0`, `$t1`, and `$s0`
- `diffosums` 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 by growing down (from higher to lower memory addresses) when more space is needed
- *Contracts*: uses less memory when the space is no longer needed
- 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>7FFFFFFFC</td>
<td>12345678</td>
</tr>
<tr>
<td>7FFFFFF8</td>
<td></td>
<td>7FFFFFF8</td>
<td>AABBCCDD</td>
</tr>
<tr>
<td>7FFFFFF4</td>
<td></td>
<td>7FFFFFF4</td>
<td>11223344</td>
</tr>
<tr>
<td>7FFFFFF0</td>
<td></td>
<td>7FFFFFF0</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

47
How procedures use the stack

- Called procedures must have no other 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
```
Storing register values on the stack

# $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

Address | Data     | Address | Data     | Address | Data
---------|----------|---------|----------|---------|----------
FC       | ?        | FC      | ?        | FC      | ?
F8       |          | F8      | $s0      | F8      |          
F4       |          | F4      | $t0      | F4      |          
F0       |          | F0      | $t1      | F0      |          
:         |          | :       |          | :       |          |
## Protocol for preserving 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>stack above $sp</td>
<td>stack below $sp</td>
</tr>
</tbody>
</table>
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
diffsofsums:

    addi $sp, $sp, -4  # make space on stack to store one register
    sw  $s0, 0($sp)    # save $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 calls

**High-level code**

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

MIPS assembly code

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

<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></td>
<td>FC</td>
<td>$sp</td>
<td>FC</td>
<td>$sp</td>
</tr>
<tr>
<td>F8</td>
<td>$a0 (0x3)</td>
<td>F8</td>
<td>$sp</td>
<td>F8</td>
<td>$sp</td>
</tr>
<tr>
<td>F4</td>
<td>$ra</td>
<td>F4</td>
<td>$sp</td>
<td>F4</td>
<td>$sp</td>
</tr>
<tr>
<td>F0</td>
<td>$a0 (0x2)</td>
<td>F0</td>
<td>$sp</td>
<td>F0</td>
<td>$sp</td>
</tr>
<tr>
<td>EC</td>
<td>$ra (0xBC)</td>
<td>EC</td>
<td>$sp</td>
<td>EC</td>
<td>$sp</td>
</tr>
<tr>
<td>E8</td>
<td>$a0 (0x1)</td>
<td>E8</td>
<td>$sp</td>
<td>E8</td>
<td>$sp</td>
</tr>
<tr>
<td>E4</td>
<td>$ra (0xBC)</td>
<td>E4</td>
<td>$sp</td>
<td>E4</td>
<td>$sp</td>
</tr>
<tr>
<td>E0</td>
<td></td>
<td>E0</td>
<td></td>
<td>E0</td>
<td></td>
</tr>
<tr>
<td>DC</td>
<td></td>
<td>DC</td>
<td></td>
<td>DC</td>
<td></td>
</tr>
<tr>
<td>$a0 = 1</td>
<td>$v0 = 1 \times 1</td>
<td>$a0 = 2</td>
<td>$v0 = 2 \times 1</td>
<td>$a0 = 3</td>
<td>$v0 = 3 \times 2</td>
</tr>
<tr>
<td>$a0 = 2</td>
<td>$v0 = 2 \times 1</td>
<td>$a0 = 3</td>
<td>$v0 = 3 \times 2</td>
<td>$a0 = 1</td>
<td>$v0 = 1 \times 1</td>
</tr>
</tbody>
</table>

$sp$ points to the stack pointer.
Procedural 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
x86 ISA: Example of Complex Instruction Set Computer (CISC) architecture

- Backward compatibility $\Rightarrow$ instruction set doesn’t change
- But they do accumulate more instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td></td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

- Simple instructions dominate instruction frequency
x86 registers

General-purpose Registers

- EAX
  - AX
  - AH
  - AL
- EBX
  - BX
  - BH
  - BL
- ECX
  - CX
  - CH
  - CL
- EDX
  - DX
  - DH
  - DL
- ESI
-EDI
- ESP (stack pointer)
- EBP (base pointer)
x86 assembly language examples

# move the 4 bytes in memory at the address contained in EBX into EAX
mov eax, [ebx]

# move the 4 bytes of data at address ESI+4*EBX into EDX
mov edx, [esi+4*ebx]

# eax = eax + ebx
add eax, ebx

# add ebx to memory content at address memory_location
add memory_location, ebx

# add 10 the data at address EAX
add [eax], 10

# push content of eax into top of stack
push eax

# compare two registers, register - memory or memory memory
cmp eax, ebx

cmp [eax], [ebx]

# conditional jumps
jl, jeq, jg, jge, jle, jne