The Instruction Set Architecture (ISA)

Goals

- Understand how programs are encoded
  - What does the OS (loader) see?
- Impact of ISA on program encodings
- Why are ISAs different?
Reading

- Chapter 2
  - 2.1, Figure 2.1, 2.2 – 2.7
  - 2.9, Figure 2.15 (2.9.1 in online text)
  - 2.10, 2.16, 2.17
- Appendix A9, A10
- Practice Problems: 1,2,8, 15

Module Outline

Review ISA (MIPS) and understand instruction encodings

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls – Not Covered here - Review on your own from ECE 2035 (will not be tested)
Instruction Set Architecture

- A very important abstraction
  - interface between hardware and low-level software
  - *standardizes* instructions, machine language bit patterns, etc.
  - advantage: *different implementations of the same architecture*
  - disadvantage: *sometimes prevents using new innovations*

- Modern instruction set architectures:
  - 80x86 (aka iA32), PowerPC (e.g. G4, G5)
  - Xscale, ARM, MIPS
  - Intel/HP EPIC (iA64), AMD64, Intel’s EM64T, SPARC, HP PA-RISC, DEC/Compaq/HP Alpha

Economics of an ISA

- Thermal Design Power 130W
  - 3.6 GHz
- Thermal Design Power 4W
  - 1.6 GHz

*Software/binary portability*
Instructions

• We’ll be working with the MIPS instruction set architecture
  ▶ Representative of Reduced Instruction Set Computer (RISC)
  ▶ Similar to other architectures developed since the 1980's
  ▶ Used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: Maximize performance and Minimize cost, Reduce design time

Instruction Set Architecture (ISA)

RISC vs. CISC
Basic Principles

- What are the operations supported in hardware → opcodes
- Where are the source and destination data or operands for the instruction?
  - Memory, registers, caches, etc.
- How is the location of the operands determined
  - Address in memory, and register name
  - Addressing modes

Module Outline

Review ISA and understand instruction encodings
- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding
MIPS Programmer Visible Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Names</th>
<th>Usage by Software Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$zero</td>
<td>Hardwired to zero</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>Reserved by assembler</td>
</tr>
<tr>
<td>$2 - $3</td>
<td>$v0 - $v1</td>
<td>Function return result registers</td>
</tr>
<tr>
<td>$4 - $7</td>
<td>$a0 - $a3</td>
<td>Function passing argument value registers</td>
</tr>
<tr>
<td>$8 : $15</td>
<td>$t0 : $t7</td>
<td>Temporary registers, caller saved</td>
</tr>
<tr>
<td>$16 : $23</td>
<td>$s0 : $s7</td>
<td>Saved registers, callee saved</td>
</tr>
<tr>
<td>$24 : $25</td>
<td>$t8 - $t9</td>
<td>Temporary registers, caller saved</td>
</tr>
<tr>
<td>$26 - $27</td>
<td>$k0 - $k1</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>Return address (pushed by call instruction)</td>
</tr>
<tr>
<td>$32</td>
<td>$hi</td>
<td>High result register (remainder/div, high word/mult)</td>
</tr>
<tr>
<td>$33</td>
<td>$lo</td>
<td>Low result register (quotient/div, low word/mult)</td>
</tr>
</tbody>
</table>

MIPS Register View

- Arithmetic instruction operands must be registers
- Compiler associates variables with registers
- Other registers that are not visible to the programmer,
  - Program counter
  - Status register
  - ......
  - Kernel registers
  - Page table pointer

Program execution control
Operating system state
**Instruction Set Architecture (ISA)**

- **Register File**
  - 0x00
  - 0x01
  - 0x02
  - 0x03
  - 0x1F

- **Data flow for computation**

- **Arithmetic Logic Unit (ALU)**

- **Address Space**

- **RISC vs. CISC**

---

**MIPS arithmetic**

- **Design Principle 1**: simplicity favors regularity.
- Of course this complicates some things...

