MIPS Instructions

• Instructions: Language of the Machine
• Instruction set architecture - the attributes of a computer system seen by the programmer
• Attributes
  – organization of programmable storage
  – data types and data structures, encodings, representations
  – instruction format, operands
  – instruction set (operation code set)
  – addressing modes, accessing data items and instructions
  – exceptional (interrupt) conditions
• We’ll be working with the MIPS instruction set architectures.
• MIPS: A typical RISC machine, 50M MIPS chips sold, used in computers, printers, network cards, video game (Nintendo).

Why do we study MIPS Instruction Set Architecture?

<table>
<thead>
<tr>
<th>Instruction set</th>
<th>Number sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>80x86</td>
<td>&gt; 50,000,000</td>
</tr>
<tr>
<td>MIPS</td>
<td>&gt; 50,500,000</td>
</tr>
<tr>
<td>PowerPC</td>
<td>&gt; 3,300,000</td>
</tr>
<tr>
<td>SPARC</td>
<td>&gt; 700,000</td>
</tr>
<tr>
<td>HP PA-RISC</td>
<td>&gt; 300,000</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>&gt; 200,000</td>
</tr>
</tbody>
</table>

Processor Performance Increase
Where is the Market?

- Power consumption – especially in the embedded market where battery life is important (and passive cooling)
  - For power-limited applications, the most important metric is energy efficiency

Specification - bottom-up

- Instruction Format or Encoding
  - How is it decoded?
- Location of the operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor Instruction
  - jumps, conditions, branches

RISC vs CISC

- RISC (reduced instruction set computer) instructions
  - only load/store instructions access memory
  - data (i.e., operands) must be in registers to perform operation
  - each instruction roughly taking same amount of time
  - simple addressing modes
  - virtually all new instruction sets since 1982 have been RISC (M 68000 announced in year 1980)
  - etc.
- CISC (complex instruction set computer) instructions
  - alu/logic instructions access memory to fetch operands
  - load/store instructions access memory
  - some instructions’ execution time is much longer than other instructions
  - complex addressing modes
  - etc.
RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, ...
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

MIPS Instructions

- MIPS is a RISC machine
- programmable storage: 32 32-bit general purpose registers (GPR), represented as $0, $1, $2, ..., $31, the content of $0 is always $0.
- 2^th memory words, such as Memory[0], Memory[4], etc. Memory holds data structures such as arrays, and spilled registers
- MIPS uses byte addresses so sequential words differ by 4
- 32 32-bit floating point registers, paired for double precision

What is a spilled register?
A variable is less commonly used and saved in memory rather than in registers

MIPS Addressing Mode

- Simple addressing modes
- All instructions 32 bits wide
- register addressing
  ```
  op   rs   rt   rd  --  --
  ; rs, rt specify the operand (i.e., register number) directly
  ```
- immediate addressing
  ```
  op   rs   rt   immediate
  ; immediate is the operand
  ```
- base addressing (base + offset or displacement)
  ```
  op   rs   rt   immediate
  ; rt + immediate is the address where the operand is located
  ```
- PC-relative
  ```
  op   rs   rt   immediate
  ; PC + immediate is the address
  ```

MIPS Instructions

- R type instruction

<table>
<thead>
<tr>
<th>no. of bits</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Some arithmetic and logic instructions are R-type

and $1, $2, $3 ; $1 = $2 & $3
add $1, $2, $3 ; $1 = $2 + $3
slt $1, $2, $3 ; if ($2 < $3) then $1 = 1; else $1 = 0
addu $1, $2, $3 ; add unsigned, no exceptions
sub $1, $2, $3 ;
subu $1, $2, $3 ; sub unsigned, no exceptions
MIPS Arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)
  Example:
  C code: \( A = B + C \)
  MIPS code: \( \text{add } $s0, $s1, $s2 \)
  (associated with variables by compiler)
  MIPS compiler uses $s0, $s1, ... for registers that correspond to variables in C program and $t0, $t1, ...
  for temporary registers needed to compile a program
- Another Example
  C code: \( A = B + C + D; \quad E = F - A; \)
  MIPS code: \( \text{add } $t0, $s1, $s2 \)
  \( \text{add } $s0, $t0, $s3 \)
  \( \text{sub } $s4, $s5, $s0 \)
- Operand must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

