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
    o Handling input/output
    o Managing memory and storage
    o 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
- Data flow for transfers
- Data flow for computation
- Arithmetic Logic Unit (ALU)
- Address Space
- Byte addressed memory

---
• Many features are a byproduct of backward compatibility issues
• Distinctive relative to the MIPS ISA
• 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

$t2 0000 0000 0000 0000 0000 1101 1100 0000
$t1 0000 0000 0000 0000 0011 1100 0000 0000
$t0 0000 0000 0000 0000 0000 1100 0000 0000
OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```markdown
or $t0, $t1, $t2
```

```
| $t2     | 0000 0000 0000 0000 0000 1101 1100 0000 |
| $t1     | 0000 0000 0000 0000 0011 1100 0000 0000 |
| $t0     | 0000 0000 0000 0000 0011 1101 1100 0000 |
```

(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 NOR b == NOT ( a OR b )

```markdown
nor $t0, $t1, $zero
```

```
| $t1      | 0000 0000 0000 0000 0011 1100 0000 0000 |
| $t0      | 1111 1111 1111 1111 1100 0011 1111 1111 |
```

(18)
Instructions, like registers and words of data, are also 32 bits long.

Example:  add $t0, $s1, $s2

Example:  add $t0, $s1, $s2

Registers have numbers, $t0 = 9, $s1 = 17, $s2 = 18

MIPS Encoding: R-Type

Encoding = 0x00622020
MIPS Encoding: R-Type

```
31 26 25 24 21 20 16 15 14 11 10 6 5 0
<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

sll $3, $5, 7

```
00000000000000000000000000000000
```

Encoding = 0x000519C0

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.

---

<table>
<thead>
<tr>
<th></th>
<th>8 bits of data</th>
<th>8 bits of data</th>
<th>8 bits of data</th>
<th>8 bits of data</th>
<th>8 bits of data</th>
<th>8 bits of data</th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

(23)

---

Memory Organization

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

---

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

---

(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 B-47

(25)

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

(26)
Example of Endian

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

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

<table>
<thead>
<tr>
<th>BIG ENDIAN</th>
<th>LITTLE ENDIAN</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>

Instruction Set Architecture (ISA)

- Register File
- Data flow for transfers
- byte addressed memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x0003</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x1F</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

- Address Space
- 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!
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)`

```
35 18 8 32
```

```
op rs rt 16 bit number
```

**MIPS Encoding: I-Type**

```
31 26 25 21 20 16 15
0
```

```
opcode rs rt Immediate Value
```

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

```
10001100010010100001011110110000
```

```
Encoding = 0x8C450BB8
```
MIPS Encoding: I-Type

\[
\begin{array}{ccccccc}
31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 \\
\text{opcode} & \text{rs} & \text{rt} & \text{Immediate Value}
\end{array}
\]

\[\text{sw} \ 5, \ 3000($2)\]

\[
\begin{array}{ccccccc}
31 & 26 & 25 & 21 & 20 & 16 & 15 & 0 \\
\text{rs} & \text{rt} & \text{Immediate}
\end{array}
\]

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

(compiler)

ISA

HW support
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
    \[ \text{add } t2, s1, \text{zero} \]

- **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

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

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

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

Now consider \( \text{la } t0, L1 \) (a pseudo instruction)
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: \[0000\ldots0000\]
  - \(-1\): \[1111\ldots1111\]
  - Most-negative: \[1000\ldots0000\]
  - Most-positive: \[0111\ldots1111\]

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 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
  h=i+j;
  add $s3, $s4, $s5
  else
  j Lab2
  h=i-j;
  Lab1: sub $s3, $s4, $s5
  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

  ```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
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111$
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001$
  - `slt $t0, $s0, $s1` # signed
    - $-1 < +1 \Rightarrow t0 = 1$
  - `sltu $t0, $s0, $s1` # unsigned
    - $+4,294,967,295 > +1 \Rightarrow t0 = 0$

Instructions:

- `bne $t4,$t5,Label` Next instruction is at Label if $t4 \neq t5$
- `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 B-50
Encodings – Section B10
BEQ/BNE uses I-Type

31 26 25 21 20 16 15 10
<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Signed Offset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(encoded in words, e.g. 4-bytes)</td>
</tr>
</tbody>
</table>

```
beq $0, $9, 40
```

Offset Encoded by 40/4 = 10

31 26 25 21 20 16 15 10
<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immediate Value</th>
</tr>
</thead>
</table>

