18-447 MIPS ISA

James C. Hoe
Dept of ECE, CMU
Instruction Set Architecture

- A stable platform, typically 15~20 years
  - guarantees binary compatibility for SW investments
  - permits adoption of foreseeable technology advances

- User-level ISA
  - program visible state and instructions available to user processes
  - single-user abstraction on top of HW/SW virtualization

- “Virtual Environment” Architecture
  - state and instructions to control virtualization (e.g., caches, sharing)
  - user-level, but not used by your average user programs

- “Operating Environment” Architecture
  - state and instructions to implement virtualization
  - privileged/protected access reserved for OS
Terminologies

- **Instruction Set Architecture**
  - the machine behavior as observable and controllable by the programmer

- **Instruction Set**
  - the set of commands understood by the computer

- **Machine Code**
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware

- **Assembly Code**
  - a collection of instructions expressed in “textual” format
    - e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code
    (mostly true: compound instructions, address labels ....)
What are specified/decided in an ISA?

- Data format and size
  - character, binary, decimal, floating point, negatives
- “Programmer Visible State”
  - memory, registers, program counters, etc.
- Instructions: how to transform the programmer visible state?
  - what to perform and what to perform next
  - where are the operands
- Instruction-to-binary encoding
- How to interface with the outside world?
- Protection and privileged operations
- Software conventions

Very often you compromise immediate optimality for future scalability and compatibility
**MIPS R2000 Program Visible State**

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

<table>
<thead>
<tr>
<th>M[0]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M[1]</td>
<td></td>
</tr>
<tr>
<td>M[2]</td>
<td></td>
</tr>
<tr>
<td>M[3]</td>
<td></td>
</tr>
<tr>
<td>M[4]</td>
<td></td>
</tr>
<tr>
<td>M[N-1]</td>
<td></td>
</tr>
</tbody>
</table>

`**Note**` $r0=0$
- r1
- r2

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

**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 . . . . . . .

Big Endian

<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

Little Endian

<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)
  
  ADD $rd \leftarrow reg \quad rs \quad reg \quad rt \quad reg$

- Machine encoding

<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>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
  - GPR[$rd] \leftarrow GPR[rs] + GPR[rt]
  - PC \leftarrow 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

#### Special Function Encoding

<table>
<thead>
<tr>
<th>5...3</th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SLL</td>
<td>*</td>
<td>SRL</td>
<td>SRA</td>
<td>SLLV</td>
<td>*</td>
<td>SRLV</td>
<td>SRAV</td>
</tr>
<tr>
<td>1</td>
<td>JR</td>
<td>JALR</td>
<td>*</td>
<td></td>
<td>SYSCALL</td>
<td>BREAK</td>
<td>*</td>
<td>SYNC</td>
</tr>
<tr>
<td>2</td>
<td>MFHI</td>
<td>MTHI</td>
<td>MFLO</td>
<td>MTLO</td>
<td>DSLLV&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>*</td>
<td>DSRLV&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DSRAV&lt;sub&gt;ε&lt;/sub&gt;</td>
</tr>
<tr>
<td>3</td>
<td>MULT</td>
<td>MULTU</td>
<td>DIV</td>
<td>DIVU</td>
<td>DMULT&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DMULTU&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DDIV&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DDIVU&lt;sub&gt;ε&lt;/sub&gt;</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>
<td>NOR</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>SLT</td>
<td>SLTU</td>
<td>DADD&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DADDU&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DSUB&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DSUBU&lt;sub&gt;ε&lt;/sub&gt;</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>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>DSLL&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>*</td>
<td>DSRL&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DSRA&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DSLL32&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>*</td>
<td>DSRL32&lt;sub&gt;ε&lt;/sub&gt;</td>
<td>DSRA32&lt;sub&gt;ε&lt;/sub&gt;</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)
  \[ \text{ADDI } rt_{\text{reg}} \; \text{rs}_{\text{reg}} \; \text{immediate}_{16} \]

- Machine encoding

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

I-type

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

- Exception on “overflow”

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

<table>
<thead>
<tr>
<th>31...29</th>
<th>28...26</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPECIAL</td>
<td>REGIMM J JAL BEQ BNE BLEZ BGTZ</td>
</tr>
<tr>
<td>1</td>
<td>ADDI</td>
<td>ADDIU SLTI SLTIU ANDI ORI XORI LUI</td>
</tr>
<tr>
<td>2</td>
<td>COP0</td>
<td>COP1 COP2 * BEQL BNEL BLEZL BGTZL</td>
</tr>
<tr>
<td>3</td>
<td>DADDIε</td>
<td>DADDIUε LDLε LDRε * * *</td>
</tr>
<tr>
<td>4</td>
<td>LB</td>
<td>LH LWL LW LBU LHU LWR LWUε</td>
</tr>
<tr>
<td>5</td>
<td>SB</td>
<td>SH SWL SW SDLε SDRε SWR CACHE δ</td>
</tr>
<tr>
<td>6</td>
<td>LL</td>
<td>LWC1 LWC2 * LLDε LDC1 LDC2 LDε</td>
</tr>
<tr>
<td>7</td>
<td>SC</td>
<td>SWC1 SWC2 * SCDε SDC1 SDC2 SDε</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
  
