Lecture 4: ISA Tradeoffs (Continued) and MIPS ISA
Agenda for Today

- Finish off ISA tradeoffs
- A quick tutorial on MIPS ISA

Upcoming schedule:

- **Lab 1.5 & 2** are out today
- Friday (1/23): **Lab 1** due
- Friday (1/23): Recitation
- Wednesday (1/28): **HW 1** due
Upcoming Readings

- Next week (Microarchitecture):
  - P&H, Chapter 4, Sections 4.1-4.4
  - P&P, revised Appendix C – LC3b datapath and microprogrammed operation
Last Lecture Recap

- Instruction processing style
  - 0, 1, 2, 3 address machines
- Elements of an ISA
  - Instructions, data types, memory organizations, registers, etc
- Addressing modes
- Complex (CISC) vs. simple (RISC) instructions
- Semantic gap
- ISA translation
ISA-level Tradeoffs: Instruction Length

- **Fixed length**: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions *(Why is this bad?)*
  - -- Harder-to-extend ISA (how to add new instructions?)

- **Variable length**: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding *(Why is this good?)*
    - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. *How?*
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently

- **Tradeoffs**
  - Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ISA extensibility and expressiveness vs. hardware complexity
  - Performance? Energy? Smaller code vs. ease of decode
**ISA-level Tradeoffs: Uniform Decode**

- **Uniform decode**: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, ...
  - Many “RISC” ISAs: Alpha, MIPS, SPARC
  - Easier decode, simpler hardware
  - Enables parallelism: generate target address before knowing the instruction is a branch
  -- Restricts instruction format (fewer instructions?) or wastes space

- **Non-uniform decode**
  - E.g., opcode can be the 1st-7th byte in x86
  - More compact and powerful instruction format
  -- More complex decode logic
x86 vs. Alpha Instruction Formats

- **x86:**

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

- **Alpha:**

  ![Alpha Instruction Format Diagram]
MIPS Instruction Format

- **R-type, 3 register operands**
  
<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>

- **I-type, 2 register operands and 16-bit immediate operand**
  
<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

- **J-type, 26-bit immediate operand**
  
<table>
<thead>
<tr>
<th>opcode</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</td>
</tr>
</tbody>
</table>

- **Simple Decoding**
  
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b’00)
  - format and fields easy to extract in hardware
**Figure 4-1: ARM instruction set formats**

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000 A S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>000001 U A S</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rn</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>00010 B 00</td>
<td>Rn</td>
<td>Rd</td>
<td>00001001</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>0001001001</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 PU0 WL</td>
<td>Rn</td>
<td>Rd</td>
<td>00001 SH 1</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>0000 PU1 WL</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
<td>1 SH 1 Offset</td>
<td></td>
</tr>
<tr>
<td>011 PU BW L</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>100 PU SW L</td>
<td>Rn</td>
<td></td>
<td>Register List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 L</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 PU NW L</td>
<td>Rn</td>
<td>CRd</td>
<td>CP#</td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>1110 CP Opc</td>
<td>CRn</td>
<td>CRd</td>
<td>CP#</td>
<td>CP 0 CRm</td>
<td></td>
</tr>
<tr>
<td>1110 CP Opc</td>
<td>CRn</td>
<td>R</td>
<td>CP#</td>
<td>CP 1 CRm</td>
<td></td>
</tr>
</tbody>
</table>

**Data Processing / PSR Transfer**

**Multiply**

**Multiply Long**

**Single Data Swap**

**Branch and Exchange**

**Halfword Data Transfer: register offset**

**Halfword Data Transfer: immediate offset**

**Single Data Transfer**

**Undefined**

**Block Data Transfer**

**Branch**

**Coprocessor Data Transfer**

**Coprocessor Data Operation**

**Coprocessor Register Transfer**

**Software Interrupt**

**Ignored by processor**
A Note on Length and Uniformity

- Uniform decode usually goes with fixed length

- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths
A Note on RISC vs. CISC

- Usually, ...

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes
ISA-level Tradeoffs: Number of Registers

- **Affects:**
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- **Large number of registers:**
  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  -- Larger instruction size
  -- Larger register file size
ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)

- More modes:
  + help better support programming constructs (arrays, pointer-based accesses)

- make it harder for the architect to design
- too many choices for the compiler?
  - Many ways to do the same thing complicates compiler design
x86 vs. Alpha Instruction Formats

- **x86:**

  ![x86 Instruction Format Diagram]

  - Instruction Prefixes: Up to four prefixes of 1 byte each (optional)
  - Opcode: 1-, 2-, or 3-byte opcode
  - ModR/M: 1 byte (if required)
  - SI B: 1 byte (if required)
  - Displacement: Address displacement of 1, 2, or 4 bytes or none
  - Immediate: Immediate data of 1, 2, or 4 bytes or none

- **Alpha:**

  ![Alpha Instruction Format Diagram]

  - Opcode: RA | RB | Function | RC
  - Number: Disp
  - PALcode Format
  - Branch Format
  - Memory Format
  - Operate Format
Table 2.2: 32-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX] + disp8  2</td>
<td>01</td>
<td>001</td>
<td>04  48  50  58  60  68  70  78</td>
</tr>
<tr>
<td>[ECX] + disp8  2</td>
<td>01</td>
<td>002</td>
<td>04  49  51  59  61  69  71  79</td>
</tr>
<tr>
<td>[EDX] + disp8  2</td>
<td>01</td>
<td>003</td>
<td>04  4A  53  5A  62  6A  72  7A</td>
</tr>
<tr>
<td>[EBX] + disp8  2</td>
<td>01</td>
<td>004</td>
<td>04  4B  54  5B  63  6B  73  7B</td>
</tr>
<tr>
<td>[ESP] + disp8  2</td>
<td>01</td>
<td>005</td>
<td>04  4C  55  5C  64  6C  74  7C</td>
</tr>
<tr>
<td>[EBP] + disp8  2</td>
<td>01</td>
<td>006</td>
<td>04  4D  56  5D  65  6D  75  7D</td>
</tr>
<tr>
<td>[ESI] + disp8</td>
<td>01</td>
<td>007</td>
<td>04  4E  57  5E  66  6E  76  7E</td>
</tr>
<tr>
<td>[EDI] + disp8</td>
<td>01</td>
<td>008</td>
<td>04  4F  58  5F  67  6F  77  7F</td>
</tr>
<tr>
<td>[EAX] + disp32</td>
<td>10</td>
<td>010</td>
<td>08  88  90  98  A0  A8  B0  B8</td>
</tr>
<tr>
<td>[ECX] + disp32</td>
<td>10</td>
<td>011</td>
<td>08  89  91  99  A1  A9  B1  B9</td>
</tr>
<tr>
<td>[EDX] + disp32</td>
<td>10</td>
<td>012</td>
<td>08  8A  92  9A  A2  AA  B2  BA</td>
</tr>
<tr>
<td>[EBX] + disp32</td>
<td>10</td>
<td>013</td>
<td>08  8B  93  9B  A3  AB  B3  BB</td>
</tr>
<tr>
<td>[ESP] + disp32</td>
<td>10</td>
<td>014</td>
<td>08  8C  94  9C  A4  AC  B4  BC</td>
</tr>
<tr>
<td>[EBP] + disp32</td>
<td>10</td>
<td>015</td>
<td>08  8D  95  9D  A5  AD  B5  BD</td>
</tr>
<tr>
<td>[ESI] + disp32</td>
<td>10</td>
<td>016</td>
<td>08  8E  96  9E  A6  AE  B6  BE</td>
</tr>
<tr>
<td>[EDI] + disp32</td>
<td>10</td>
<td>017</td>
<td>08  8F  97  9F  A7  AF  B7  BF</td>
</tr>
</tbody>
</table>

Register

1. The [-][-] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.
## Table 2-3. 32-Bit Addressing Forms with the SIB Byte

<table>
<thead>
<tr>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>000</td>
<td>0B 01 02 03 04 05 06 07</td>
</tr>
<tr>
<td>[ECX]</td>
<td>001</td>
<td>001</td>
<td>0B 09 0A 0B 0C 0D 0E 0F</td>
</tr>
<tr>
<td>[EDX]</td>
<td>010</td>
<td>010</td>
<td>18 19 1A 1B 1C 1D 1E 1F</td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>100</td>
<td>30 29 28 27 26 25 24 23</td>
</tr>
<tr>
<td>[EBP]</td>
<td>101</td>
<td>101</td>
<td>38 39 3A 3B 3C 3D 3E 3F</td>
</tr>
<tr>
<td>[ESI]</td>
<td>110</td>
<td>110</td>
<td>40 41 42 43 44 45 46 47</td>
</tr>
<tr>
<td>[EDI]</td>
<td>111</td>
<td>111</td>
<td>50 51 52 53 54 55 56 57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX*2]</td>
<td>00</td>
<td>000</td>
<td>4B 49 4A 4B 4C 4D 4E 4F</td>
</tr>
<tr>
<td>[ECX*2]</td>
<td>001</td>
<td>001</td>
<td>58 59 5A 5B 5C 5D 5E 5F</td>
</tr>
<tr>
<td>[EDX*2]</td>
<td>010</td>
<td>010</td>
<td>60 61 62 63 64 65 66 67</td>
</tr>
<tr>
<td>[EBX*2]</td>
<td>011</td>
<td>011</td>
<td>70 71 72 73 74 75 76 77</td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>100</td>
<td>80 81 82 83 84 85 86 87</td>
</tr>
<tr>
<td>[EBP*2]</td>
<td>101</td>
<td>101</td>
<td>90 91 92 93 94 95 96 97</td>
</tr>
<tr>
<td>[ESI*2]</td>
<td>110</td>
<td>110</td>
<td>0B 09 0A 0B 0C 0D 0E 0F</td>
</tr>
<tr>
<td>[EDI*2]</td>
<td>111</td>
<td>111</td>
<td>1B 1A 19 18 17 16 15 14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scaled Index</th>
<th>SS</th>
<th>Index</th>
<th>Value of SIB Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX*4]</td>
<td>00</td>
<td>000</td>
<td>B0 B1 B2 B3 B4 B5 B6 B7</td>
</tr>
<tr>
<td>[ECX*4]</td>
<td>001</td>
<td>001</td>
<td>B8 B9 BA BB BC BD BE BF</td>
</tr>
<tr>
<td>[EDX*4]</td>
<td>010</td>
<td>010</td>
<td>C0 C1 C2 C3 C4 C5 C6 C7</td>
</tr>
<tr>
<td>[EBX*4]</td>
<td>011</td>
<td>011</td>
<td>C8 C9 CA CB CC CD CE CF</td>
</tr>
<tr>
<td>none</td>
<td>100</td>
<td>100</td>
<td>D0 D1 D2 D3 D4 D5 D6 D7</td>
</tr>
<tr>
<td>[EBP*4]</td>
<td>101</td>
<td>101</td>
<td>E0 E1 E2 E3 E4 E5 E6 E7</td>
</tr>
<tr>
<td>[ESI*4]</td>
<td>110</td>
<td>110</td>
<td>F0 F1 F2 F3 F4 F5 F6 F7</td>
</tr>
<tr>
<td>[EDI*4]</td>
<td>111</td>
<td>111</td>
<td>FA FB FC FD FE FF</td>
</tr>
</tbody>
</table>

### NOTES:
1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

**MOD bits** | **Effective Address**
---|---
00 | [scaled index] + disp32
01 | [scaled index] + disp8 + [EBP]
10 | [scaled index] + disp32 + [EBP]
X86 SIB-D Addressing Mode

Figure 3-11. Offset (or Effective Address) Computation

```
native 0
```

x86 Manual Vol. 1, page 3-22  -- see course resources on website
Also, see Section 3.7.3 and 3.7.5
The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

- **Base** — A base address is used to specify an offset to the operand. Since the value in the base register can change, it can be used to access a series of variables and data structures.

- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
  
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.
• (Index * Scale) + Displacement — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

• Base + Index + Displacement — Using two registers together supports either a two-dimensional array. Displacement holds the address of the beginning of the array or one of several instances of an array of records (the displacement is an offset to a field within the record).

• Base + (Index * Scale) + Displacement — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.
Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- ...

...
Many ISA features designed to aid programmers
But, complicate the hardware designer’s job

Virtual memory
- vs. overlay programming
- Should the programmer be concerned about the size of code blocks fitting physical memory?

Addressing modes

Unaligned memory access
- Compiler/programmer needs to align data
MIPS: Aligned Access

- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the “infrequent” case

<table>
<thead>
<tr>
<th>MSB</th>
<th>byte-3</th>
<th>byte-2</th>
<th>byte-1</th>
<th>byte-0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte-7</td>
<td>byte-6</td>
<td>byte-5</td>
<td>byte-4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWL  rd 6(r0)</td>
<td>byte-6</td>
<td>byte-5</td>
<td>byte-4</td>
<td>D</td>
</tr>
<tr>
<td>LWR  rd 3(r0)</td>
<td>byte-6</td>
<td>byte-5</td>
<td>byte-4</td>
<td>byte-3</td>
</tr>
</tbody>
</table>

- LWL/LWR is slower
- Note LWL and LWR still fetch within word boundary
X86: Unaligned Access

- LD/ST instructions automatically align data that spans a “word” boundary.
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location).

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.
X86: Unaligned Access

Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory
What About ARM?

- https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf
  - Section A2.8
Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment

- Cons of having no restrictions on alignment

- Filling in the above: an exercise for you...
18-447 MIPS ISA

James C. Hoe
Dept of ECE, CMU
MIPS R2000 Program Visible State

**Program Counter**
32-bit memory address of the current instruction

|-------|------|------|------|------|--------|

**General Purpose Register File**
32 32-bit words named r0...r31

**Note**
\[ r0=0 \]

**r1**

**r2**

**Memory**
2^{32} by 8-bit locations (4 Giga Bytes)
32-bit address

(there is some magic going on)
Data Format

- Most things are 32 bits
  - instruction and data addresses
  - signed and unsigned integers
  - just bits
- Also 16-bit word and 8-bit word (aka byte)
- Floating-point numbers
  - IEEE standard 754
  - float: 8-bit exponent, 23-bit significand
  - double: 11-bit exponent, 52-bit significand
Big Endian vs. Little Endian

(Part I, Chapter 4, Gulliver’s Travels)

- 32-bit signed or unsigned integer comprises 4 bytes

- On a byte-addressable machine . . . . . . .

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 0</td>
<td>byte 1</td>
</tr>
<tr>
<td>byte 4</td>
<td>byte 5</td>
</tr>
<tr>
<td>byte 8</td>
<td>byte 9</td>
</tr>
<tr>
<td>byte 12</td>
<td>byte 13</td>
</tr>
<tr>
<td>byte 16</td>
<td>byte 17</td>
</tr>
</tbody>
</table>

pointer points to the **big end**

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 3</td>
<td>byte 2</td>
</tr>
<tr>
<td>byte 7</td>
<td>byte 6</td>
</tr>
<tr>
<td>byte 11</td>
<td>byte 10</td>
</tr>
<tr>
<td>byte 15</td>
<td>byte 14</td>
</tr>
<tr>
<td>byte 19</td>
<td>byte 18</td>
</tr>
</tbody>
</table>

pointer points to the **little end**

- What difference does it make?

check out htonl(), ntohl() in in.h
Instruction Formats

◆ 3 simple formats
  - R-type, 3 register operands
    
    | 0 | rs | rt | rd | shamt | funct |
    |---|----|----|----|-------|-------|
    | 6-bit | 5-bit | 5-bit | 5-bit | 5-bit | 6-bit |
  
  - I-type, 2 register operands and 16-bit immediate operand
    
    | opcode | rs | rt | immediate |
    |--------|----|----|-----------|
    | 6-bit  | 5-bit | 5-bit | 16-bit |
  
  - J-type, 26-bit immediate operand
    
    | opcode | immediate |
    |--------|-----------|
    | 6-bit  | 26-bit |

◆ Simple Decoding
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b’ 00)
  - format and fields readily extractable
ALU Instructions

- **Assembly** (e.g., register-register signed addition)
  
  $\text{ADD rd}_{\text{reg}} \text{ rs}_{\text{reg}} \text{ rt}_{\text{reg}}$

- **Machine encoding**
  
<table>
<thead>
<tr>
<th>$0$</th>
<th>$\text{rs}$</th>
<th>$\text{rt}$</th>
<th>$\text{rd}$</th>
<th>$0$</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>6-bit</td>
</tr>
</tbody>
</table>

- **Semantics**
  
  - $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] + \text{GPR}[\text{rt}]$
  - $\text{PC} \leftarrow \text{PC} + 4$

- **Exception on “overflow”**

- **Variations**
  
  - Arithmetic: \{signed, unsigned\} x \{ADD, SUB\}
  - Logical: \{AND, OR, XOR, NOR\}
  - Shift: \{Left, Right-Logical, Right-Arithmetic\}
Reg-Reg Instruction Encoding

<table>
<thead>
<tr>
<th>2...0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SLL</td>
<td>*</td>
<td>SRL</td>
<td>SRA</td>
<td>SLLV</td>
<td>*</td>
<td>SRLV</td>
</tr>
<tr>
<td>1</td>
<td>JR</td>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>SYSCALL</td>
<td>BREAK</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>MFHI</td>
<td>MTHI</td>
<td>MFLO</td>
<td>MTLO</td>
<td>DSLLV(____)</td>
<td>*</td>
<td>DSRLV(____)</td>
</tr>
<tr>
<td>3</td>
<td>MULT</td>
<td>MULTU</td>
<td>DIV</td>
<td>DIVU</td>
<td>DMULT(____)</td>
<td>DMULTU(____)</td>
<td>DDIV(____)</td>
</tr>
<tr>
<td>4</td>
<td>ADD</td>
<td>ADDU</td>
<td>SUB</td>
<td>SUBU</td>
<td>AND</td>
<td>OR</td>
<td>XOR</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>SLT</td>
<td>SLTU</td>
<td>DADD(____)</td>
<td>DADDU(____)</td>
<td>DSUB(____)</td>
</tr>
<tr>
<td>6</td>
<td>TGE</td>
<td>TGEU</td>
<td>TLT</td>
<td>TLTU</td>
<td>TEQ</td>
<td>*</td>
<td>TNE</td>
</tr>
<tr>
<td>7</td>
<td>DSLL(____)</td>
<td>*</td>
<td>DSRL(____)</td>
<td>DSRA(____)</td>
<td>DSLL32(____)</td>
<td>*</td>
<td>DSRL32(____)</td>
</tr>
</tbody>
</table>

[MIPS R4000 Microprocessor User’s Manual]

What patterns do you see? Why are they there?
ALU Instructions

◆ Assembly (e.g., regi-immediate signed additions)
  \[
  \text{ADDI } r_t \text{ reg } r_s \text{ reg immediate}_{16}
  \]

◆ Machine encoding

<table>
<thead>
<tr>
<th>ADDI</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

◆ Semantics
  - GPR\[rt\] \leftarrow GPR\[rs\] + sign-extend (immediate)
  - PC \leftarrow PC + 4

◆ Exception on “overflow”

◆ Variations
  - Arithmetic: \{signed, unsigned\} x \{ADD, SUB\}
  - Logical: \{AND, OR, XOR, LUI\}
# Reg-Immed Instruction Encoding

<table>
<thead>
<tr>
<th>28...26</th>
<th>31...29</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>SPECIAL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>REGIMM</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>J</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>JAL</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>BEQ</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>BNE</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>BLEZ</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BGTZ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ADDI</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>ADDIU</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>SLTI</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>SLTIU</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>ANDI</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>ORI</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>XORI</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>LUI</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>COP0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>COP1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>COP2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>BEQ</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>BEQL</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>BNEL</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>BNE</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>BLEZ</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>BLEZL</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>BGTZ</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>BGTZL</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>DADDIe</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>DADDIUe</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>LDLε</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>LDRε</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>LB</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>LH</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>LWL</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>LW</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>LBU</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>LHU</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>LWR</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>LWUE</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>SB</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>SH</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>SWL</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>SW</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>SDLε</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>SDRε</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>SWR</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>CACHEδ</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>LL</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>LWC1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>LWC2</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>LLDε</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>LDC1</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>LDC2</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>LDε</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>SCDε</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>SC</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>SWC1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>SWC2</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>*</td>
</tr>
</tbody>
</table>

[MIPS R4000 Microprocessor User's Manual]
Assembly Programming 101

- Break down high-level program constructs into a sequence of elemental operations

- E.g. High-level Code
  \[ f = (g + h) - (i + j) \]

- Assembly Code
  - suppose \( f, g, h, i, j \) are in \( r_f, r_g, r_h, r_i, r_j \)
  - suppose \( r_{\text{temp}} \) is a free register
  
  \[
  \begin{align*}
  \text{add} & \ r_{\text{temp}} \ r_g \ r_h \quad \# \ r_{\text{temp}} = g+h \\
  \text{add} & \ r_f \ r_i \ r_j \quad \# \ r_f = i+j \\
  \text{sub} & \ r_f \ r_{\text{temp}} \ r_f \quad \# \ f = r_{\text{temp}} - r_f
  \end{align*}
  \]
Load Instructions

- Assembly (e.g., load 4-byte word)
  \[ \text{LW } \text{rt}_{\text{reg}} \text{ offset}_{16} (\text{base}_{\text{reg}}) \]

- Machine encoding
  \[
  \begin{array}{cccc}
  \text{LW} & \text{base} & \text{rt} & \text{offset} \\
  \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{16-bit} \\
  \end{array}
  \]

- Semantics
  - effective_address = sign-extend(offset) + GPR[base]
  - GPR[rt] ← MEM[ translate(effective_address) ]
  - PC ← PC + 4

- Exceptions
  - address must be “word-aligned”
    What if you want to load an unaligned word?
  - MMU exceptions
Store Instructions

- Assembly (e.g., store 4-byte word)
  
  \[ \text{SW } rt_{\text{reg}} \text{ offset}_{16} (\text{base}_{\text{reg}}) \]

- Machine encoding

<table>
<thead>
<tr>
<th>SW</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

- Semantics
  - effective_address = sign-extend(offset) + GPR[base]
  - MEM[ translate(effective_address) ] ← GPR[rt]
  - PC ← PC + 4

- Exceptions
  - address must be “word-aligned”
  - MMU exceptions
Assembly Programming 201

- E.g. High-level Code
  \[ A[8] = h + A[0] \]

  where \( A \) is an array of integers (4-byte each)

- Assembly Code
  - suppose \&A, \( h \) are in \( r_A, r_h \)
  - suppose \( r_{\text{temp}} \) is a free register

```
LW  r_{\text{temp}}  0(r_A)  # r_{\text{temp}} = A[0]
add r_{\text{temp}}  r_h  r_{\text{temp}}  # r_{\text{temp}} = h + A[0]
SW  r_{\text{temp}}  32(r_A)  # A[8] = r_{\text{temp}}

# note A[8] is 32 bytes
#    from A[0]
```
Load Delay Slots

- R2000 load has an architectural latency of 1 inst*.
  - the instruction immediately following a load (in the “delay slot”) still sees the old register value
  - the load instruction no longer has an atomic semantics

Why would you do it this way?

- Is this a good idea? (hint: R4000 redefined LW to complete atomically)

*BTW, notice that latency is defined in “instructions” not cyc. or sec.
Control Flow Instructions

- **C-Code**

```
{ code A }
if X==Y then
  { code B }
else
  { code C }
{ code D }
```

These things are called basic blocks.
(Conditional) Branch Instructions

- Assembly (e.g., branch if equal)
  \[ \text{BEQ } rs_{\text{reg}} \text{ rt}_{\text{reg}} \text{ immediate}_{16} \]

- Machine encoding
  
  \[
  \begin{array}{|c|c|c|c|}
  \hline
  \text{BEQ} & \text{rs} & \text{rt} & \text{immediate} \\
  \hline
  \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{16-bit} \\
  \hline
  \end{array}
  \]

- Semantics
  - target = PC + sign-extend(immediate) \times 4
  - if GPR[rs] == GPR[rt] then \( \text{PC} \leftarrow \text{target} \) 
    else \( \text{PC} \leftarrow \text{PC} + 4 \)

- How far can you jump?