Register vs. Memory

- Arithmetic instructions operands must be registers,
  - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables
  - use memory to store variables when all registers are used.
- Memory viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- “Byte addressing” means that the index points to a byte of memory.
- For MIPS, a word is 32 bits or 4 bytes.
  \( 2^{32} \) words with byte addresses 0, 4, 8, ... \( 2^{32} - 4 \)
  \( 2^{32} \) bytes with byte addresses from 0 to \( 2^{32} \) - 1
- Words are aligned

MIPS Instructions

- I type instruction; small constants are used frequently
  \begin{align*}
  \text{op} & \quad \text{rs} & \quad \text{rt} & \quad \text{immediate} \\
  \text{no.of bits} & \quad 6 & \quad 5 & \quad 5 & \quad 16
  \end{align*}
  instruction examples
  \begin{align*}
  \text{add immediate} & \\
  \text{add$1, $2, 100; } & \text{immediate = 100, rs = $2, rt = $1; op =8} \\
  & \text{; $1 = $2 + 100} \\
  \text{and immediate} & \\
  \text{and$1, $2, 100; } & \text{immediate = 100, rs = $2, rt = $1; op = 12} \\
  & \text{; $1 = $2 & 100}
  \end{align*}

Good design demands good compromises.
Where is the compromise here?
(rt used as destination reg.) (rt used as operand reg. in R type)
MIPS Instructions

• I type instructions
  \[ \text{lw } $1, 100($3); \text{ rs = }$3, \text{ immediate = } 100, \text{ rt = }$1 \]
  \[ $1 = \text{Memory}[100 + 3] \]
  \[ \text{sw } $1, 150($2); \text{ rs = }$2, \text{ immediate = } 150, \text{ rt = }$1 \]
  \[ \text{Memory}[2 + 150] = $1 \]
  \[ \text{lui }$1, 108; \text{ immediate = } 108 \text{ rs = }0, \text{ rt = }$1; \text{ load 108 into upper 16 bits of register }$1, \text{$1 = 108 \times 2^{16} \]

• J type instruction
  \[ \text{op target address } \]
  \[ \text{no. of bits } 6 \quad 26 \]
  \[ \text{jump instruction;} \]
  \[ \text{j 10000; go to target address 10000} \]
  \[ \text{jal 10000; }$31 = \text{PC }+ 4; \text{ go to 10000; for procedure call,} \]
  \[ ; \text{return address is saved in }$31 \]

• Instructions for making decisions
  – alter the control flow,
  – change the “next” instruction to be executed

  \[ \text{beq }$1, $2, L1; \text{ (I type )} \]
  \[ \text{go to the instruction labeled L1, if the value in }$1 \text{ equals the value in }$2 \]

  \[ \text{bne }$3, $4, L1; \text{ (I type)} \]
  \[ \text{go to the instruction labeled L1, if the value in }$3 \text{ not equal the value in }$4 \]

  – example: C code

  \[ \text{if } (i==j) \text{ } h = i+j; \]

  \[ \text{MIPS code } \]
  \[ \text{beq }$s0, $s1, Label \]
  \[ \text{add }$s3, $s4, $s5 \]
  \[ \text{.....} \]
  \[ \text{Label: } \text{.....} \]

• Control
  – MIPS unconditional branch instructions:

  \[ \text{j label} \]

  \[ \text{Example:} \]
  \[ \text{if } (i \! = \! j) \text{ beq }$s4, $s5, Lab1 \]
  \[ \text{h = i }+ \text{ j; add }$s3, $s4, $s5 \]
  \[ \text{else j Lab2} \]
  \[ \text{h = i }- \text{ j; Label1: sub }$s3, $s4, $s5 \]
  \[ \text{Lab2: } \text{.........} \]

• Other cases such as compiling a while loop, a switch statement, a loop with array index, a procedure call, nested procedures into MIPS code (skip, this is too much into assembly language programming)

• This course is to design a processor
Addressing Objects

- Big Endian: address of most significant byte = word address, (xxx...00 = big end of the word)
  - IBM, Motorola 68K, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address, (xxx...00 = little end of the word)
  - Intel 80x86, DEC Vax, DEC Alpha
- alignment: require that objects fall on address that is multiple of their size

<table>
<thead>
<tr>
<th>leftmost byte is word address</th>
<th>Big Endian</th>
<th>rightmost byte is word address</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 0</td>
<td>byte 0</td>
<td>byte 1</td>
<td>byte 2</td>
</tr>
<tr>
<td>word 0</td>
<td>byte 3</td>
<td>byte 2</td>
<td>byte 1</td>
</tr>
</tbody>
</table>

Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- 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 (Windows NT)

