Chapter 2

Instructions: Language of the Computer
Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets
The MIPS Instruction Set

- Used as the example throughout the book
- 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
  - See MIPS Reference Data tear-out card, and Appendixes B and E
Instruction Categories

- Computational
- Load/Store
- Jump and Branch
- Floating Point (coprocessor)
- Memory Management
- Special

3 Instruction Formats: all 32 bits wide

- **R format**
  - op
  - rs
  - rt
  - rd
  - shamt
  - funct
- **I format**
  - op
  - rs
  - rt
  - immediate
- **J format**
  - op
  - jump target
MIPS Instruction Classes Distribution

Frequency of MIPS instruction classes for SPEC2006

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>16%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>35%</td>
</tr>
<tr>
<td>Logical</td>
<td>12%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>34%</td>
</tr>
<tr>
<td>Jump</td>
<td>2%</td>
</tr>
</tbody>
</table>
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 favours regularity*

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost
Arithmetic Example

- **C code:**
  
  \[ f = (g + h) - (i + j); \]

- **Compiled MIPS code:**
  
  ```
  add t0, g, h  # temp t0 = g + h
  add t1, i, j  # temp t1 = i + j
  sub f, t0, t1  # f = t0 - t1
  ```
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 (convention)
  - $t0, t1, \ldots, t9$ for temporary values
  - $s0, s1, \ldots, s7$ for saved variables
- Design Principle 2: Smaller is faster
  - (main memory: millions of locations)
MIPS Register File (1)

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
MIPS Register File (2)

- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since registers are named with fewer bits than a memory location)
Register Operand Example

- **C code:**
  
  \[ f = (g + h) - (i + j); \]

  - \( f, \ldots, j \) in \( s_0, \ldots, s_4 \)

- **Compiled MIPS code:**
  
  ```
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1
  ```
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 least address of a word
  - Little Endian: least-significant byte at least address
Big Endian / Little Endian

- **Big Endian:**
  leftmost byte is word address
  IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA

- **Little Endian:**
  rightmost byte is word address
  Intel 80x86, DEC Vax, DEC Alpha

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

- *Big endian byte 0*
- *Little endian byte 0*
Memory Operand Example 1

- **C code:**
  
  ```
  g = h + A[8];
  ```
  
  - g in $s1, h in $s2, base address of A in $s3

- **Compiled MIPS code:**
  
  ```
  lw $t0, 32($s3) # load word
  add $s1, $s2, $t0
  ```
  
  - Index 8 requires offset of 32
  - 4 bytes per word
  
  ![Diagram showing offset and base register in MIPS code]
Memory Operand Example 2

- C code:
  \[ A[12] = h + A[8]; \]
  - 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 memory data 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**
  \[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_12^1 + x_02^0 \]

- **Range:** 0 to +2^n – 1

- **Example**
  - 0000 0000 0000 0000 0000 0000 0000 1011_2
  - \(= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\)
  - \(= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}\)

- **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
- $-(-2^{n-1})$ can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 ... 0000
  - $-1$: 1111 1111 ... 1111
  - Most-negative: 1000 0000 ... 0000
  - Most-positive: 0111 1111 ... 1111
Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
    
    \[
    x + \bar{x} = 1111 \ldots 111_2 = -1
    \]
    \[
    \bar{x} + 1 = -x
    \]

- Example: negate +2
  - +2 = 0000 0000 \ldots 0010_2
  - −2 = 1111 1111 \ldots 1101_2 + 1
    = 1111 1111 \ldots 1110_2
Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - addi: extend immediate value
  - l b, l h: extend loaded byte/halfword
  - beq, bne: extend the displacement
- Replicate the sign bit to the left
  - compare 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
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 (convention)
  - $t0 – $t7 are reg’s 8 – 15
  - $t8 – $t9 are reg’s 24 – 25
  - $s0 – $s7 are reg’s 16 – 23
## MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26-27</td>
<td>reserved for OS</td>
<td></td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Instruction Categories

- Computational
- Load/Store
- Jump and Branch
- Floating Point
  - coprocessor
- Memory Management
- Special
3 Instruction Formats: all 32 bits wide

- **R format**
  - `op` 5 bits
  - `rs` 5 bits
  - `rt` 5 bits
  - `rd` 5 bits
  - `shamt` 5 bits
  - `funct` 6 bits

- **I format**
  - `op` 6 bits
  - `rs` 5 bits
  - `rt` 5 bits
  - `immediate` 16 bits

- **J format**
  - `op` 6 bits
  - `jump target` 26 bits
MIPS Instruction Fields

- MIPS fields are given names to make them easier to refer to

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6</td>
<td>opcode that specifies the operation</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
<td>register file address of the first source operand</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
<td>register file address of the second source operand</td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
<td>register file address of the result’s destination</td>
</tr>
<tr>
<td>shamt</td>
<td>5</td>
<td>shift amount (for shift instructions)</td>
</tr>
<tr>
<td>funct</td>
<td>6</td>
<td>function code augmenting the opcode</td>
</tr>
</tbody>
</table>
### R-format Example