  C code: \[ A = B + C + D; \]
  \[ E = F - A; \]

  MIPS code:
  
  - `add $t0, $s1, $s2`
  - `add $s0, $t0, $s3`
  - `sub $s4, $s5, $s0`
  - `andi $3, $4, $5`

  Note the need for intermediate registers

- Operands must be registers, only 32 registers provided
- All memory accesses accomplished via loads and stores
  - A common feature of RISC processors
Logical Operations

• Instructions for bitwise manipulation

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

- Useful for extracting and inserting groups of bits in a word

Encoding: Instruction Format

R-Format

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers, $t0=9, $s1=17, $s2=18

Opcodes on page A-50 (Figure 7.10.2)
Encodings – Section A10 (7.10)
MIPS Encoding: R-Type

```
0000000000110000100010000000000000000000
```

Encoding = 0x00622020

SPIM Example
Module Outline

Review ISA and understand instruction encodings
- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

Memory Organization

- 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.
Memory Organization

- Bytes are nice, but most data items use larger "words"
- MIPS provides \texttt{lw/lh/lb} and \texttt{sw/sh/sb} instructions
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

Registers hold 32 bits of data

- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
  - i.e., what are the least 2 significant bits of a word address?

Endianness [defined by Danny Cohen 1981]

- Byte ordering — How is a multiple byte data word stored in memory
- Endianness (from Gulliver’s Travels)
  - Big Endian
    - Most significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Sun Sparc, PowerPC
  - Little Endian
    - Least significant byte of a multi-byte word is stored at the lowest memory address
    - e.g. Intel x86
- Some embedded & DSP processors would support both for interoperability
Example of Endian

- Store 0x87654321 at address 0x0000, byte-addressable

<table>
<thead>
<tr>
<th>Lower Memory Address</th>
<th>Lower Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x0001</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x0002</td>
</tr>
<tr>
<td>0x0003</td>
<td>0x0003</td>
</tr>
</tbody>
</table>

Big Endian: 0x21 0x43 0x65 0x87

Little Endian: 0x87 0x65 0x43 0x21

Data Directives

- For placement of data in memory

.data
.word 0x1234
.byte 0x08
.asciiz "Hello World"
.ascii "Hello World"
.align 2
.space 64

Example: See page A-47 (7.10.1)
Instruction Set Architecture (ISA)

Module Outline

Review ISA and understand instruction encodings

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls – ECE 2035
Memory Instructions

- Load & store instructions: Orthogonal ISA
- Example:

  C code:  
  ```c
  long A[100];
  ```

  MIPS code:  
  ```mips
  lw $t0, 32($s3)  #load word
  add $t0, $s2, $t0
  sw $t0, 36($s3)
  ```

- Remember arithmetic operands are registers, not memory!

MIPS Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Names</th>
<th>Usage by Software Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$zero</td>
<td>Hardwired to zero</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>Reserved by assembler</td>
</tr>
<tr>
<td>$2 - $3</td>
<td>$v0 - $v1</td>
<td>Function return result registers</td>
</tr>
<tr>
<td>$4 - $7</td>
<td>$s0 - $s3</td>
<td>Function passing argument value registers</td>
</tr>
<tr>
<td>$8 - $15</td>
<td>$t0 - $t7</td>
<td>Temporary registers, caller saved</td>
</tr>
<tr>
<td>$16 - $23</td>
<td>$a0 - $a7</td>
<td>Saved registers, callee saved</td>
</tr>
<tr>
<td>$24 - $25</td>
<td>$t8 - $t9</td>
<td>Temporary registers, caller saved</td>
</tr>
<tr>
<td>$26 - $27</td>
<td>$halt - $at1</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>Global pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>Return address (pushed by call instruction)</td>
</tr>
<tr>
<td>$hi</td>
<td>$hi</td>
<td>High result register (remainder/div, high word/mult)</td>
</tr>
<tr>
<td>$lo</td>
<td>$lo</td>
<td>Low result register (quotient/div, low word/mult)</td>
</tr>
</tbody>
</table>

ISA defines what registers can be used for address calculations
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
- **Design Principle 3**: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
</table>

**Orthogonal ISA**

---

**MIPS Encoding: I-Type**

```
   rs  rt
   0   0

lw  $5,  3000($2)
```

Encoding = 0x8C450BB8
MIPS Encoding: I-Type

```
sw $5, 3000($2)
```

Encoding = 0xAC450BB8

Instruction Set Architecture (ISA)
• Small constants are used quite frequently (50% of operands)
  e.g., \( A = A + 5; \)
  \( B = B + 1; \)
  \( C = C - 18; \)

• Solutions?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
  - Use immediate values

