Instruction Set Architecture (ISA)

- The “deal” between hardware and software
- Languages get compiled down to the ISA
- Hardware must implement the ISA
- An “interface”; can be extended, but nothing can be removed!
- Code compatibility; backward compatibility

Examples of ISAs
- x86, AMD64, PA-RISC, ARM, ..., MIPS!
Levels of Program Code

High-level language
• Level of abstraction closer to problem domain
• Provides for productivity and portability

Assembly language
• Textual representation of instructions

Hardware representation
• Binary digits (bits)
• Encoded instructions and data
Translation and Startup

Many compilers produce object modules directly

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory
Instruction Set

Different computers often have different instruction sets
  – But with many aspects in common

Early computers had very simple instruction sets
  – Simplified implementation

The old RISC vs. CISC debate
The MIPS Instruction Set

Used as the example throughout the book

Stanford MIPS commercialized by MIPS Technologies ([www.mips.com](http://www.mips.com))

Large share of embedded core market

- Applications in consumer electronics, network/storage equipment, cameras, printers, ...

Typical of many modern ISAs

- See MIPS Reference Data tear-out card, and Appendicies B and E
Arithmetic Operations

Add and subtract, three operands
• Two sources and one destination
  add a, b, c  # a gets b + c
All arithmetic operations have this form

*Design Principle 1: Simplicity favors regularity*
• Regularity makes implementation simpler
• Simplicity enables higher performance at lower cost
Arithmetic Example

C code:

\[ f = (g + h) - (i + j); \]

MIPS:

```
add $t0, $t4, $t5   # temp t0 = g + h
add $t1, $t6, $t7   # temp t1 = i + j
sub $t2, $t0, $t1   # f = t0 - t1
```
Registers vs. Memory

Registers are faster to access than memory

Operating on data in memory requires loads and stores
Register Operands

Arithmetic instructions use register operands

MIPS has a $32 \times 32$-bit register file
- Use for frequently accessed data
- Numbered 0 to 31
- 32-bit data called a “word”

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

Design Principle 2: Smaller is faster
- c.f. main memory: millions of locations
Memory Operands

Main memory used for composite data
• Arrays, structures, dynamic data

To apply arithmetic operations
• Load values from memory into registers
• Store result from register to memory

Memory is byte addressed
• Each address identifies an 8-bit byte

Words are aligned in memory
• Address must be a multiple of 4

MIPS is Big Endian
• Most-significant byte at lowest address of a word
• Little Endian: least-significant byte at lowest address
Memory Operand Example 1

C code:

\[ g = h + A[8]; \]
\[ g \text{ in } \$s1, \ h \text{ in } \$s2, \ \text{base address of } A \text{ in } \$s3 \]

Compiled MIPS code:

- Index 8 requires offset of 32
- 4 bytes per word

\[
lw \ \$t0, 32(\$s3) \quad \# \text{ load word}
\]
\[
add \ \$s1, \ \$s2, \ \$t0
\]
Memory Operand Example 2

C code:


• h in $s2, base address of A in $s3

Compiled MIPS code:

• Index 8 requires offset of 32

lw $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw $t0, 48($s3)    # store word
Registers vs. Memory

Registers are faster to access than memory

Operating on data in memory requires loads and stores

- More instructions to be executed

Compiler must use registers for variables as much as possible

- Only spill to memory for less frequently used variables
- Register optimization is important!
Immediate Operands

Constant data specified in an instruction
   addi $s3, $s3, 4

No subtract immediate instruction
   • Just use a negative constant
     addi $s2, $s1, -1

Design Principle 3: Make the common case fast
   • Small constants are common
   • Immediate operand avoids a load instruction
The Constant Zero

MIPS register 0 ($zero) is the constant 0
• Cannot be overwritten

Useful for common operations
• E.g., move between registers
  add $t2, $s1, $zero
Unsigned Binary Integers

- **Given an n-bit number**

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- **Range:** 0 to \( +2^n - 1 \)

- **Example**
  
  \[
  \begin{align*}
  0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2 &= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 \\
  &= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}
  \end{align*}
  \]

- **Using 32 bits**
  
  - 0 to \(+4,294,967,295\)
2s-Complement Signed Integers

- Given an n-bit number

\[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: \(-2^{n-1}\) to \(+2^{n-1} - 1\)

- Example
  - 1111 1111 1111 1111 1111 1111 1111 1100_2
  - \(-1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\)
  - \(-2,147,483,648 + 2,147,483,644 = -4_{10}\)

- Using 32 bits
  - \(-2,147,483,648\) to \(+2,147,483,647\)
2s-Complement Signed Integers

Bit 31 is sign bit
• 1 for negative numbers
• 0 for non-negative numbers
\[-(-2^{n-1})\text{ can’t be represented}\]

Non-negative numbers have the same unsigned and 2s-complement representation

Some specific numbers
• 0: \(0000\ 0000\ \ldots\ \ 0000\)
• \(-1:\ \ 1111\ 1111\ \ldots\ \ 1111\)
• Most negative: \(1000\ 0000\ \ldots\ \ 0000\)
• Most positive: \(0111\ 1111\ \ldots\ \ 1111\)
Signed Negation

- **Complement and add 1**
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

\[
\overline{x + x} = \overline{1111...111_2} = -1
\]
\[
\overline{x + 1} = -x
\]

- **Example: negate +2**
  - $+2 = 0000 \ 0000 \ ... \ 0010_2$
  - $-2 = 1111 \ 1111 \ ... \ 1101_2 + 1$
    - $= 1111 \ 1111 \ ... \ 1110_2$
Sign Extension

Representing a number using more bits
• Preserve the numeric value

In MIPS instruction set
• addi: extend immediate value
• lb, lh: extend loaded byte/halfword
• beq, bne: extend the displacement

Replicate the sign bit to the left
• c.f. unsigned values: extend with 0s

Examples: 8-bit to 16-bit
• +2: 0000 0010 => 0000 0000 0000 0010
• –2: 1111 1110 => 1111 1111 1111 1110
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination

move $t0, $t1 → add $t0, $zero, $t1
blt $t0, $t1, L → slt $at, $t0, $t1
bne $at, $zero, L

- $at (register 1): assembler temporary
Stored Program Computers

The BIG Picture

Instructions represented in binary, 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
Levels of Program Code

High-level language
- Level of abstraction closer to problem domain
- Provides for productivity and portability

Assembly language
- Textual representation of instructions

Hardware representation
- Binary digits (bits)
- Encoded instructions and data
Representing Instructions

Instructions are encoded in binary
• Called machine code

MIPS instructions
• Encoded as 32-bit instruction words
• Small number of formats encoding operation code (opcode), register numbers, ...
• Regularity!

Register numbers
• $t0 – t7$ are reg’s 8 – 15
• $s0 – s7$ are reg’s 16 – 23
• $t8 – t9$ are reg’s 24 – 25
MIPS R-format Instructions

<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></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>

**Instruction fields**

- **op**: operation code (opcode)
- **rs**: first source register number
- **rt**: second source register number
- **rd**: destination register number
- **shamt**: shift amount (000000 for now)
- **funct**: function code (extends opcode)
From MIPS to Binary: R-format Example

<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

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

0000001000110010010000000001000000_2 = 02324020_{16}
Hexadecimal

- **Base 16**
  - Compact representation of bit strings
  - 4 bits per hex digit

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>e</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>d</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
<td>1010</td>
<td>c</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</td>
<td>1011</td>
<td>f</td>
<td>1111</td>
</tr>
</tbody>
</table>

- **Example: eca8 6420**
  - 1110 1100 1010 1000 0110 0100 0010 0000
MIPS I-format Instructions

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

Immediate arithmetic and load/store instructions

• rt: destination or source register number
• Constant: $-2^{15}$ to $+2^{15} - 1$
• Address: offset added to base address in rs

Design Principle 4: Good design demands good compromises

• Different formats complicate decoding, but allow 32-bit instructions uniformly
• Keep formats as similar as possible
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><code>&lt;&lt;</code></td>
<td><code>&lt;&lt;</code></td>
<td><code>sll</code></td>
</tr>
<tr>
<td>Shift right</td>
<td><code>&gt;&gt;</code></td>
<td><code>&gt;&gt;&gt;</code></td>
<td><code>srl</code></td>
</tr>
<tr>
<td>Bitwise AND</td>
<td><code>&amp;</code></td>
<td><code>&amp;</code></td>
<td><code>and, andi</code></td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>`</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td><code>~</code></td>
<td><code>~</code></td>
<td><code>nor</code></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
  - \texttt{sll} by \( i \) bits multiplies by \( 2^i \)
- **Shift right logical**
  - Shift right and fill with 0 bits
  - \texttt{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 $t0$, $t1$, $t2$
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></th>
<th>$t0$</th>
<th>$t1$</th>
<th>$t2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000 0000 0000 0011 1101 1100 0000</td>
<td>0000 0000 0000 0000 0011 1101 1100 0000</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
  - \( a \text{ NOR } b = \text{ NOT ( } a \text{ OR } b \text{ )} \)

