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
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

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

**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 ........

pointer points to the big end  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
    \[
    \begin{array}{cccccc}
    0 & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
    \text{R-type} & \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{5-bit} & \text{6-bit} \\
    \end{array}
    \]
  - I-type, 2 register operands and 16-bit immediate operand
    \[
    \begin{array}{ccc}
    \text{opcode} & \text{rs} & \text{rt} & \text{immediate} \\
    \text{I-type} & \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{16-bit} \\
    \end{array}
    \]
  - J-type, 26-bit immediate operand
    \[
    \begin{array}{ccc}
    \text{opcode} & \text{immediate} \\
    \text{J-type} & \text{6-bit} & \text{26-bit} \\
    \end{array}
    \]
- Simple Decoding
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned \((2\,\text{lsb of PC must be 2b'00})\)
  - 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
  \[
  \begin{array}{cccccc}
  0 & \text{rs} & \text{rt} & \text{rd} & 0 & \text{ADD} \\
  \text{R-type} & \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{5-bit} & \text{6-bit} \\
  \end{array}
  \]
- Semantics
  - \(\text{GPR[rd] } \leftarrow \text{GPR[rs]} + \text{GPR[rt]}\)
  - \(\text{PC } \leftarrow \text{PC} + 4\)
- Exception on “overflow”
- Variations
  - Arithmetic: \{signed, unsigned\} \times \{ADD, SUB\}
  - Logical: \{AND, OR, XOR, NOR\}
  - Shift: \{Left, Right-Logical, Right-Arithmetic\}
Reg-Reg Instruction Encoding

What patterns do you see? Why are they there?

ALU Instructions

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

* Machine encoding
  \[
  \begin{array}{cccc}
  \text{ADDI} & \text{rs} & \text{rt} & \text{immediate} \\
  \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{16-bit} \\
  \end{array}
  \]

* Semantics
  - \[ \text{GPR}[r_t] \leftarrow \text{GPR}[r_s] \text{ + sign-extend (immediate)} \]
  - \[ \text{PC} \leftarrow \text{PC} + 4 \]

* Exception on “overflow”

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

<table>
<thead>
<tr>
<th>31..29</th>
<th>28..26</th>
<th>25..22</th>
<th>21..18</th>
<th>17..14</th>
<th>13..10</th>
<th>9..6</th>
<th>5..2</th>
<th>1..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPECIAL</td>
<td>REGIMM</td>
<td>J</td>
<td>JAL</td>
<td>BEQ</td>
<td>BNE</td>
<td>BLEZ</td>
<td>BGTZ</td>
</tr>
<tr>
<td>1</td>
<td>ADDI</td>
<td>ADDIU</td>
<td>SLTI</td>
<td>SLTIU</td>
<td>ANDI</td>
<td>ORI</td>
<td>XORI</td>
<td>LUI</td>
</tr>
<tr>
<td>2</td>
<td>COP0</td>
<td>COP1</td>
<td>COP2</td>
<td>*</td>
<td>BEQ</td>
<td>BNE</td>
<td>BLEZ</td>
<td>BGTZL</td>
</tr>
<tr>
<td>3</td>
<td>DADD</td>
<td>DADDU</td>
<td>LDL</td>
<td>LDR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>LB</td>
<td>LH</td>
<td>LW</td>
<td>LW</td>
<td>LBU</td>
<td>LHU</td>
<td>LWR</td>
<td>LWU</td>
</tr>
<tr>
<td>5</td>
<td>SB</td>
<td>SH</td>
<td>SW</td>
<td>SW</td>
<td>SDL</td>
<td>SDR</td>
<td>SWR</td>
<td>CACHE</td>
</tr>
<tr>
<td>6</td>
<td>LL</td>
<td>LWC1</td>
<td>LWC2</td>
<td>*</td>
<td>LDL</td>
<td>LDC1</td>
<td>LDC2</td>
<td>LDC</td>
</tr>
<tr>
<td>7</td>
<td>SC</td>
<td>SWC1</td>
<td>SWC2</td>
<td>*</td>
<td>SDC</td>
<td>SDC1</td>
<td>SDC2</td>
<td>SDC</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
  ```
  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)
  \[ \text{LW } r_t \text{ offset}_{16} \text{ (base}_{reg} \text{)} \]
- Machine encoding
  \[
  \begin{array}{cccc}
  \text{LW} & \text{base} & \text{rt} & \text{offset} \\
  \end{array}
  \]
  6-bit 5-bit 5-bit 16-bit
- 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

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
  \[
  \begin{array}{cccc}
  \text{LWL } & \text{rd} & 6(\text{r0}) & \text{byte-6} & \text{byte-5} & \text{byte-4} & \text{D} \\
  \text{LWR } & \text{rd} & 3(\text{r0}) & \text{byte-6} & \text{byte-5} & \text{byte-4} & \text{byte-3} \\
  \end{array}
  \]
  - 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)
  \[
  SW \ rt_{reg} \ offset_{16} (base_{reg})
  \]
- 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
  - \( \text{effective_address} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}] \)
  - \( \text{MEM} \left[ \text{translate}(\text{effective_address}) \right] \leftarrow \text{GPR[rt]} \)
  - \( \text{PC} \leftarrow \text{PC} + 4 \)
- Exceptions
  - address must be “word-aligned”
  - MMU exceptions

Assembly Programming 201

- E.g. High-level Code

\[
A[8] = h + A[0]
\]

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

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

# 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 *BTW, notice 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)**
  \[ \text{BEQ } rs_{reg} \text{ rt}_{reg} \text{ immediate}_{16} \]
- **Machine encoding**
  \[
  \begin{array}{cccc}
  \text{BEQ} & \text{rs} & \text{rt} & \text{immediate} \\
  \text{6-bit} & \text{5-bit} & \text{5-bit} & \text{16-bit}
  \end{array}
  \]
- **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**
  \[ \text{J} \text{ immediate}_{26} \]
- **Machine encoding**
  \[
  \begin{array}{c}
  \text{J} \\
  \text{6-bit}
  \end{array}
  \quad
  \begin{array}{c}
  \text{immediate} \\
  \text{26-bit}
  \end{array}
  \quad
  \begin{array}{c}
  \text{J-type}
  \end{array}
  \]
- **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
  
  ```
  if (i == j) then
      e = g
  else
      e = h
  f = e
  ```

- Assembly Code
  
  suppose e, f, g, h, i, j are in re, rf, rg, rh, ri, rj

  ```
  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

  ```
  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
  ```
**Strangeness in the Semantics**

Where do you think you will end up?

\[
\begin{align*}
  _s & : j L1 \\
  & : j L2 \\
  & : j L3 \\
L1 & : j L4 \\
L2 & : j L5 \\
L3 & : \text{foo} \\
L4 & : \text{bar} \\
L5 & : \text{baz}
\end{align*}
\]

**Function Call and Return**

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

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

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

  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

**Caller**

```
... code A ... 
JAL _myfxn 
... code C ... 
JAL _myfxn 
... code D ... 
```

- ..... A → call B → return C → call B → 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

**Callee**

```
_myfxn: ... code B ... 
J R r31 
```

**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**

- **Stack pointer:** GPR[r29]

- **Binary executable:**
Calling Convention

- caller saves caller-saved registers
- caller loads arguments into r4~r7
- caller jumps to callee using JAL
- callee allocates space on the stack (dec. stack pointer)
- callee saves callee-saved registers to stack (also r4~r7, old r29, r31)
- body of callee (can “nest” additional calls)
- callee loads results to r2, r3
- callee restores saved register values
- JR r31
- 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