Concocting an Instruction Set

```
move flour, bowl
add milk, bowl
add egg, bowl
move bowl, mixer
rotate mixer
...
```

Nerd Chef at work.

A General-Purpose Computer
The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:

Central Processing Unit (CPU): containing several registers, as well as logic for performing a specified set of operations on their contents.

Memory: storage of N words of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.

I/O: Devices for communicating with the outside world.
The Stored Program Computer

The von Neumann architecture easily addresses the first two limitations of our simple programmable machine example:

• A richer repertoire of operations, and
• An expandable memory.

But how does it achieve programmability?

Key idea: Memory holds not only data, but coded instructions that make up a program.

CPU fetches and executes — interprets — successive instructions of the program ...

• Program is simply data for the interpreter — as in a Universal Turing Machine!
• Single expandable resource pool — main memory — constrains both data and program size.

Anatomy of a von Neumann Computer

• INSTRUCTIONS coded as binary data
• PROGRAM COUNTER or PC: Address of next instruction to be executed
• Logic to translate instructions into control signals for data path
Instruction Set Architecture (ISA)

**Coding of instructions raises some interesting choices...**

- **Tradeoffs**: performance, compactness, programmability
- **Uniformity**: Should different instructions
  - Be the same size?
  - Take the same amount of time to execute?
- **Complexity**: How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
  - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed

**Mix of Engineering & Art...**

- Trial (by simulation) is our best technique for making choices!

**Our representative example: the miniMIPS architecture!**

---

**MIPS Programming Model**

*a representative, simple, contemporary RISC*

**Processor State**
- PC: 00
- r0, r1, r2, ..., r31

**Main Memory**
- 32-bit "words" (4 bytes)
- next instruction

**Fetch/Execute loop:**
- fetch Mem[PC]
- PC = PC + 4†
- execute fetched instruction (may change PC!)
- repeat!

†MIPS uses byte memory addresses. However, each instruction is 32-bits wide, and "must" be aligned on a multiple of 4 (word) address. Each word contains four B-bit bytes. Addresses of consecutive instructions (words) differ by 4.
MIPS Instruction Formats

All MIPS instructions fit in a single 32-bit word. Every instruction includes various “fields” that encode combinations of:

- a 6-bit operation or “OPCODE” specifying one of < 64 basic operations
- escape codes to enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants (“immediate” values) of various sizes, 16-bits, 5-bits, and 26-bits. Sometimes treated as signed values, sometimes not.

There are three basic instruction formats:

- **R-type**, 3 register operands (2 sources, destination)
  
<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

- **I-type**, 2 register operands, 16-bit literal constant
  
<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>16-bit constant</th>
</tr>
</thead>
</table>

- **J-type**, no register operands, 26-bit literal constant
  
<table>
<thead>
<tr>
<th>OP</th>
<th>26-bit constant</th>
</tr>
</thead>
</table>

MIPS ALU Operations

Sample coded operation: ADD instruction

- **R-type**: The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.
  
  | 00000000101101010100000010000 |
  |----|----|----|----|----|----|
  | op = 0x00 | dictating an ALU function |
  | rt = 9 Reg[9] | source |
  | rd = 10 Reg[10] | destination |
  | func = 0x20 | dictating an add |

- References to register contents are prefixed by a “$” to distinguish them from constants or memory addresses.

What we prefer to write: add $10, $11, $9 ("assembly language")

```
add rd, rs, rt:
Reg[rd] = Reg[rs] + Reg[rt]
```

“Add the contents of rs to the contents of rt; store the result in rd”

Similar instructions for other ALU operations:

- arithmetic: add, sub, addu, subu, mult, mulu, div, divu
- compare: slt, sltu
- logical: and, or, xor, nor
- shift: sll, srl, sra, sllv, srav, srlv
MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction

R-type:

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

- \( \text{op} = 0x00 \): dictating an ALU function
- \( \text{rt} = 2 \): Reg[2] source
- \( \text{rd} = 2 \): Reg[2] destination
- \( \text{shamt} = 4 \): dictates a shift of 4-bits
- \( \text{func} = 0x00 \): dictating an \text{sll}

Assembly:

```
sll $2, $2, 4
```

```
sllv $2, $2, $8
```

- \( \text{Reg}[\text{rd}] = \text{Reg}[\text{rt}] \ll \text{shamt} \)
- \( \text{Reg}[\text{rd}] = \text{Reg}[\text{rt}] \ll \text{Reg}[\text{rs}] \)

This is peculiar syntax for MIPS; in this ALU instruction the \( \text{rt} \) operand precedes the \( \text{rs} \) operand, usually, it's the other way around.