• MIPS Instructions:
  
  \[
  \begin{align*}
  &\text{addi} \quad $29, \quad $29, \quad 4 \\
  &\text{slti} \quad $8, \quad $18, \quad 10 \\
  &\text{andi} \quad $29, \quad $29, \quad 6 \\
  &\text{ori} \quad $29, \quad $29, \quad 4
  \end{align*}
  \]

Immediate Operands

• No subtract immediate instruction
  - Just use a negative constant
    \[
    \text{addi} \quad $s2, \quad $s1, \quad -1
    \]

• Hardwired values useful for common operations
  - E.g., move between registers
    \[
    \text{add} \quad $t2, \quad $s1, \quad $zero
    \]

• Design Principle 4: Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction
How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

```
0000000000000000
1010101010101010
```

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```

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
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

Encoding: Constants & Immediates

- Use the I-format

- Compromise:
  - Use instruction sequences to construct larger constants
  - Avoid another adding another format ➔ impact on the hardware?

Example
Module Outline

Review ISA and understand instruction encodings

- Arithmetic and Logical Instructions
- Review memory organization
- Memory (data movement) instructions
- Control flow instructions
- Procedure/Function calls
- Program assembly, linking, & encoding

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
  bne $t0, $t1, Label
  beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

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

---

(39)

---

(40)
MIPS unconditional branch instructions:

\[ j \text{ label} \]

Example:

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

Can you build a simple for loop?

C code:

\[
\text{while (save[i] == k) i += 1;} \\
\]

\[ i \text{ in } $s3, \text{ k in } $s5, \text{ address of save in } $s6 \]

Compiled MIPS code:

\[
\text{Loop: sll } $t1, $s3, 2 \# \text{multiply by 4} \\
\text{add } $t1, $t1, $s6 \\
\text{lw } $t0, 0($t1) \\
\text{bne } $t0, $s5, \text{Exit} \\
\text{addi } $s3, $s3, 1 \\
\text{j Loop} \\
\text{Exit: ...}
\]
Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  \[
  \begin{align*}
  \text{if } & s1 < s2 \text{ then} \\
  & t0 = 1 \\
  \text{slt } & t0, s1, s2 \quad \text{else} \\
  & t0 = 0
  \end{align*}
  \]
- Can use this instruction to build "blt $s1, $s2, Label"
  - can now build general control structures
- For ease of assembly programmers, the assembler allows "blt" as a "pseudo-instruction"
  - assembler substitutes them with valid MIPS instructions
  - there are policy of use conventions for registers

```
blt $4, $5, loop
slt $1, $4, $5
bne $1, $0, loop
```

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
    - $t0 = 1
  - sltu $t0, $s0, $s1 # unsigned
    - $t0 = 0

**Encoding: Branches & Jumps**

- **Instructions:**
  
  \[
  \begin{align*}
  \text{bne } & \text{t4, } \text{t5, Label} & \text{Next instruction is at Label if } \text{t4} \neq \text{t5} \\
  \text{beq } & \text{t4, } \text{t5, Label} & \text{Next instruction is at Label if } \text{t4} = \text{t5} \\
  \text{j } & \text{Label} & \text{Next instruction is at Label}
  \end{align*}
  \]

- **Formats:**
  
  - Opcodes on page A-50 (Figure 7.10.2)
  - Encodings – Section A10 (7.10)

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
</table>

  |   | op |    |    | 26 bit address |

  - Use Instruction Address Register (PC = program counter)
  - Most branches are local (principle of locality)

- **Jump instructions just use high order bits of PC**
  - address boundaries of 256 MB

---

**BEQ/BNE uses I-Type**

- **BEQ/BNE** uses I-Type

  - **beq $0, $9, 40**

  - Offset Encoded by 40/4 = 10

  - **Encoding = 0x1009000A**

---

(46)
**MIPS Encoding: J-Type**

- **Jal** will jump and push return address in $ra ($31)

\[
jal \ 0x00400030
\]

Target Address: 0000 0000 0100 0000 0000 0000 0011 0000

Encoding = 0x0C10000C

**SPIM Example**

- **JR (Jump Register)**
  - Unconditional jump

\[
jr \ 0x00000002
\]

Instruction=4 bytes

(47)
**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>0</th>
<th>19</th>
<th>9</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Exit:** ...