Data Types

- Bit
- Bit String:
  - 4 bits is a nibble
  - 8 bits is a byte
  - 16 bits is a half word
  - 32 bits is a word
  - 64 bits is a double-word
- Character: ASCII 7 bit code
- Decimals: digits 0-9 encoded as 0000 through 1001
- Integers: 2’s complement
- Floating Point:
  - single precision, double precision

MIPS Instructions

- MIPS arithmetic instructions
  - add, addi, addu, sub, subi, subu, addiu
  - all arithmetic operations operate on words

Which add for address arithmetic?
Which add for integers?
MIPS Instructions

- **Logical Instructions**
  - and, or, xor, nor, andi (and immediate), ori (or immediate),
  - sll (shift left logical), srl (shift right logical), sra (shift right arithmetic),
- **Examples**
  - sll $1, $2, 10 ; $1 = $2 << 10 (shift left logical 10 bits of $2)
  - srl $1, $2, 10 ; $1 = $2 >> 10 (shift right logical 10 bit)
  - or $1, $2, $3 ; $1 = $2 | $3
  - and $1, $2, $3 ; $1 = $2 & $3
  - ori $2, $3, 99 : $2 = $3 | 99

- **data transfer instructions**
  - lw load word
  - sw store word
  - lbu load byte unsigned
  - sb store byte
  - lui load upper immediate
  - lui $1, 100 ; $1 = 100 x 2^16

- **Signed vs Unsigned Comparison**

  
  $1 = \text{0000 0000 0000}......00001$
  
  $2 = \text{1111 1111 1111}.......11111$
  
  $3 = \text{0000 0000 0000}.......00000$

  
  **Signed**
  
  slt $4, $1, $2
  
  slti $5, $1, 79

  **Unsigned**
  
  sltu $6, $2, $3
  
  sltiu $7, $2, $3

- **jump register**
  - jr $31 ; go to register $31

- **Examples**

  
  $4= \text{_________}$; $5 = \text{_________}$
  
  $6= \text{_________}$; $7 = \text{_________}$
MIPS Instructions

- $0$ is always zero (you can not change it)
- Branch and link save the return address PC+4 into $31$
- All instructions change all 32 bits of the destination register
- Logic immediates are zero extended to 32 bits
- Arithmetic immediates are sign extended to 32 bits
- Data loaded by the instructions lb, lh are extended as follows:
  - lb, lh are zero extended (unsigned load)
  - lbu, lhu are zero extended (unsigned load)
- Overflow can occur in signed arithmetic/logic instructions, will not occur in unsigned arithmetic/logic instructions.

MIPS Summary

- Fixed instruction format (3 formats: R, I, J)
- 3-address, reg-reg arithmetic/logical instructions
- Single addressing mode for load/store instructions
  - base + displacement
- Simple branch conditions
  - Compare against zero or two registers for equal, not_equal, no integer condition code
- Delayed branch (more examples in chapter 6)
  - Execute instruction after the branch (or jump) even if the branch is not taken. Compiler fills a delayed branch with a useful instruction (90% correct).
- Three design principles:
  - Simplicity favors regularity
  - Smaller is faster
  - Good design demands good compromises

All Time Popular Instruction Set - 80x86

- 1978: 8086 is announced (16 bit architecture)
- 1980: 8087 floating point coprocessor is added
- 1982: 80286 increases address space to 24 bits + some instructions
- 1985: 80386 extends to 32 bits, new addressing modes
- 1989 - 1995: 80486, Pentium, Pentium Pro add a few instructions, mostly designed for higher performance
- 1997: MMX is added
- 1999: Intel added another 70 instructions labeled as SSE (Streaming SIMD Extensions) as part of Pentium III
- 2001: Intel added another 144 instructions as SSE II
- 2003: IA-32 (but AMD announced AMD64)
- 2004: Intel capitulates and embraces AMD64.

All Time Popular Instruction Set: 80x86

Complexity of 80x86 instructions from 1 to 17 bytes long
one operand must act as both a source and destination
one operand can come from memory
complex addressing modes

This history illustrates the impact of the “golden handcuffs” of compatibility on 80x86. The existing software base was too important to jeopardize with significant architectural changes.
MIPS Summary

more powerful instructions mean higher performance T F
writing assembly language in order to obtain the highest performance T F
smaller is faster T F
simplicity favors regularity T F
good design demands compromise T F
make the common case fast T F
evaluate instruction sets
  – design time metrics: how long at what cost to implement
  – static metrics: how many bytes program occupy memory
  – dynamic metrics: