MIPS ISA-I: The Instruction Set Architecture

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
Reading

- Chapter 2
  - 2.1, Figure 2.1, 2.2 – 2.7
  - 2.9, Figure 2.15
  - 2.10, 2.16, 2.17

- Appendix A9, A10

- Goals:
  - Understand how programs are encoded
  - Impact of ISA on program encodings
  - Why are ISAs different?

---

Economics of an ISA

- Thermal Design Power 130W
  - 3.6 GHz

- Thermal Design Power 4W
  - 1.6 GHz

*Software/binary portability*
Below Your Program

- **Application software**
  - Written in high-level language

- **System software**
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources

- **Hardware**
  - Processor, memory, I/O controllers

---

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
Instructions

- Language of the Machine
- More primitive than higher level languages  
  e.g., no sophisticated control flow
- Very restrictive  
  e.g., MIPS Arithmetic 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)

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

Data flow for transfers

Data flow for computation

byte addressed memory
0x00000000
0x00000001
0x00000002
0x00000003

Address Space

Arithmetic Logic Unit (ALU)

0xFFFFFFFF
Intel IA-32 Register View

- Many features are byproduct of backward compatibility issues
- Distinctive relative to the MIPS ISA
- More later.....

ARM ISA View

More later...
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>$31</td>
<td>$hi</td>
<td>High result register (remainder/div, high word/mult)</td>
</tr>
<tr>
<td>$31</td>
<td>$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
  - ......

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
  ......
  ```

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

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

and $t0, $t1, $t2

<table>
<thead>
<tr>
<th>$t2</th>
<th>0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>0000 0000 0000 0000 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

\[
\begin{array}{c}
\text{or}$t0, $t1, $t2 \\
\hline
\text{$t2$} & 0000 & 0000 & 0000 & 0000 & 0000 & 1101 & 1100 & 0000 \\
\text{$t1$} & 0000 & 0000 & 0000 & 0000 & 0111 & 1100 & 0000 & 0000 \\
\text{$t0$} & 0000 & 0000 & 0000 & 0000 & 0111 & 1101 & 1100 & 0000 \\
\end{array}
\]

(17)

NOT Operations

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

nor $t0, $t1, $zero

\[
\begin{array}{c}
nor$t0, $t1, $zero \\
\hline
\text{$t1$} & 0000 & 0000 & 0000 & 0000 & 0111 & 1100 & 0000 & 0000 \\
\text{$t0$} & 1111 & 1111 & 1111 & 1111 & 1100 & 0011 & 1111 & 1111 \\
\end{array}
\]

(18)
- 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
Encodings – Section A10

MIPS Encoding: R-Type

Encoding = 0x00622020
MIPS Encoding: R-Type

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.

0 8 bits of data
1 8 bits of data
2 8 bits of data
3 8 bits of data
4 8 bits of data
5 8 bits of data
6 8 bits of data
...

(23)

Memory Organization

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

0 32 bits of data
4 32 bits of data
8 32 bits of data
12 32 bits of data

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?

(24)
Data Directives

- For placement of data in memory

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

Example: See page A-47

---

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

BIG ENDIAN

LITTLE ENDIAN

Instruction Set Architecture (ISA)

Register File

Data flow for transfers

byte addressed memory

Address Space

words

Arithmetic Logic Unit (ALU)
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 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!

  ![Diagram](image)
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 usage optimization is important!
- **Design Principle 2**: Smaller is faster
  - c.f. main memory: millions of locations
- Rationale for the Memory Hierarchy

### 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 - $7</td>
<td>$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>$hi</td>
<td>$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>
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>

**MIPS Encoding: I-Type**

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

Encoding = 0x8C450BB8
MIPS Encoding: I-Type

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

sw $5, 3000($2)

Encoding = 0xAC450BB8

SPIM Example

Constants

- 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:
  - addi $29, $29, 4
  - slti $8, $18, 10
  - andi $29, $29, 6
  - ori $29, $29, 4
Immediate Operands

- No subtract immediate instruction
  - Just use a negative constant
    \[ \text{addi } \$s2, \$s1, -1 \]
- Hardwired values useful for common operations
  - E.g., move between registers
    \[ \text{add } \$t2, \$s1, \$zero \]
- **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

\[
\text{lui } \$t0, 1010101010101010{}^{\text{filled with zeros}} \\
1010101010101010 \ 0000000000000000
\]

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

\[
\text{ori } \$t0, \$t0, 1010101010101010
\]

Now consider \( \text{la } \$t0, L1 \) (a **pseudo instruction**)

\[
\begin{array}{c}
\text{lui } \$
\end{array}
\]

---

(37)

(38)
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
• 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: ....

• MIPS unconditional branch instructions:

  j label

• Example: if (i!=j)

  beq $s4, $s5, Lab1
  add $s3, $s4, $s5
  j Lab2
  sub $s3, $s4, $s5
  Lab1: ...
  Lab2: ...

• Can you build a simple for loop?
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: ...
  ```

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  ```mips
  if $s1 < $s2 then
    $t0 = 1
  slt $t0, $s1, $s2
  else
    $t0 = 0
  ```

- 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

- Example:
  ```mips
  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 0000 0001`
  - `slt $t0, $s0, $s1` # signed
    - `-1 < +1 ⇒ $t0 = 1`
  - `sltu $t0, $s0, $s1` # unsigned
    - `+4,294,967,295 > +1 ⇒ $t0 = 0`

Instructions:

- `bne $t4,$t5,Label` Next instruction is at Label if `$t4 ≠ $t5`
- `beq $t4,$t5,Label` Next instruction is at Label if `$t4 = $t5`
- `j Label` Next instruction is at Label