```
0000 0000 0000 0000 0000 0000 0000 0101 1010 0000 0000 0000 0000 0000 1011 0
```

Encoding = 0x1009000A

(49)

MIPS Encoding: J-Type

31 26 25 16 15 10
<table>
<thead>
<tr>
<th>opcode</th>
<th>Target Address</th>
</tr>
</thead>
</table>

```
jal 0x00400030
```

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

0000 0000 0100 0000 0000 0000 0011 0006

Target Address Instruction=4 bytes

31 26 25 16 15 10
<table>
<thead>
<tr>
<th>opcode</th>
<th>Target Address</th>
</tr>
</thead>
</table>

```
jal 0x00400030
```

0000 0000 0100 0000 0000 0000 0000 0011 0006

Target Address Instruction=4 bytes

31 26 25 16 15 10
<table>
<thead>
<tr>
<th>opcode</th>
<th>Target Address</th>
</tr>
</thead>
</table>

```
jal 0x00400030
```

0000 0000 0100 0000 0000 0000 0011 0006

Encoding = 0x0C10000C

(50)
• JR (Jump Register)
  - Unconditional jump

jr $2 0000000000110000000000000000000000000000000000000000000000000000
  
  opcode rs 0 0 0 0 funct

Target Addressing Example

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

Loop:  
  sll $t1, $s3, 2 80000 0 0 19 9 4 0
  add $t1, $t1, $s6 80004 0 9 22 9 0 32
  lw $t0, 0($t1) 80008 35 9 8 0
  bne $t0, $s5, Exit 80012 5 8 21 2 0
  addi $s3, $s3, 1 80016 8 19 19 1
  j Loop 80020 2 20000

Exit: ... 80024
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
Addressing Modes

Operand is constant

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>addi</td>
<td>$s1 = $s2 + 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>$s1 = Memory[+$s2]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>load byte</td>
<td>lb</td>
<td>$s1 = Memory[+$s2]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>$s1 = 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
</tbody>
</table>

Operand is in register

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>addi</td>
<td>$s1 = $s2 + 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>$s1 = Memory[+$s2]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>load byte</td>
<td>lb</td>
<td>$s1 = Memory[+$s2]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>$s1 = 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
</tbody>
</table>

Base addressing

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>addi</td>
<td>$s1 = $s2 + 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>$s1 = Memory[+$s2]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>load byte</td>
<td>lb</td>
<td>$s1 = Memory[+$s2]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>$s1 = 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
</tbody>
</table>

PC-relative addressing

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>addi</td>
<td>$s1 = $s2 + 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>$s1 = Memory[+$s2]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>load byte</td>
<td>lb</td>
<td>$s1 = Memory[+$s2]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>$s1 = 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
</tbody>
</table>

Predefined addressing

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>addi</td>
<td>$s1 = $s2 + 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>$s1 = Memory[+$s2]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>load byte</td>
<td>lb</td>
<td>$s1 = Memory[+$s2]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>$s1 = 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
</tbody>
</table>

To Summarize

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>jal</td>
<td>2500</td>
<td>Jump to target address</td>
</tr>
<tr>
<td>jr</td>
<td>$ra</td>
<td>Jump to target address</td>
</tr>
<tr>
<td>slti</td>
<td>$s1, $s2, 100</td>
<td>Test less than</td>
</tr>
<tr>
<td>slt</td>
<td>$s1, $s2, $s3</td>
<td>Test less than; for beq, bne</td>
</tr>
<tr>
<td>bne</td>
<td>$s1, $s2, 25</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td>beq</td>
<td>$s1, $s2, 25</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td>lui</td>
<td>$s1, 100</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>sb</td>
<td>$s1, 100($s2)</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td>lb</td>
<td>$s1, 100($s2)</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td>sw</td>
<td>$s1, 100($s2)</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td>lw</td>
<td>$s1, 100($s2)</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td>addi</td>
<td>$s1, $s2, 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>add</td>
<td>$s1, $s2, $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td>sub</td>
<td>$s1, $s2, $s3</td>
<td>Three operands; data in registers</td>
</tr>
</tbody>
</table>

Comments:
- Operands may be in registers, immediates, or constants.
- Memory holds data structures, such as arrays.
- MIPS assembly language
- Comments indicate the purpose of each instruction.
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>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>16 bit address</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
<td></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 B-50
Encodings – Section B10

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
• Fetch & Execute Cycle → 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 ← 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
• 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

• Synthesize complex inequality tests with the \textit{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