#### Table

<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>

#### Instruction

**add** \( \$t0, \$s1, \$s2 \)

#### Special Case

<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>

\[000000|10001|10010|01000|00000|100000_2 = 02324020_{16}\]
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>c</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>e</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
Load/Store (I format)

`lw $t0, 24($s3)`

Memory data word address (hex)

| 0x00000000 | 0x00000004 | 0x00000008 | 0x0000000c | $f f f f f f f |

$24_{10} + $s3 =

\[ \ldots 0001 \ 1000 + \ldots 1001 \ 0100 = \ldots 1010 \ 1100 = 0x120040ac \]

Memory:

- $t0$: 0x120040ac
- $s3$: 0x12004094
- Others:
  - 0x00000000
  - 0x00000008
  - 0x00000004
  - 0x00000000

182.690 Rechnerstrukturen 2 – Instructions: Language of the Computer
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits

- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes

- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

- Good design demands good compromises
  - three instruction formats
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
## 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
- $sll$ 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

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0$</td>
<td>0000 0000 0000 0000 0000 1100 0000 0000</td>
</tr>
<tr>
<td>$t1$</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t2$</td>
<td>0000 0000 0000 0000 0000 1100 0000 0000</td>
</tr>
</tbody>
</table>
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>Variable</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t2</td>
<td>0000 0000 0000 0000 0000 0000 1101 1100 0000</td>
</tr>
<tr>
<td>$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
  - \( a \text{ NOR } b = \text{ NOT ( } a \text{ OR } b \text{ )} \)

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
</tbody>
</table>

Register 0: always read as zero
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `beq rs, rt, L1`
  - if \((rs == rt)\) branch to instruction labeled \(L1\);

- `bne rs, rt, L1`
  - if \((rs != rt)\) branch to instruction labeled \(L1\);

- `j L1`
  - unconditional jump to instruction labeled \(L1\)
Compiling If Statements

- **C code:**

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

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

- **Compiled MIPS code:**

  ```mips
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

  **Assembler calculates addresses**

182.690 Rechnerstrukturen 2 – Instructions: Language of the Computer
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
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
  ```
Branch Instruction Design

- Why not bl t, 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

- \( -1 < +1 \Rightarrow t0 = 1 \)

`sltu $t0, $s0, $s1` \# unsigned

- \( +4,294,967,295 > +1 \Rightarrow t0 = 0 \)
Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
sltu $t0, $s1, $t2  # $t0 = 0 if
    # $s1 > $t2 (max)
    # or $s1 < 0 (min)
beq $t0, $zero, IOOB  # go to IOOB if
    # $t0 = 0
```

- The key is that negative integers in two’s complement look like large numbers in unsigned notation. Thus, an unsigned comparison of $x < y$ also checks if $x$ is negative as well as if $x$ is less than $y$
Procedure Calling

Main routine (caller) places parameters in a place where the procedure (callee) can access them

- $a0 - a3$: four argument registers (convention)

Caller transfers control to the callee

1. Callee acquires the storage resources needed
2. Callee performs the desired task
3. Callee places the result value in a place where the caller can access it
   - $v0 - v1$: two value registers for result values (convention)
4. Callee returns control to the caller
   - $ra$: one return address register to return to the point of origin
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)
Procedure Call Instructions

- **Procedure call: jump and link**
  ```assembly
  jal ProcedureLabel
  ```
  - Address of following instruction put in $ra
  - Jumps to target address

- **Procedure return: jump register**
  ```assembly
  jr $ra
  ```
  - Copies $ra to program counter
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

**MIPS code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Register</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$sp</td>
<td>$sp</td>
<td>-4</td>
</tr>
<tr>
<td>sw</td>
<td>$s0</td>
<td>0</td>
<td>($sp)</td>
</tr>
<tr>
<td>add</td>
<td>$t0</td>
<td>$a0</td>
<td>$a1</td>
</tr>
<tr>
<td>add</td>
<td>$t1</td>
<td>$a2</td>
<td>$a3</td>
</tr>
<tr>
<td>sub</td>
<td>$s0</td>
<td>$t0</td>
<td>$t1</td>
</tr>
<tr>
<td>add</td>
<td>$v0</td>
<td>$s0</td>
<td>$zero</td>
</tr>
<tr>
<td>lw</td>
<td>$s0</td>
<td>0</td>
<td>($sp)</td>
</tr>
<tr>
<td>addi</td>
<td>$sp</td>
<td>$sp</td>
<td>4</td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Save $s0 on stack

Procedure body

Result

Restore $s0

Return
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 f;
    else return n * fact(n - 1);
}
```

- Argument n in $a0
- Result in $v0
Non-Leaf Procedure Example

- **MIPS code:**

```mips
fact:
  addi $sp, $sp, -8  # adjust stack for 2 items
  sw $ra, 4($sp)    # save return address
  sw $a0, 0($sp)    # save argument
  slti $t0, $a0, 1  # test for n < 1
  beq $t0, $zero, L1
  addi $v0, $zero, 1  # if so, result is 1
  addi $sp, $sp, 8  # pop 2 items from stack
  jr $ra  # and return
L1:  addi $a0, $a0, -1  # else decrement n
    jal fact  # recursive call
  lw $a0, 0($sp)  # restore original n
  lw $ra, 4($sp)  # and return address
  addi $sp, $sp, 8  # pop 2 items from stack
  mul $v0, $a0, $v0  # multiply to get result
  jr $ra  # and return
```

Local Data on the Stack

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
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
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
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient

- For the occasional 32-bit constant
  - \texttt{lui} \text{ rt, constant}
    - Copies 16-bit constant to left 16 bits of \text{rt}
    - Clears right 16 bits of \text{rt} to 0

\texttt{lui} $s0, 61

\texttt{ori} $s0, $s0, 2304
Branch Addressing

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

- Most branch targets are near branch
  - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
Specifying Branch Destinations

- Instruction Address Register (the PC) implicitly used. PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction!
- Limits the branch distance to \(-2^{15}\) to \(+2^{15}-1\) (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway.

![Diagram showing branch destination calculation]

from the low order 16 bits of the branch instruction

PC + offset \(\rightarrow\) branch dst address

182.690 Rechnerstrukturen 2 – Instructions: Language of the Computer
MIPS conditional branch instructions:

- `bne $s0, $s1, Lbl`  # go to Lbl if $s0 ≠ $s1
- `beq $s0, $s1, Lbl`  # go to Lbl if $s0 = $s1

Ex:  

```
if (i==j) h = i + j;
```

```
bne $s0, $s1, Lbl1
add $s3, $s0, $s1
Lbl1: ... 
```

Instruction Format (I format):

<table>
<thead>
<tr>
<th></th>
<th>0x05</th>
<th>16</th>
<th>17</th>
<th>16 bit offset</th>
</tr>
</thead>
</table>

182.690 Rechnerstrukturen 2 – Instructions: Language of the Computer
Jump Addressing

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

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address × 4)
from the low order 26 bits of the jump instruction

PC region branch: effective target address is in the “current” 256 MB-aligned region
## Target Addressing Example

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Target</th>
<th>Immediate</th>
<th>80000</th>
<th>80004</th>
<th>80008</th>
<th>80012</th>
<th>80016</th>
<th>80020</th>
<th>80024</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>0 0 19</td>
<td>9 4 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0 9 22</td>
<td>9 0 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>35 9 8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>5 8 21</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>8 19 19</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2</td>
<td>20000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop: ...
Branching Far Away

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

Example

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

1. Immediate addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Immediate} \]

2. Register addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \ldots \quad \text{func} \]
   Registers
   Register

3. Base addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \]
   Registers
   Memory
   Byte
   Halfword
   Word

4. PC-relative addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \]
   Memory
   Word

5. Pseudodirect addressing
   \[ \text{op} \quad \text{Address} \]
   Memory
   Word
Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on the 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)`
  - Returns 1 in rt
  - Returns 0 in rt

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

- **Example: atomic swap (to test/set lock variable)**
  ```
  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
  ```
Many compilers produce object modules directly.
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
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

- Relative Performance
- Instruction count
- Clock Cycles
- CPI

182.690 Rechnerstrukturen 2 – Instructions: Language of the Computer
Effect of Language and Algorithm

Bubblesort Relative Performance

Quicksort Relative Performance

Quicksort vs. Bubblesort Speedup
Compiler Benefits

- Comparing performance for bubble (exchange) sort

<table>
<thead>
<tr>
<th>gcc opt</th>
<th>Relative performance</th>
<th>Clock cycles (M)</th>
<th>Instr count (M)</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>1.00</td>
<td>158,615</td>
<td>114,938</td>
<td>1.38</td>
</tr>
<tr>
<td>O1 (medium)</td>
<td>2.37</td>
<td>66,990</td>
<td>37,470</td>
<td>1.79</td>
</tr>
<tr>
<td>O2 (full)</td>
<td>2.38</td>
<td>66,521</td>
<td>39,993</td>
<td>1.66</td>
</tr>
<tr>
<td>O3 (proc mig)</td>
<td>2.41</td>
<td>65,747</td>
<td>44,993</td>
<td>1.46</td>
</tr>
</tbody>
</table>

- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest.
Lessons Learnt

- 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>
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
- 80386 (1985): 32-bit extension (now IA-32)
  - 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
    - New microarchitecture (see 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 = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index} + \text{displacement}$
x86 Instruction Encoding

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...
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 $\Rightarrow$ 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 $\Rightarrow$ more errors and less productivity
Fallacies

- Backward compatibility $\Rightarrow$ 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!
Concluding Remarks

- 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
  - c.f. x86
## Concluding Remarks

- **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>