Formats:

<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>op</td>
<td>26 bit address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

Opcodes on page A-50
Encodings – Section A10
BEQ/BNE uses I-Type

```
opcode   rs   rt   Signed Offset Value
```

(encoded in words, e.g. 4-bytes)

- `beq $0, $9, 40`
- Offset Encoded by 40/4 = 10

```
0011000000000000010010000000000010110
```

Encoding = 0x1009000a

---

MIPS Encoding: J-Type

```
opcode   Target Address
```

- `jal` will jump and push return address in $ra ($31)

```
jal  0x00400030
```

```
0000 0000 0100 0000 0000 0000 0000 0000
```

Target Address Instruction=4 bytes

```
0011000000000000010010000000000010110
```

Encoding = 0xC10000C

---

SPIM Example
• JR (Jump Register)
  - Unconditional jump

jr $2

opcode   rs   0   0   0   funct 1
000000010000000000000000000010100

Target Addressing Example

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

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rs2</th>
<th>Rs3</th>
<th>Opcode2</th>
<th>Rs4</th>
<th>Rs5</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>00100100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>00100111</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>00100100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>00100100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>00100100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>00100100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
<td>00100100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>000000000000000000000000000010100</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
  
  ```
  beq $s0,$s1, L1
  ↓
  bne $s0,$s1, L2
  j L1
  
  L2: ...
  ```

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
What does this imply about targets?

Addressing Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>immediate</td>
<td>Immediate value</td>
<td>Used for constant operands</td>
</tr>
<tr>
<td>Register</td>
<td>reg</td>
<td>Value from register</td>
<td>Always a 32-bit value</td>
</tr>
<tr>
<td>Address</td>
<td>address</td>
<td>Memory location</td>
<td>Can be a 32-bit or 64-bit value</td>
</tr>
<tr>
<td>PC-relative</td>
<td>PC offset</td>
<td>Value from program counter</td>
<td>Can be a 32-bit or 64-bit value</td>
</tr>
</tbody>
</table>

To Summarize

MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Add immediate, data in registers</td>
<td></td>
</tr>
<tr>
<td>Data transfer</td>
<td>lb $t0, 48($s0)</td>
<td>Memory[48] = $t0</td>
<td>Load byte, memory to register</td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>bne $4, $5, Label</td>
<td>if ($s2 &lt; $s3) go to Label</td>
<td>Branch on not equal, PC-relative jump</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>j Label</td>
<td>PC = Label</td>
<td>Jump to target address</td>
<td></td>
</tr>
<tr>
<td>Undefined</td>
<td>j Label</td>
<td>PC = Label</td>
<td>Jump to target address</td>
<td></td>
</tr>
</tbody>
</table>

Comments:

- Immediate: Used for constant operands.
- Register: Value from register, always a 32-bit value.
- Address: Memory location, can be a 32-bit or 64-bit value.
- PC-relative: Value from program counter, can be a 32-bit or 64-bit value.

Jump registers:

- The jump register, often $ra, is used for jump instructions.
- It is also used for return from procedure calls.

Branch on less than:

- bgt, bge, bles, ble:
  - Compare values, branch if greater or equal.
- slt, sltu:
  - Compare less than, branch if less.

Load upper immediate:

- lui $s1, 100:
  - Load upper 16 bits.

Store byte:

- sb $s1, 100($s2):
  - Store Byte to memory.

Load byte:

- lb $s1, 100($s2):
  - Load Byte from memory.

Store word:

- sw $s1, 100($s2):
  - Store Word to memory.

Load word:

- lw $s1, 100($s2):
  - Load Word from memory.

Add immediate:

- add $s1, $s2, $s3:
  - Add immediate, data in registers.

Add:

- add $s1, $s2, $s3:
  - Add, data in registers.

Sub:

- sub $s1, $s2, $s3:
  - Subtract, data in registers.

Subtract:

- sub $s1, $s2, $s3:
  - Subtract, data in registers.

Load immediate:

- lui $s1, 100:
  - Load immediate in upper 16 bits.
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>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td>funct</td>
</tr>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td>rt</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
Encodings – Section A10

(57)

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
Stored Program Concept

- Fetch & Execute Cycle \(\rightarrow\) sequential flow of control
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue
  - Program Counter \(\leftarrow\) Program Counter + 4 (byte addressing!)
  - Von Neumann execution model

Example:

Instruction Set Architecture (ISA)

Who sees what?
**Summary**

- 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!

**Study Guide**

- 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

• Use SPIM to assemble some small programs
  ◆ Manually disassemble the code

Study Guide (cont.)

• Synthesize complex inequality tests with the slt instruction
  ◆ e.g., bgt, ble, bge

• Some simple learning exercises – write SPIM programs for
  ◆ Reversing the order of bytes in a register
  ◆ Reversing the order of bytes in a memory location
  ◆ Compute the exclusive-OR of the contents of two registers
  ◆ Create a linked list to store an array of four numbers, one number per element
  ◆ Traverse the preceding linked list to compute the sum of the numbers
  ◆ Fetch a word starting an non-word boundary
Study Guide (cont.)

- Take a block of simple MIPS code and
  - Translate it to equivalent ARM code
  - Translate it to equivalent x86 code
- Name two advantages of a CISC ISA over a RISC ISA
- Name two disadvantages of a CISC ISA over a RISC ISA

Glossary

- Basic block
- Big endian
- Binary compatibility
- Byte aligned memory access
- CISC
- Data directives
- Destination operand
- Frame pointer
- General purpose registers
- Global pointer
- 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
- Stack pointer
- System software vs. application software
- Unsigned vs. signed instructions
- Word aligned memory access
- Von Neumann execution model