MIPS ALU Operations with Immediate

\text{addi} \hspace{1em} \text{instruction: adds register contents, signed-constant:}

\begin{align*}
\text{I-type:} & \hspace{1em} \begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array} \\
\text{OP} & = 0x08, \hspace{1em} \text{dictating addi} \\
\text{rs} & = 11, \hspace{1em} \text{Reg}[11] \hspace{1em} \text{source} \\
\text{rt} & = 9, \hspace{1em} \text{Reg}[9] \hspace{1em} \text{destination} \\
\text{constant field} & \text{indicating -3 as second operand (sign-extended!)}
\end{align*}

Symbolic version: \text{addi} \hspace{1em} $9, \hspace{1em} $11, \hspace{1em} -3

\text{addi} \hspace{1em} \text{rt, rs, imm:}

\[
\text{Reg}[\text{rt}] = \text{Reg}[\text{rs}] + \text{ext}(\text{imm})
\]

"Add the contents of \( \text{rs} \) to \( \text{const} \); store the result in \( \text{rt} \)"

Similar instructions for other ALU operations:

- arithmetic: \text{addi, addiu}
- compare: \text{slt, sltu}
- logical: \text{andi, ori, xor, lui}
Why Do Built-in Constants?

Solutions? Why not?
- put constants in memory (was common in older instruction sets)
- create more hard-wired registers for constants (like $0).

SMALL constants are used frequently (50% of operands)
- In a C compiler (gcc) 52% of ALU operations involve a constant
- In a circuit simulator (spice) 69% involve constants
  e.g., \( B = B + 1; C = W \& 0x00ff; A = B + 0; \)

ISA Design Principle: Make the common cases fast

MIPS Instructions:

```
addi $29, $29, 4
sla $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

One way to answer architectural questions is to evaluate the consequences of different choices using carefully chosen representative benchmarks (programs and/or code sequences). Make choices that are "best" according to some metric (cost, performance, ...).

How About Larger Constants?

In order to load a 32-bit constant into a register a two instruction sequence is used, "load upper immediate"

```
lui $8, 1010101010101010

1010101010101010 0000000000000000
```

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

```
ori $8, $8, 1010101010101010

1010101010101010 0000000000000000
```

Reminder: In MIPS, Logical Immediate Instructions do not align-extend their constant operand.
First MIPS Program
(fragment)

Suppose you want to compute the following expression:

\[ f = (g + h) - (i + j) \]

Where the variables \( f, g, h, i, \) and \( j \) are assigned to registers \$16, \$17, \$18, \$19, \) and \$20 respectively. What is the MIPS assembly code?

```
add $8,$17,$18     # (g + h)
add $9,$19,$20     # (i + j)
sub $16,$8,$9      # f = (g + h) - (i + j)
```

These three instructions do what our little ad-hoc factorial machine did. Of course, limiting ourselves to registers for storage falls short of our ambitions.... it amounts to the finite storage limitations of an FSM!

Needed: instruction-set support for reading and writing locations in main memory...

MIPS Load & Store Instructions

MIPS is a LOAD/STORE architecture. This means that data memory accesses are limited to load and store instructions, which transfer register contents to-and-from memory. ALU operations work only on registers.

I-type:

\[ \text{lw \ rt, imm(rs)} \quad \text{sw \ rt, imm(rs)} \]

\[ \text{Reg[rt]} = \text{Mem[Reg[rs] + sxt(const)]]} \]

"Fetch into \( rt \) the contents of the memory location whose address is const plus the contents of \( rs \)"

Abbreviation: \( \text{lw \ rt, \ imm} \) for \( \text{lw \ rt, imm(0)} \)

\[ \text{Mem[Reg[rs] + sxt(const)]]} = \text{Reg[rt]} \]

"Store the contents of \( rt \) into the memory location whose address is const plus the contents of \( rs \)"

Abbreviation: \( \text{sw \ rt, \ imm} \) for \( \text{sw \ rt, imm(0)} \)

BYTE ADDRESSES, but \( \text{lw} \) and \( \text{sw} \) 32-bit word access word-aligned addresses. Lowest two address bits must be 0!
Storage Conventions

- Data and Variables are stored in memory
- Operations done on registers
- Registers hold Temporary results

Example:
```
int x, y;
y = x + 37;
```

```
lw $t0, 0x1008($0)
add $t0, $t0, 37
sw $t0, 0x100C($0)
```

MIPS Register Usage Conventions

By convention, the MIPS registers are assigned to specific uses, and names. These are supported by the assembler, and higher-level languages. We'll use these names increasingly.

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>
Common “Addressing Modes”

MIPS can do these with appropriate choices for Ra and const

- **Absolute**: lw $0, 0x1000
  - Value = Mem[constant]
  - Use: accessing static data
- **Indirect**: lw $0, ($9)
  - Value = Mem[Reg[x]]
  - Use: pointer accesses
- **Displacement**: lw $0, 16($9)
  - Value = Mem[Reg[x] + constant]
  - Use: access to local variables
- **Indexed**: lw $0, 16($9)
  - Value = Mem[Reg[x] + Reg[y]]
  - Use: array accesses (base+Index)
- **Memory indirect**: lw $0, Mem[Reg[x]]
  - Value = Mem[Mem[Reg[x]]]
  - Use: access thru pointer in mem
- **Autoincrement**: lw $0, 16(Reg[x])
  - Value = Mem[Reg[x]]; Reg[x]++
  - Use: sequential pointer accesses
- **Autodecrement**: lw $0, -16(Reg[x])
  - Value = Mem[Reg[x]]; Reg[x]--
  - Use: stack operations
- **Scaled**: lw $0, 16(Reg[x])
  - Value = Mem[Reg[x] + c + d*Reg[y]]
  - Use: array accesses (base+index)

Argh! Is the complexity worth the cost? Need a cost/benefit analysis!

Memory Operands: Usage

Usage of different memory operand modes

© 2003 Elsevier Science (USA). All rights reserved.
Capability so far: Expression Evaluation

Translation of an Expression:

```plaintext
int x, y;
y = (x-3)*(y+123456)
```

```plaintext
x: .word 0
y: .word 0
c: .word 123456
...
lw $t0, x
addi $t0, $t0, -3
lw $t1, y
lw $t2, c
add $t1, $t1, $t2
mul $t0, $t0, $t1
sw $t0, y
```

- VARIABLES are allocated storage in main memory
- VARIABLE references translate to LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to ALU instructions w/ built-in constant
- "LARGE" CONSTANTS translate to initialized variables

NB: Here we assume that variable addresses fit into 16-bit constants!

Can We Run Any Algorithm?

Model thus far:
- Executes instructions sequentially
- Number of operations executed = number of instructions in our program!

Good news: programs can't "loop forever"!
- Halting problem is solvable for our current MIPS subset!

Bad news: can't compute Factorial:
- Only supports bounded-time computations;
- Can't do a loop, e.g. for Factorial!

Needed: ability to change the PC.
MIPS Branch Instructions

MIPS branch instructions provide a way of conditionally changing the PC to some nearby location...

\[
\text{I-type: } \begin{array}{c|c|c|c|}
\text{OPCODE} & \text{rs} & \text{rt} & \text{16-bit signed constant} \\
\end{array}
\]

\[
\text{beq rs, rt, label } & \quad \text{bne rs, rt, label } \\
\text{if } (\text{REG[RS]} == \text{REG[RT]}) & \quad \text{if } (\text{REG[RS]} != \text{REG[RT]}) \\
\{ & \{ \\
\quad \text{PC} = \text{PC} + 4 + 4\times\text{offset}; & \quad \text{PC} = \text{PC} + 4 + 4\times\text{offset}; \\
\} & \}
\]

NB: Branch targets are specified relative to the current instruction (actually relative to the next instruction, which would be fetched by default). The assembler hides the calculation of these offset values from the user, by allowing them to specify a target address (usually a label) and it does the job of computing the offset’s value. The size of the constant field (16-bits) limits the range of branches.

MIPS Jumps

The range of MIPS branch instructions is limited to approximately \(\pm 64K\) instructions from the branch instruction. In order to branch farther an unconditional jump instruction is used.

Instructions:

\[
\begin{align*}
\text{j label} & \quad \# \text{jump to label } (\text{PC} = \text{PC}[31-28] \| \text{CONST}[25:0]<<2) \\
\text{jal label} & \quad \# \text{jump to label and store } \text{PC}+4 \text{ in } \text{S31} \\
\text{jc } & \text{rt0} \quad \# \text{jump to address specified by register’s contents} \\
\text{jalr } & \text{rt0, ra} \quad \# \text{jump to address specified by register’s contents}
\end{align*}
\]

Formats:

\[
\begin{align*}
\text{J-type: used for } & \quad \text{OP} = 2 \quad \text{26-bit constant} \\
\text{J-type: used for } & \quad \text{OP} = 3 \quad \text{26-bit constant} \\
\text{R-type, used for } & \quad \text{OP} = 0 \quad \text{r_s} \quad 0 \quad 0 \quad 0 \quad \text{func} = 8 \\
\text{R-type, used for } & \quad \text{OP} = 0 \quad \text{r_s} \quad 0 \quad \text{r_d} \quad 0 \quad \text{func} = 9
\end{align*}
\]
Now we can do Factorial...

Synopsis (in C):
- Input in n, output in ans
- r1, r2 used for temporaries
- follows algorithm of our earlier data paths.

MIPS code, in assembly language:

```asm
n:       .word 123
ans:     .word 0

...  
addi $t0, 1, 0  # t0 = 1
lw   $t1, n    # t1 = n
loop:   beq $t1, 0, done    # while (t1 != 0)
mul $t0, $t0, $t1    # t0 = t0 * t1
addi $t1, $t1, -1  # t1 = t1 - 1
beq $0, 0, loop    # Always branch
done:   sw $t0, ans  # ans = r1
```

To summarize:

<table>
<thead>
<tr>
<th>MIPS operands</th>
<th>Name</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
</table>
| 32 registers   | $a0-$a3, $v0-$v1, $t0-$t9, $zero | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. | Each register is a 32-bit value.
|               | $fp, $sp, $ra, $at | Reserved for the assembler to handle large constants. | These registers are used for temporary storage and procedure calls.

<table>
<thead>
<tr>
<th>MIPS assembly language</th>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants.</td>
<td></td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw</td>
<td>lw $s1, 100($s2)</td>
<td>Memory[$s2 + 100]</td>
<td>Word from memory to register.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100]</td>
<td>Word from register to memory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lui</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2</td>
<td>Loads constant in upper 16 bits.</td>
<td></td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 100</td>
<td>Equal test; PC-relative branch.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 100</td>
<td>Not equal test; PC-relative branch.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>slti</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>jal 2500 $ra</td>
<td>go to 10000</td>
<td>For procedure call.</td>
<td></td>
</tr>
</tbody>
</table>

Marc Pollefeys 2/24/2005 00:36
Summary

• We will use a subset of MIPS instruction set as a prototype
  • Fixed-size 32-bit instructions
  • Mix of three basic instruction formats
    • R-type - Mostly 2 source and 1 destination register
    • I-type - 1-source, a small (16-bit) constant, and
      a destination register
    • J-type - A large (26-bit) constant used for jumps
  • Load/Store architecture
  • 31 general purpose registers, one hardwired to 0, and, by
    convention, several are used for specific purposes.
• ISA design requires tradeoffs, usually based on
  • History
  • Art
  • Engineering
  • Benchmark results