  ```assembly
  add r_{\text{temp}} r_g r_h  # r_{\text{temp}} = g+h
  add r_f r_i r_j  # r_f = i+j
  sub r_f r_{\text{temp}} r_f  # f = r_{\text{temp}} - r_f
  ```
Load Instructions

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

- **Machine encoding**

<table>
<thead>
<tr>
<th>LW</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]
  - GPR[rt] \leftarrow \text{MEM[ translate(effective_address) ]}
  - PC \leftarrow PC + 4

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

- LW/SW alignment restriction
  - not optimized to fetch memory bytes not within a word boundary
  - not optimized to rotate unaligned bytes into registers
- Provide separate opcodes for the infrequent case

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

- LWL rd 6(r0)
  - byte-6 | byte-5 | byte-4 | D
- LWR rd 3(r0)
  - byte-6 | byte-5 | byte-4 | byte-3

- LWL/LWR is slower but it is okay
- note LWL and LWR still fetch within word boundary
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>

I-type

◆ Semantics

- \( \text{effective\_address} = \text{sign\_extend}(\text{offset}) + \text{GPR}[\text{base}] \)
- \( \text{MEM}[\ \text{translate}(\text{effective\_address})]\ \leftarrow \text{GPR}[rt] \)
- \( \text{PC} \leftarrow \text{PC} + 4 \)

◆ Exceptions

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

◆ E.g. High-level Code

\[ \text{A}[8] = h + \text{A}[0] \]

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

◆ Assembly Code

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

\[
\begin{align*}
\text{LW} & \quad r_{temp} \ 0(r_A) & \# r_{temp} = \text{A}[0] \\
\text{add} & \quad r_{temp} \ r_h \ r_{temp} & \# r_{temp} = h + \text{A}[0] \\
\text{SW} & \quad r_{temp} \ 32(r_A) & \# \text{A}[8] = r_{temp} \\
\end{align*}
\]

# 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**

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

Control Flow Graph

- **True**
  - Code A
  - if X==Y
  - Code B
  - Code C
- **False**
  - Code D

Assembly Code (linearized)

- Code A
  - if X==Y
  - goto
- Code C
  - goto
- Code B
- Code D

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

- **Assembly** (e.g., branch if equal)
  \[
  \text{BEQ } rs_{\text{reg}} \ rt_{\text{reg}} \ immediate_{16}
  \]

- **Machine encoding**
  
<table>
<thead>
<tr>
<th>BEQ</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**
  - target = PC + sign-extend(immediate) x 4
  - if GPR[rs] == GPR[rt] then PC \(\leftarrow\) target
  - else PC \(\leftarrow\) PC + 4

- **How far can you jump?**

- **Variations**
  - BEQ, BNE, BLEZ, BGTZ

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

- **Assembly**
  \[ J \text{ immediate} \]

- **Machine encoding**

  \[
  \begin{array}{c|c}
  \text{J} & \text{immediate} \\
  \hline
  \text{6-bit} & \text{26-bit}
  \end{array}
  \]

  - **J-type**

- **Semantics**
  - target = PC[31:28]x2^{28} | bitwise-or zero-extend(immediate)x4
  - PC \leftarrow target

- **How far can you jump?**

- **Variations**
  - Jump and Link
  - Jump Registers
Assembly Programming 301

◆ E.g. High-level Code

```plaintext
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

```assembly
bne r_i r_j L1  # L1 and L2 are addr labels
add r_e r_g r0  # e = g
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 2nd instruction

```plaintext
bne r_i r_j L1
add r_e r_g r0
j L2
L1:  add r_e r_h r0
L2:  add r_f r_e r0
      . . . .
```

```plaintext
bne r_i r_j L1
nop
j L2
add r_e r_g r0
L1:  add r_e r_h r0
L2:  add r_f r_e r0
      . . . .
```
Strangeness in the Semantics

Where do you think you will end up?

\_s: \ j \ L1
\ j \ L2
\ j \ L3

L1: \ j \ L4
L2: \ j \ L5

L3: \ foo
L4: \ bar
L5: \ baz
Function Call and Return

- **Jump and Link:** \( \text{JAL \ offset}_{26} \)
  - return address = \( \text{PC} + 8 \)
  - target = \( \text{PC}[31:28] \times 2^{28} |_{\text{bitwise-or zero-extend(immediate)}} \times 4 \)
  - \( \text{PC} \leftarrow \text{target} \)
  - \( \text{GPR}[r31] \leftarrow \text{return address} \)

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

- **Jump Indirect:** \( \text{JR \ rs}_{\text{reg}} \)
  - target = \( \text{GPR}[rs] \)
  - \( \text{PC} \leftarrow \text{target} \)

  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).
Assembly Programming 401

.... A → \text{call} B → \text{return} C → \text{call} B → \text{return} D ....

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 Pointer**: GPR[r29]
- **Stack Space**: Grow down
- **Free Space**: Grow up
- **Dynamic Data**
- **Static Data**
- **Text**
- **Binary Executable**
- **Reserved**

High address

Low address
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
We didn’t talk about

- Privileged Modes
  - User vs. supervisor
- Exception Handling
  - Trap to supervisor handling routine and back
- Virtual Memory
  - Each user has 4-GBytes of private, large, linear and fast memory?
- Floating-Point Instructions