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

$\text{Register 0: always read as zero}$

| $\$t0$ | 1111 1111 1111 1111 1100 0011 1111 1111 |
| $\$t1$ | 0000 0000 0000 0000 0011 1100 0000 0000 |
Conditional Operations

Branch to a labeled instruction if a condition is true (otherwise, continue sequentially)

\[\text{beq \ rs, \ rt, \ L1}\]
- if \(\text{rs} == \text{rt}\) branch to instruction labeled \(\text{L1}\)

\[\text{bne \ rs, \ rt, \ L1}\]
- if \(\text{rs} \neq \text{rt}\) branch to instruction labeled \(\text{L1}\)

\[\text{j \ L1}\]
- unconditional jump to instruction labeled \(\text{L1}\)
Compiling If Statements

- 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: ...

Assembler calculates addresses
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: ...
  ```
Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks
More Conditional Operations

- **Set result to 1 if a condition is true**
  - Otherwise, set to 0

\[
\text{slt } \text{rd, rs, rt} \\
\text{if (rs < rt) rd = 1; else rd = 0;}
\]

\[
\text{slti } \text{rt, rs, constant} \\
\text{if (rs < constant) rt = 1; else rt = 0;}
\]

- **Use in combination with beq, bne**
  
  \[
  \text{slt } \$t0, \$s1, \$s2 \ \# \ \text{if ($s1 < $s2)} \\
  \text{bne } \$t0, \$\text{zero, L} \ \# \ \text{branch to L}
  \]
Branch Instruction Design

- **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**
Signed vs. Unsigned

- **Signed comparison:** `slt`, `slti`
- **Unsigned comparison:** `sltu`, `sltui`
- **Example**
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt $t0, $s0, $s1  # signed`
    - `-1 < +1 ⇒ $t0 = 1`
  - `sltu $t0, $s0, $s1  # unsigned`
    - `+4,294,967,295 > +1 ⇒ $t0 = 0`
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  \[ \text{lui } rt, \text{ constant} \]
  - Copies 16-bit constant to left 16 bits of \( rt \)
  - Clears right 16 bits of \( rt \) to 0

\[
\begin{align*}
lui & \; $s0, 61 & \quad & \text{0000 0000 0111 1101 0000 0000 0000 0000} \\
\text{ori} & \; $s0, $s0,0x900 & \quad & \text{0000 0000 0111 1101 0000 1001 0000 0000}
\end{align*}
\]
Branch Addressing

- **Branch instructions specify**
  - Opcode, two registers, target address

- **Most branch targets are near branch**
  - Forward or backward

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

- **PC-relative addressing**
  - Target address = PC + offset × 4
  - PC has already updated (PC += 4)
Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = address × 4
  - Each instruction is 4 bytes long, so addresses are referenced in words, and therefore enable jumps up to 4x as far.
**Target Addressing Example**

- Loop code from earlier example
  - Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Offset</th>
<th>Loop Counter</th>
<th>Index Counter</th>
<th>Exit Counter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sll $t1, $s3, 2</code></td>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>9</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td><code>add $t1, $t1, $s6</code></td>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td><code>lw $t0, 0($t1)</code></td>
<td>80008</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><code>bne $t0, $s5, Exit</code></td>
<td>80012</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><code>addi $s3, $s3, 1</code></td>
<td>80016</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><code>j Loop</code></td>
<td>80020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Exit: ...</strong></td>
<td>80024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```assembly
beq $s0,$s1, L1
↓
bne $s0,$s1, L2
j L1
L2: ...
```
Addressing Mode Summary

1. Immediate addressing
   \[
   \begin{array}{c|c|c|c|c}
   \hline
   \text{op} & \text{rs} & \text{rt} & \text{Immediate} \\
   \hline
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{c|c|c|c|c|c|c}
   \hline
   \text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{funct} \\
   \hline
   \end{array}
   \]
   Registers
   \[
   \begin{array}{c|c|c|c}
   \hline
   \text{Register} \\
   \hline
   \end{array}
   \]
   Memory
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \hline
   \text{Byte} & \text{Halfword} & \text{Word} \\
   \hline
   \end{array}
   \]

3. Base addressing
   \[
   \begin{array}{c|c|c|c|c}
   \hline
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c|c|c}
   \hline
   \text{Register} \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \hline
   \text{Byte} & \text{Halfword} & \text{Word} \\
   \hline
   \end{array}
   \]