---

**Branching Far Away**

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

- Example
    ```
    beq $s0,$s1, L1
    ↓
    bne $s0,$s1, L2
    j L1
    L2: ...
    ```
Addressing Modes

Operands are constant

1. Immediate addressing

Operand in register

2. Register addressing

Base addressing

3. Addressing

Memory

Lb $t0, 48($s0)

Immediate and spilled registers, such as those saved on procedure calls. MIPS register $zero always equals 0. Register $at is a jump register. It is always used for the assembler to handle procedure calls.

Conditional jump and link register

Jump and link register

Immediate

Compare less than constant

Equal test; PC-relative branch

Jump to target address

For switch, procedure return

Jump to target address

Equal test; PC-relative branch

Jump to target address

Conditional jump

Unconditional jump

To Summarize

MIPS assembly language

Instructions Example Meaning Comments

End add $t0, $t1, 200 200 = $t0 + $t1 200 = $t0 + $t1

Arithmetic add $t0, $t1, $t2 200 = $t0 + $t1 + $t2 200 = $t0 + $t1 + $t2

Data Transfer lw $t0, 100($s1) $s1 = $t0 + 100 $s1 = $t0 + 100

sw $t0, 100($s1) $s1 = $t0 + 100 $s1 = $t0 + 100

branch on equal $t0 $t0 = $t0, $t2, 200 $t0 = $t0, $t2, 200 $t0 = $t0, $t2, 200

branch on not equal $t0 $t0 = $t0, $t2, 200 $t0 = $t0, $t2, 200 $t0 = $t0, $t2, 200

Conditional branch

Miscellaneous

Print register

Jump to target address

Miscellaneous

Miscellaneous

Miscellaneous
Summary To Date: MIPS ISA

• Simple instructions all 32 bits wide
• Very structured
• Only three instruction formats

<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>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

• Rely on compiler to achieve performance — what are the compiler's goals?
• Help compiler where we can

Full Example
Opcodes on page A-50 (Figure 7.10.2)
Encodings — Section A10 (7.10)

(53)

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

(54)
Instruction Set Architecture (ISA)

Instruction Set Architectures (ISA)

- Instruction set architectures are characterized by several features

1. Operations
   - Types, precision, size
2. Organization of internal storage
   - Stack machine
   - Accumulator
   - General Purpose Registers (GPR)

3. Memory addressing
   - Operand location and addressing
Instruction Set Architectures

4. Memory abstractions
   - Virtual address spaces (more later)
   - Memory mapped I/O (later)

5. Control flow
   - Condition codes
   - Types of control transfers – conditional vs. unconditional

- ISA design is the result of many tradeoffs
  - Decisions determine hardware implementation
  - Impact on time, space, and energy
- Check out ISAs for PowerPC, ARM, x86, SPARC, etc.

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    - instructions from 1 to 54 bytes long!

- We’ll look at 80x86
1978: The Intel 8086 is announced (16 bit architecture)
1980: The 8087 floating point coprocessor is added
1982: The 80286 increases address space to 24 bits, +instructions
1985: The 80386 extends to 32 bits, new addressing modes
1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
(mostly designed for higher performance)
1997: MMX (SIMD-INT) is added (PPMT and P-II)
1999: SSE (single prec. SIMD-FP and cacheability instructions) is
added in P-III
2001: SSE2 (double prec. SIMD-FP) is added in P4
2004: Nocona introduced (compatible with AMD64 or once called x86-64)

“This history illustrates the impact of the “golden handcuffs” of compatibility
“adding new features as someone might add clothing to a packed bag”
“an architecture that is difficult to explain and impossible to love”

---

80x86

IA-32 Overview

- Complexity:
  - 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
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too
difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”
IA-32 Registers & Data Addressing

- Registers in the 32-bit subset that originated with 80386

<table>
<thead>
<tr>
<th>Name</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPRI 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPRI 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPRI 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPRI 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPRI 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPRI 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPRI 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPRI 7</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} + displacement
  - Address = R_{base} + 2^{scale} \times R_{index} (scale = 0, 1, 2, or 3)
  - Address = R_{base} + 2^{scale} \times R_{index} + displacement
IA-32 Register Restrictions