- Variations
  - BEQ, BNE, BLEZ, BGTZ

Why isn’t there a BLE or BGT instruction?
Jump Instructions

- **Assembly**
  
  $J \text{ immediate}_{26}$

- **Machine encoding**

<table>
<thead>
<tr>
<th>J</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>26-bit</td>
</tr>
</tbody>
</table>
  
  J-type

- **Semantics**

  - target = $\text{PC}[31:28] \times 2^{28} | \text{bitwise-or} \text{ zero-extend}(\text{immediate}) \times 4$

  - $\text{PC} \leftarrow \text{target}$

- **How far can you jump?**

- **Variations**

  - Jump and Link
  - Jump Registers

- $\text{PC} + 4 \text{ w/ branch delay slot}$
Assembly Programming 301

◆ E.g. High-level Code

```
if (i == j) then
  e = g
else
  e = h
f = e
```

◆ Assembly Code

- suppose e, f, g, h, i, j are in r_e, r_f, r_g, r_h, r_i, r_j

```
bne r_i r_j L1  # L1 and L2 are addr labels
add r_e r_g r0  # assembler computes offset
j L2
L1:  add r_e r_h r0  # e = h
L2:  add r_f r_e r0  # f = e
      ... ...
```
Branch Delay Slots

- R2000 branch instructions also have an architectural latency of 1 instructions
  - the instruction immediately after a branch is always executed (in fact PC-offset is computed from the delay slot instruction)
  - branch target takes effect on the 2\textsuperscript{nd} instruction

\begin{verbatim}
 L1:    add r_e r_h r0
 bne r_i r_j L1
         add r_e r_g r0
         j L2

 L2:    add r_f r_e r0
 . . .  
\end{verbatim}
Strangeness in the Semantics

Where do you think you will end up?

<table>
<thead>
<tr>
<th></th>
<th><em>s</em></th>
<th>j</th>
<th>L1</th>
<th>j</th>
<th>L2</th>
<th>j</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1</td>
<td>j</td>
<td>L4</td>
<td>j</td>
<td>L5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>j</td>
<td></td>
<td>foo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td></td>
<td>foo</td>
<td></td>
<td>bar</td>
<td>baz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|   |     |     |     |     |     |     |     |
Function Call and Return

- **Jump and Link:**  
  
  - JAL offset\_26
  - return address = PC + 8
  - target = PC[31:28]x2\_28 | bitwise-or zero-extend(immediate)x4
  - PC ← target
  - GPR[r31] ← return address

  On a function call, the callee needs to know where to go back to afterwards.

- **Jump Indirect:**  
  
  - JR rs\_reg
  - target = GPR [rs]
  - PC ← target

  PC-offset jumps and branches always jump to the same target every time the same instruction is executed. Jump Indirect allows the same instruction to jump to any location specified by rs (usually r31)
How do you pass argument between caller and callee?

If A set r10 to 1, what is the value of r10 when B returns to C?

What registers can B use?

What happens to r31 if B calls another function?
Caller and Callee Saved Registers

◆ Callee-Saved Registers
  - Caller says to callee, “The values of these registers should not change when you return to me.”
  - Callee says, “If I need to use these registers, I promise to save the old values to memory first and restore them before I return to you.”

◆ Caller-Saved Registers
  - Caller says to callee, “If there is anything I care about in these registers, I already saved it myself.”
  - Callee says to caller, “Don’t count on them staying the same values after I am done.”
R2000 Register Usage Convention

- r0: always 0
- r1: reserved for the assembler
- r2, r3: function return values
- r4~r7: function call arguments
- r8~r15: “caller-saved” temporaries
- r16~r23: “callee-saved” temporaries
- r24~r25: “caller-saved” temporaries
- r26, r27: reserved for the operating system
- r28: global pointer
- r29: stack pointer
- r30: callee-saved temporaries
- r31: return address
R2000 Memory Usage Convention

- **Stack Space**: Grow down.
- **Free Space**: Grow up.
- **Dynamic Data**
- **Static Data**
- **Text**
- **Reserved**

High address

Low address

**Stack Pointer**: GPR[r29]

**Binary Executable**
Calling Convention

1. caller saves caller-saved registers
2. caller loads arguments into r4~r7
3. caller jumps to callee using JAL
4. callee allocates space on the stack (dec. stack pointer)
5. callee saves callee-saved registers to stack (also r4~r7, old r29, r31)
6. callee loads results to r2, r3
7. callee restores saved register values
8. JR r31
9. caller continues with return values in r2, r3
To Summarize: MIPS RISC

◆ Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - few alternatives for accomplishing the same thing

◆ Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - “Load-store” architecture

◆ Simple branches
  - limited varieties of branch conditions and targets

◆ Simple instruction encoding
  - all instructions encoded in the same number of bits
  - only a few formats

Loosely speaking, an ISA intended for compilers rather than assembly programmers