4. PC-relative addressing
   \[
   \begin{array}{c|c|c|c|c}
   \hline
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \hline
   \text{PC} \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \hline
   \text{Word} \\
   \hline
   \end{array}
   \]

5. Pseudodirect addressing
   \[
   \begin{array}{c|c|c|c|c}
   \hline
   \text{op} & \text{Address} \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \hline
   \text{PC} \\
   \hline
   \end{array}
   \]
   \[
   \begin{array}{c|c|c|c|c|c|c|c}
   \hline
   \text{Word} \\
   \hline
   \end{array}
   \]
Another Abstraction: Procedures

In C:

```c
void main()
{
    int a = foo(1);
    printf("%d\n", a);
}

int foo(int myarg)
{
    int x = myarg++;
    return x;
}
```

Steps required

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure’s operations
5. Place result in register for caller
6. Return to place of call
MIPS Calling Convention

$\text{a0-}\text{a3}: \text{four argument registers for passing parameters}
$\text{v0-}\text{v1}: \text{two return value registers}
$\text{ra}: \text{one return address register (for jumping back to caller)}
Procedure Call Instructions

- **Procedure call: jump and link**
  
  \[
  \text{jal \ ProcedureLabel}
  \]
  
  - Address of following instruction put in $ra
  - Jumps to target address

- **Procedure return: jump register**
  
  \[
  \text{jr \ $ra}
  \]
  
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements
Memory Layout

- **Text**: program code
- **Static data**: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp$ initialized to address allowing ±offsets into this segment
- **Dynamic data**: heap
  - E.g., malloc in C, new in Java
- **Stack**: automatic storage
Local Data on the Stack

- Local data allocated by callee
- Procedure frame (activation record)
How MIPS uses the stack

- At the beginning of a program, the stack pointer ($sp) is set to a high value (e.g. 0x7fffffff0c).
- If a function needs to place data on the stack, it “allocates” space (a “push”) by simply moving the stack pointer down in memory by the amount needed (usually a word), and then saves the value at the new $sp position:
  
  ```
  addi $sp, $sp, -4      # adjust stack for 1 item
  sw  $s0, 0($sp)        # save $s0
  ```

- The function can keep using the stack as long as it needs to, but it should return the stack to its original value at the end of the function, by subtracting or by keeping the frame pointer ($fp) set to the original stack value.

- When a function uses a value on the top of the stack, it is said to “pop” the value off the stack.
Register Usage

- $a0 – $a3: arguments (reg’s 4 – 7)
- $v0, $v1: result values (reg’s 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)
Leaf Procedure Example

- **C code:**
  ```c
  int leaf_example (int g, h, i, j)
  {
      int f;
      f = (g + h) - (i + j);
      return f;
  }
  ```

- Arguments g, ..., j in $a0, ..., $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0
Leaf Procedure Example

C code:

```c
int leaf_example (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

MIPS code:

```mips
addi $sp, $sp, -4
sw $s0, 0($sp)
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
```

- Leaf Example:
  - Procedure body
  - Result
  - Return
  - Save $s0 on stack
  - Restore $s0
Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call
Non-Leaf Procedure Example

- **C code:**
  ```c
  int fact (int n)
  {
    if (n < 1) return 1;
    else return n * fact(n - 1);
  }
  ```
  - Argument n in $a0
  - Result in $v0
**Non-Leaf Procedure Example**

**MIPS code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$sp, $sp, -8</td>
<td># adjust stack for 2 items</td>
</tr>
<tr>
<td>sw</td>
<td>$ra, 4($sp)</td>
<td># save return address</td>
</tr>
<tr>
<td>sw</td>
<td>$a0, 0($sp)</td>
<td># save argument</td>
</tr>
<tr>
<td>slti</td>
<td>$t0, $a0, 1</td>
<td># test for n &lt; 1</td>
</tr>
<tr>
<td>beq</td>
<td>$t0, $zero, L1</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>$v0, $zero, 1</td>
<td># if so, result is 1</td>
</tr>
<tr>
<td>addi</td>
<td>$sp, $sp, 8</td>
<td># pop 2 items from stack</td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
<td># and return</td>
</tr>
<tr>
<td>L1:</td>
<td>addi $a0, $a0, -1</td>
<td># else decrement n</td>
</tr>
<tr>
<td>jal</td>
<td>fact</td>
<td># recursive call</td>
</tr>
<tr>
<td>lw</td>
<td>$a0, 0($sp)</td>
<td># restore original n</td>
</tr>
<tr>
<td>lw</td>
<td>$ra, 4($sp)</td>
<td># and return address</td>
</tr>
<tr>
<td>addi</td>
<td>$sp, $sp, 8</td>
<td># pop 2 items from stack</td>
</tr>
<tr>
<td>mul</td>
<td>$v0, $a0, $v0</td>
<td># multiply to get result</td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
<td># and return</td>
</tr>
</tbody>
</table>
Character Data

- **Byte-encoded character sets**
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters

- **Unicode: 32-bit character set**
  - Used in Java, C++ wide characters, ...
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings
Byte/Halfword Operations

- Could use bitwise operations
- **MIPS byte/halfword load/store**
  - String processing is a common case

