Chapter 2

Instruction Set Architecture (ISA)
MIPS arithmetic

• Design Principle: simplicity favors regularity. Why?
• Of course this complicates some things...

C code: \[ A = B + C + D; \]
\[ E = F - A; \]

MIPS code: (s0 ← A, s1 ← B, s2 ← C, s3 ← D, s4 ← E, s5 ← F)

  add $t0, $s1, $s2  # t0 ← s1 + s2
  add $s0, $t0, $s3  # s0 ← t0 + s3
  sub $s4, $s5, $s0  # s4 ← s5 - s0

• Operands must be registers, only 32 registers provided
• Design Principle: smaller is faster. Why?
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

\[2^{32}\] 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?

Registers hold 32 bits of data

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

• Load and store instructions  \((\text{Memory Read and Write})\)

• Example:

**C code:** \[ A[8] = h + A[8]; \]

**MIPS code:**

```
lw $t0, 32($s3)     # t0 ↔ M[s3+32]
add $t0, $s2, $t0   # t0 ↔ s2 + t0
sw $t0, 32($s3)     # M[s3+32] ↔ t0
```

• Store word has destination last
• Remember arithmetic operands are registers, not memory!
So far we’ve learned:

- **MIPS**
  - loading words but addressing bytes
  - arithmetic on registers only

### Instruction | Meaning
--- | ---
`add $s1, $s2, $s3` | $s1 = $s2 + $s3
`sub $s1, $s2, $s3` | $s1 = $s2 - $s3
`lw $s1, 100($s2)` | $s1 = Memory[$s2+100]
`sw $s1, 100($s2)` | Memory[$s2+100] = $s1
Machine Language

• Instructions, like registers and words of data, are also 32 bits long
  – Example: \( add \ $t0, \ $s1, \ $s2 \)
  – registers have numbers, \( $t0=8, \ $s1=17, \ $s2=18 \)

• Instruction Format:

<table>
<thead>
<tr>
<th>Code</th>
<th>num.bits</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000 10001 10010 01000 00000 100000</td>
<td>6 5 5 5 5 6</td>
<td>op rs rt rd shamt funct</td>
</tr>
</tbody>
</table>

• Can you guess what the field names stand for?
Machine Language

• Consider the load-word and store-word instructions,
  – What would the regularity principle have us do?
  – New principle: 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)`

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>8</td>
<td>18</td>
<td>32</td>
</tr>
</tbody>
</table>

• Where's the compromise?
Control

• 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: ....
```
Control

• MIPS unconditional branch instructions:
  \[ j \text{ label} \]

• Example:

  \[
  \text{if (i!=j)} \quad \text{beq } \$s4, \$s5, \text{Lab1} \\
  \text{h=i+j; } \quad \text{add } \$s3, \$s4, \$s5 \\
  \text{else} \quad \text{Lab2} \\
  \text{h=i-j; } \quad \text{Lab1:sub } \$s3, \$s4, \$s5 \\
  \text{Lab2:...}
  \]

• Can you build a simple for loop?
So far:

- **Instruction**

  - `add $s1,$s2,$s3`  
    - Meaning: $s1 = s2 + s3$
  - `sub $s1,$s2,$s3`  
    - Meaning: $s1 = s2 - s3$
  - `lw $s1,100($s2)`  
    - Meaning: $s1 = \text{Memory}[$s2+100$]
  - `sw $s1,100($s2)`  
    - Meaning: \text{Memory}[$s2+100$] = $s1$
  - `bne $s4,$s5,L`  
    - Next instr. is at Label if $s4 \neq s5$
  - `beq $s4,$s5,L`  
    - Next instr. is at Label if $s4 = s5$
  - `j Label`  
    - Next instr. is at Label

- **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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>
Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

\[
\text{if } \$s1 < \$s2 \\
\text{then } \$t0 = 1 \\
\text{slt } \$t0, \$s1, \$s2 \\
\text{else } \$t0 = 0
\]

- Can use this instruction to build "\text{blt } \$s1, \$s2, Label" — can now build general control structures
- Note that the assembler needs a register to do this, — there are policy of use conventions for registers
## Conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Constants

• Small constants are used quite frequently (50% of operands)
  e.g.,
  \[ A = A + 5; \]
  \[ B = B + 1; \]
  \[ C = C - 18; \]

• Solutions? Why not?
  – put 'typical constants' in memory and load them.
  – create hard-wired registers (like $zero) for constants like one.

• MIPS Instructions:

  \[
  \begin{align*}
  \text{addi} & \quad $29, \quad $29, \quad 4 \\
  \text{slti} & \quad $8, \quad $18, \quad 10 \\
  \text{andi} & \quad $29, \quad $29, \quad 6 \\
  \text{ori} & \quad $29, \quad $29, \quad 4
  \end{align*}
  \]

• How do we make this work?
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:
  \[
  \text{lui } \$t0, \ 1010101010101010 \]

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
\end{array}
\]

- Then must get the lower order bits right, i.e.,
  \[
  \text{ori } \$t0, \$t0, \ 1010101010101010 \]

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
0000000000000000 & 1010101010101010 \\
\end{array}
\]

\[
\begin{array}{c|c}
1010101010101010 & 1010101010101010 \\
\end{array}
\]
Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., "move $t0, $t1" exists only in Assembly
  - would be implemented using "add $t0,$t1,$zero"
- When considering performance you should count real instructions
Other Issues

• Things we are not going to cover
  support for procedures
  linkers, loaders, memory layout
  stacks, frames, recursion
  manipulating strings and pointers
  interrupts and exceptions
  system calls and conventions

• Some of these we'll talk about later

• We've focused on architectural issues
  – basics of MIPS assembly language and machine code
  – we’ll build a processor to execute these instructions.
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>16 bit address</td>
<td></td>
</tr>
<tr>
<td></td>
<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
Addresses in Branches and Jumps

• 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>op</th>
<th>26 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Addresses are not 32 bits
  — How do we handle this with load and store instructions?
Addresses in Branches

- **Instructions:**
  - `bne $t4,$t5,Label` Next instruction is at Label if $t4/=$t5
  - `beq $t4,$t5,Label` Next instruction is at Label if $t4=$t5

- **Formats:**
  - I 
    - `op` | `rs` | `rt` | 16 bit address

- Could specify a register (like `lw` and `sw`) and add it to address
  - 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
# MIPS Operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td></td>
</tr>
<tr>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
<td>$Memory[0], $Memory[4], \ldots, $Memory[4294967292]</td>
<td></td>
</tr>
</tbody>
</table>
Data

1. Immediate addressing

   \[
   \begin{array}{c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{Immediate} \\
   \hline
   \end{array}
   \]

2. Register addressing

   \[
   \begin{array}{c|c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{funct} \\
   \hline
   \end{array}
   \]

3. Base addressing

   \[
   \begin{array}{c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \hline
   \end{array}
   \]

4. PC-relative addressing

   \[
   \begin{array}{c|c|c|c|c}
   \text{op} & \text{rs} & \text{rt} & \text{Address} \\
   \hline
   \end{array}
   \]

5. Pseudodirect addressing

   \[
   \begin{array}{c|c}
   \text{op} & \text{Address} \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{PC} \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{Memory} \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{Word} \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{Byte} \quad \text{Halfword} \quad \text{Word} \\
   \hline
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{Register} \\
   \hline
   \end{array}
   \]
Alternative Architectures

• Design alternative:
  – provide more powerful operations
  – goal is to reduce number of instructions executed
  – danger is a slower cycle time and/or a higher CPI

• Sometimes referred to as “RISC vs. CISC”
  – virtually all new instruction sets since 1982 have been RISC
  – VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!

• We’ll look at PowerPC and 80x86
PowerPC

- Indexed addressing
  - example: `lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?

- Others:
  - load multiple/store multiple
  - a special counter register “bc Loop”
    - decrement counter, if not 0 goto loop
Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- 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!