- Registers are not "general purpose" – note the restrictions below

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
<th>Register restrictions</th>
<th>RIP offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1600 (or 1280)</td>
</tr>
<tr>
<td>RCX</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1601 (or 1281)</td>
</tr>
<tr>
<td>RDX</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1602 (or 1282)</td>
</tr>
<tr>
<td>RBX</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1603 (or 1283)</td>
</tr>
<tr>
<td>RBP</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1604 (or 1284)</td>
</tr>
<tr>
<td>RSI</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1605 (or 1285)</td>
</tr>
<tr>
<td>RDI</td>
<td>Extra-wide 64-bit register</td>
<td>16 (or 128-bit)</td>
<td>1606 (or 1286)</td>
</tr>
</tbody>
</table>

 IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags )
  - String instructions, including string move and string compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>LDM 41-43</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>MOV</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>ADD</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>SUB</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>XOR</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>CMP</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JNE</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JLE</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JG</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JLE</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JG</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JLE</td>
<td>EIP+disp</td>
</tr>
<tr>
<td>JG</td>
<td>EIP+disp</td>
</tr>
</tbody>
</table>

 FIGURE 2.42 Some typical 32-bit instructions and their functions. A list of the 512-byte operation appears in Figure 2.46. The CLL saves the EIP of the next instruction on the stack. (EIP is the base PC.)
IA-32 instruction Formats

- Typical formats: (notice the different lengths)
  - CALL
    - JE (EBX + displacement)
    - CALL
    - MOV EBX, [ECX + 4]
  - PUSH ES
  - ADD EAX, [EBX]
  - TEST EDX, AH

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

- Instruction set design
  - Tradeoffs between compiler complexity and hardware complexity
  - Orthogonal (RISC) ISAs vs. complex ISAs (more on this later in the class)

- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

- Instruction set architecture
  - a very important abstraction indeed!

Summary
• What is i) an orthogonal instruction set, ii) load/store architecture, and iii) instruction set architecture?
• Translate small high level language (e.g., C, Matlab) code blocks into MIPS assembly
  - Allocate variables to registers
  - Layout data in memory
  - Sequence data into/out of registers as necessary
  - Write assembly instructions/program
• Write and execute the proceeding for
  - A few simple if-then-else cases (say from C)
  - for loops and while loops

Study Guide (cont.)
• Utilize data directives to layout data in memory
  - Check anticipated layout in SPIM
  - Layout a 2D matrix and a 3D matrix
  - Layout a linked list
• Manually assemble instructions and check with SPIM
• Given a program, encode branch and jump instructions
  - Use SPIM to verify your answers – remember SPIM branches are relative to the PC not PC+4
• Use SPIM to assemble some small programs
  - Manually disassemble the code
Study Guide (cont.)

- Name two advantages of a CISC ISA over a RISC ISA
- Name two disadvantages of a CISC ISA over a RISC ISA
- What is the difference between big endian and little endian storage?
- Store a sequence of words in memory using the .byte directives. What are the values at word boundaries assuming little endian or big endian format?

Glossary

- Big endian
- Binary compatibility
- Byte aligned memory access
- CISC
- Data directives
- Destination operand
- General purpose registers
- I-format
- Immediate operand
- Instruction encoding
- Instruction format
- Instruction set architecture
- J-format
- Little Endian
- Machine code (or language)
- Memory map
Glossary (cont.)

- Native instructions
- Orthogonal ISA
- PC-relative addressing
- Pseudo instructions
- R-format
- RISC
- Sign extension
- Source operand
- Unsigned vs. signed instructions
- Word aligned memory access

Intel IA-32 Register View

- Many features are a byproduct of backward compatibility issues
- Distinctive relative to the MIPS ISA
- Representative of Complex Instruction Set Computer (CISC)

Courtesy Intel IA-32 Software Developers Manual
ARM ISA View

System level view

Privileged modes

Exception modes

<table>
<thead>
<tr>
<th>User mode</th>
<th>System mode</th>
<th>Hyp. mode</th>
<th>Supervisor mode</th>
<th>Monitor mode</th>
<th>Abort mode</th>
<th>Undefined mode</th>
<th>IRQ mode</th>
<th>FIQ mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
<td>LR</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
</tr>
</tbody>
</table>

† Hyp mode and the associated branch registers are implemented only as part of the Virtualization Extensions

‡ Monitor mode and the associated branch registers are implemented only as part of the Security Extensions

Courtesy ARM University Program Presentation (73)