```
lb rt, offset(rs)          lh rt, offset(rs)
```
- Sign extend to 32 bits in rt

```
lbu rt, offset(rs)          lhu rt, offset(rs)
```
- Zero extend to 32 bits in rt

```
sb rt, offset(rs)          sh rt, offset(rs)
```
- Store just rightmost byte/halfword
String Copy Example

- **C code (naïve):**
  - Null-terminated string

  ```c
  void strcpy (char x[], char y[])
  {
    int i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
  ```
  - Addresses of `x`, `y` in `$a0`, `$a1`
  - `i` in `$s0`
# String Copy Example

## MIPS code:

```mips
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw $s0, 0($sp)          # save $s0

    add $s0, $zero, $zero  # i = 0

L1:    add $t1, $s0, $a1   # addr of y[i] in $t1
    lbu $t2, 0($t1)        # $t2 = y[i]

    add $t3, $s0, $a0     # addr of x[i] in $t3
    sb $t2, 0($t3)        # x[i] = y[i]

    beq $t2, $zero, L2    # exit loop if y[i] == 0

    addi $s0, $s0, 1      # i = i + 1
    j L1                  # next iteration of loop

L2:    lw $s0, 0($sp)      # restore saved $s0
    addi $sp, $sp, 4      # pop 1 item from stack

    jr $ra                 # and return
```

---

【String Copy Example】

【MIPS code】

```mips
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw $s0, 0($sp)          # save $s0

    add $s0, $zero, $zero  # i = 0

L1:    add $t1, $s0, $a1   # addr of y[i] in $t1
    lbu $t2, 0($t1)        # $t2 = y[i]

    add $t3, $s0, $a0     # addr of x[i] in $t3
    sb $t2, 0($t3)        # x[i] = y[i]

    beq $t2, $zero, L2    # exit loop if y[i] == 0

    addi $s0, $s0, 1      # i = i + 1
    j L1                  # next iteration of loop

L2:    lw $s0, 0($sp)      # restore saved $s0
    addi $sp, $sp, 4      # pop 1 item from stack

    jr $ra                 # and return
```
More About Endian-ness and SPIM

On Little Endian computers, a word is stored such that the lowest order byte comes lowest in memory. In a 32-bit system, this means that the word 0xABCDEF01 followed by the text “abcdefg” looks like this in the SPIM Data tab:

User data segment [10000000]..[10010000]
ABCDEF01 64636261 00676665 00000000

In SPIM, each word is shown in big endian format. In memory it is little endian.
More About Endian-ness and SPIM

User data segment [10000000]..[10010000]
ABCDEF01 64636261 00676665 00000000  0 𐉍 a b c d e

<table>
<thead>
<tr>
<th>Mem Location</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>01</td>
</tr>
<tr>
<td>10000001</td>
<td>EF</td>
</tr>
<tr>
<td>10000002</td>
<td>CD</td>
</tr>
<tr>
<td>10000003</td>
<td>AB</td>
</tr>
<tr>
<td>10000004</td>
<td>‘a’ (0x61)</td>
</tr>
<tr>
<td>10000005</td>
<td>‘b’ (0x62)</td>
</tr>
<tr>
<td>10000006</td>
<td>‘c’ (0x63)</td>
</tr>
<tr>
<td>10000007</td>
<td>‘d’ (0x64)</td>
</tr>
<tr>
<td>10000008</td>
<td>‘e’ (0x65)</td>
</tr>
<tr>
<td>10000009</td>
<td>‘f’ (0x66)</td>
</tr>
<tr>
<td>1000000a</td>
<td>‘g’ (0x67)</td>
</tr>
<tr>
<td>100000b</td>
<td>‘\0’ (null)</td>
</tr>
</tbody>
</table>

Word at 10000000: ABCDEF01

Byte at 10000000: 01
Byte at 10000004: EF  … etc.

• This affects loading and storing bytes!

lui $s0, 0x1001  ; load address into upper part of $at
lbu $t0, 0($1)  ; load value into $t0
lbu $t1, 1($1)  ; load value into $t1

• $t0 now holds 00000001  
• $t1 now holds 000000EF
Many compilers produce object modules directly

Translation and Startup

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

Static linking
Producing an Object Module

- Compiler and assembler translate program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: describes contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code
Linking Object Modules

- **Produces an executable image**
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- **Could leave location dependencies for the loader**
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
Loading a Program

- **Load from image file on disk into memory**
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, ... and calls main
     - When main returns, do exit syscall
Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
Lazy Linkage

- **Indirection table**
- **Stub**: Loads routine ID, Jump to linker/loader
- **Linker/loader code**
- **Dynamically mapped code**
Starting Java Applications

- Java program
- Compiler
- Class files (Java bytecodes)
- Java Virtual Machine
- Compiled Java methods (machine language)
- Interprets bytecodes
- Simple portable instruction set for the JVM

Compiles bytecodes of "hot" methods into native code for host machine
C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```c
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in $a0, k in $a1, temp in $t0
The Procedure Swap

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sll $t1, $a1, 2</code></td>
<td>$t1 = k * 4</td>
</tr>
<tr>
<td><code>add $t1, $a0, $t1</code></td>
<td>$t1 = v+(k*4)</td>
</tr>
<tr>
<td>[\text{# (address of v[k])}]</td>
<td></td>
</tr>
<tr>
<td><code>lw $t0, 0($t1)</code></td>
<td>$t0 (temp) = v[k]</td>
</tr>
<tr>
<td><code>lw $t2, 4($t1)</code></td>
<td>$t2 = v[k+1]</td>
</tr>
<tr>
<td><code>sw $t2, 0($t1)</code></td>
<td>v[k] = $t2 (v[k+1])</td>
</tr>
<tr>
<td><code>sw $t0, 4($t1)</code></td>
<td>v[k+1] = $t0 (temp)</td>
</tr>
<tr>
<td><code>jr $ra</code></td>
<td>return to calling rtn</td>
</tr>
</tbody>
</table>
The Sort Procedure in C

- Non-leaf (calls swap)

```c
void sort (int v[], int n) {
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
            j >= 0 && v[j] > v[j + 1];
            j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in $a0$, n in $a1$, i in $s0$, j in $s1$
The Procedure Body

<table>
<thead>
<tr>
<th>Move params</th>
<th>Outer loop</th>
<th>Inner loop</th>
<th>Pass params &amp; call</th>
<th>Inner loop</th>
<th>Outer loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $s2, $a0</td>
<td>move $s3, $a1</td>
<td>for1tst: slt $t0, $s0, $s3</td>
<td># save $a0 into $s2</td>
<td># save $a1 into $s3</td>
<td># $t0 = 0 if $s0 ≥ $s3 (i ≥ n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for2tst: slti $t0, $s1, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bne $t0, $zero, exit2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sll $t1, $s1, 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add $t2, $s2, $t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw $t3, 0($t2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw $t4, 4($t2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>slt $t0, $t4, $t3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>beq $t0, $zero, exit2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>move $a0, $s2</td>
<td>move $a1, $s1</td>
<td></td>
<td># 1st param of swap is v (old $a0)</td>
<td># 2nd param of swap is j</td>
<td># call swap procedure</td>
</tr>
<tr>
<td>jal swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1, -1</td>
<td></td>
<td></td>
<td># j = i - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j for2tst</td>
<td></td>
<td></td>
<td># jump to test of inner loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit2: addi $s0, $s0, 1</td>
<td></td>
<td></td>
<td># i += 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j for1tst</td>
<td></td>
<td></td>
<td># jump to test of outer loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Full Procedure

<table>
<thead>
<tr>
<th>sort:</th>
<th>addi $sp,$sp, –20</th>
<th># make room on stack for 5 registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sw $ra, 16($sp)</td>
<td># save $ra on stack</td>
</tr>
<tr>
<td></td>
<td>sw $s3,12($sp)</td>
<td># save $s3 on stack</td>
</tr>
<tr>
<td></td>
<td>sw $s2, 8($sp)</td>
<td># save $s2 on stack</td>
</tr>
<tr>
<td></td>
<td>sw $s1, 4($sp)</td>
<td># save $s1 on stack</td>
</tr>
<tr>
<td></td>
<td>sw $s0, 0($sp)</td>
<td># save $s0 on stack</td>
</tr>
<tr>
<td></td>
<td>…</td>
<td># procedure body</td>
</tr>
<tr>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>exit1:</td>
<td>lw $s0, 0($sp)</td>
<td># restore $s0 from stack</td>
</tr>
<tr>
<td></td>
<td>lw $s1, 4($sp)</td>
<td># restore $s1 from stack</td>
</tr>
<tr>
<td></td>
<td>lw $s2, 8($sp)</td>
<td># restore $s2 from stack</td>
</tr>
<tr>
<td></td>
<td>lw $s3,12($sp)</td>
<td># restore $s3 from stack</td>
</tr>
<tr>
<td></td>
<td>lw $ra,16($sp)</td>
<td># restore $ra from stack</td>
</tr>
<tr>
<td></td>
<td>addi $sp,$sp, 20</td>
<td># restore stack pointer</td>
</tr>
<tr>
<td></td>
<td>jr $ra</td>
<td># return to calling routine</td>
</tr>
</tbody>
</table>
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

- **Relative Performance**
- **Instruction count**
- **Clock Cycles**
- **CPI**
Effect of Language and Algorithm

**Bubblesort Relative Performance**

- C/none
- C/O1
- C/O2
- C/O3
- Java/int
- Java/JIT

**Quicksort Relative Performance**

- C/none
- C/O1
- C/O2
- C/O3
- Java/int
- Java/JIT

**Quicksort vs. Bubblesort Speedup**

- C/none
- C/O1
- C/O2
- C/O3
- Java/int
- Java/JIT
Compiler Lessons

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!
ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>15 × 32-bit</td>
<td>31 × 32-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Compare and Branch in ARM

- **Uses condition codes for result of an arithmetic/logical instruction**
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result

- **Each instruction can be conditional**
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions
Instruction Encoding

### Register-register

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>27</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Opx&lt;sup&gt;6&lt;/sup&gt;</td>
<td>Op&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Rs1&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Rd&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Opx&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Rs2&lt;sup&gt;4&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>Op&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Rs1&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Rs2&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Rd&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Opx&lt;sup&gt;8&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Data transfer

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>27</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Opx&lt;sup&gt;6&lt;/sup&gt;</td>
<td>Op&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Rs1&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Rd&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;12&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>Op&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Rs1&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Rd&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;16&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Branch

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Opx&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Op&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;24&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>Op&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Rs1&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Opx&lt;sup&gt;5&lt;/sup&gt;/Rs2&lt;sup&gt;5&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;16&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Jump/Call

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>23</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Opx&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Op&lt;sup&gt;4&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;24&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>Op&lt;sup&gt;8&lt;/sup&gt;</td>
<td>Const&lt;sup&gt;36&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Intel x86 ISA

- **Evolution with backward compatibility**
  - **8080 (1974): 8-bit microprocessor**
    - Accumulator, plus 3 index-register pairs
  - **8086 (1978): 16-bit extension to 8080**
    - Complex instruction set (CISC)
  - **8087 (1980): floating-point coprocessor**
    - Adds FP instructions and register stack
  - **80286 (1982): 24-bit addresses, MMU**
    - Segmented memory mapping and protection
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments
The Intel x86 ISA

- **Further evolution…**
  - **i486 (1989):** pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, ...
  - **Pentium (1993):** superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - **Pentium Pro (1995), Pentium II (1997):**
    - New microarchitecture (Colwell, The Pentium Chronicles)
  - **Pentium III (1999):**
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - **Pentium 4 (2001):**
    - New microarchitecture
    - Added SSE2 instructions
The Intel x86 ISA

- **And further...**
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead...
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions

- **If Intel didn’t extend with compatibility, its competitors would!**
  - Technical elegance ≠ market success
Basic x86 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
Basic x86 Addressing Modes

- Two operands per instruction

<table>
<thead>
<tr>
<th>Source/dest operand</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

- Memory addressing modes
  - Address in register
  - Address = $R_{base} + \text{displacement}$
  - Address = $R_{base} + 2^{\text{scale}} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{\text{scale}} \times R_{index} + \text{displacement}$
x86 Instruction Encoding

- **Variable length encoding**
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...

### Instruction Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Length</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. JE EIP + displacement</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>JE Condition Displacement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. CALL</td>
<td>8</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>CALL Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. MOV EBX, [EDI + 45]</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV d w r/m Postbyte Displacement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d. PUSH ESI</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PUSH Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e. ADD EAX, #6765</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>ADD Reg w Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f. TEST EDX, #42</td>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>TEST w Postbyte Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing IA-32

- **Complex instruction set makes implementation difficult**
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable

- **Comparable performance to RISC**
  - Compilers avoid complex instructions
Fallacies

- **Powerful instruction ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- **Use assembly code for high performance**
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity
Fallacies

- Backward compatibility ⇒ instruction set doesn’t change
  - But they do accrete more instructions
Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
  - Keeping a pointer to an automatic variable after procedure returns
    - In C, this might look like the following:
      ```c
      int *myFunction() {
        int newVar = 3;
        return &newVar;
      }
      ```
    - e.g., passing pointer back via an argument
    - Pointer becomes invalid when stack popped
ISA Design Principles

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

- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
Instruction Frequencies

- **Measure MIPS instruction executions in benchmark programs**
  - Consider making the common case fast
  - Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Intro to ISAs and MIPS: Summary

- **ISA**
  - Instructions supported by a processor family (e.g. x86)
  - Generally simple instructions
  - Explicitly manage registers vs memory (but not caches)

- **MIPS**
  - A sample ISA
  - Used in embedded systems
  - Used in this course!

[ch2 Fin]
Homework #2 Info / Lab #1

- **HW Due next Monday**
  - 2.6.1.b:
    - \( f = A[B[g]+1] \)
    - \( B[g] \) returns an int, to be increment by 1
  - 2.6.6 – only the instructions listed at the top of page 185 (not all the instructions on the MIPS green card!)

- **Lab #1**
  - This is a 2-week lab, so if you don’t finish this week, work on it next week.
  - You may also work on it at home, but get it checked off (and hand in) by the TAs.
Guy Steele

- “The Great Quux” (i.e., uber-nerd)
- Designed the Scheme Lisp dialect
- Edited (and illustrated) “The Hacker’s Dictionary”
- His talk:
  - Assembly language computing on the IBM 1130, a 1960s era ‘cheap’ computer (~$41,000).
  - Input: punch card (300 cards/min!), Output: line printer with a “Selectric-type ball” print-head.
  - “Re-writing code”
  - “Before stacks”
  - Interrupts
  - One 32-bit register
- [http://www.infoq.com/presentations/Thinking-Parallel-Programming](http://www.infoq.com/presentations/Thinking-Parallel-Programming)
Synchronization

- **Two processors sharing an area of memory**
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends of order of accesses

- **Hardware support required**
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- **Could be a single instruction**
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Synchronization in MIPS

- **Load linked**: `ll rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`

- **Store conditional**: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`

- **Example**: atomic swap (to test/set lock variable)

  ```assembly
  try: add $t0,$zero,$s4 ;copy exchange value
  ll $t1,0($s1)    ;load linked
  sc $t0,0($s1)    ;store conditional
  beq $t0,$zero,try ;branch store fails
  add $s4,$zero,$t1 ;put load value in $s4
  ```
Arrays vs. Pointers

- **Array indexing involves**
  - Multiplying index by element size
  - Adding to array base address

- **Pointers correspond directly to memory addresses**
  - Can avoid indexing complexity
### Example: Clearing and Array

| clear1(int array[], int size) {  
| int i;  
| for (i = 0; i < size; i += 1)  
| array[i] = 0;  
| }  
| clear2(int *array, int size) {  
| int *p;  
| for (p = &array[0]; p < &array[size];  
| p = p + 1)  
| *p = 0;  
| }  
|   
| move $t0,$zero   # i = 0  
| loop1: sll $t1,$t0,2   # $t1 = i * 4  
| add $t2,$a0,$t1   # $t2 =  
| # &array[i]  
| sw $zero, 0($t2) # array[i] = 0  
| addi $t0,$t0,1   # i = i + 1  
| slt $t3,$t0,$a1   # $t3 =  
| # (i < size)  
| bne $t3,$zero,loop1 # if (...)  
| # goto loop1  
|   
| move $t0,$a0   # p = & array[0]  
| sll $t1,$a1,2   # $t1 = size * 4  
| add $t2,$a0,$t1   # $t2 =  
| # &array[size]  
| loop2: sw $zero,0($t0) # Memory[p] = 0  
| addi $t0,$t0,4   # p = p + 4  
| slt $t3,$t0,$t2   # $t3 =  
| #(p<&array[size])  
| bne $t3,$zero,loop2 # if (...)  
| # goto loop2
Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer