Introduction to the MIPS ISA

Overview

• The MIPS ISA specifies a collection of very simple instructions (not unlike the SimpleISA instructions -- there are just more of them.)
• Terms:
  • High-level language: eg. C, C++, Java
  • Assembly language: textual representation of machine language
  • Machine language: just the ones and zeros understood by the machine

Overview: Tools

• A compiler’s job is to take a source file in a high-level language and turn it into assembly code:
  \[ \text{cc foo.c} \rightarrow \text{foo.s} \]
• An assembler’s job is to take an assembly file and turn it into machine code (object file)
  \[ \text{asm foo.s} \rightarrow \text{foo.o} \]
• A Linker’s job is to take a bunch of object files and "merge" them into a single executable:
  \[ \text{linker foo.o libc.o etc.o} \rightarrow \text{a.out} \]

A Running Example

• Here is a simple C program:
  ```c
  int array[100];
  void main() {
    int i;
    while (i < 100) {
      array[i] = i;
      i = i + 1;
    }
  }
  ```
• What instructions should the ISA include to execute it?
• Tensions: compiler quality, memory size, ease of programming, hardware design complexity

The MIPS Family

• MIPS originated from a project at Stanford Univ: Microprocessor without Interlocked Pipe Stages
• H+P posit 4 principles of design. Keep them in mind:
  • Simplicity favors regularity
  • Smaller is faster
  • Compromise
  • Make the common case fast

MIPS is a RISC

• RISC = Reduced (Regular/Limited) Instruction Set Computer
• For instance, (almost) all arithmetic operations are of the form:
  \[ \text{Rdest} = \text{op} \times \text{R1} \]
• Another restriction: MIPS is a load/store architecture (like SimpleISA).
• Another restriction: all instructions are 32-bits long
• Basic families of operations:
  • Arithmetic (add, subtract, etc)
  • Logical (or, and)
  • Control (branches and jumps)
  • Memory access (load and store)
Load-Store Architecture

• Every operand must be in a register (with a few exceptions)
• Variables must be loaded into registers
• Results must be loaded back
• Example C code...
  
  ```c
  a = b + c;
  d = a + b;
  ```

  ...would be translated into something like:
  
  ```
  load b into register Rx
  load c into register Ry
  Rz <- Rx + Ry
  store Rz into a
  ... etc
  ```

MIPS Registers

• Provides 32, 32-bit registers, for:
  • integer arithmetic
  • address calculations
  • special functions (later)
  • temporary values
  • A 32-bit program counter (PC)
  • Two 32-bit registers (HI and LO) used for multiplication & division
  • Floating point registers (later)

Register Names and Conventions

<table>
<thead>
<tr>
<th>Register #</th>
<th>Name</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td></td>
<td>Always 0</td>
<td>Can't write it!</td>
</tr>
<tr>
<td>$1</td>
<td>at</td>
<td>Reserved for assembler don't use it!</td>
<td></td>
</tr>
<tr>
<td>$2-$3</td>
<td>v0-v1</td>
<td>Function return value</td>
<td></td>
</tr>
<tr>
<td>$4-$7</td>
<td>a0-a3</td>
<td>Function call parameters</td>
<td></td>
</tr>
<tr>
<td>$8-$15</td>
<td>t0-t7</td>
<td>Volatile temporaries Not saved on call</td>
<td></td>
</tr>
<tr>
<td>$16-$23</td>
<td>s0-s7</td>
<td>Saved temporaries Saved on call</td>
<td></td>
</tr>
<tr>
<td>$24-$25</td>
<td>t8-t9</td>
<td>Volatile temporaries</td>
<td></td>
</tr>
<tr>
<td>$26-$27</td>
<td>k0-k1</td>
<td>Reserved for kernel/OS Don't use them!</td>
<td></td>
</tr>
<tr>
<td>$28</td>
<td>gp</td>
<td>Global pointer</td>
<td></td>
</tr>
<tr>
<td>$29</td>
<td>sp</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>$30</td>
<td>fp</td>
<td>Frame pointer</td>
<td></td>
</tr>
<tr>
<td>$31</td>
<td>ra</td>
<td>Return address</td>
<td></td>
</tr>
</tbody>
</table>

MIPS Information Units

• Data types and size:
  • Byte (8 bits)
  • Half-word (2 bytes)
  • Word (4 bytes)
  • Float (4 bytes)
  • Double (8 bytes)

• Memory is byte addressable

• A data type must start on an address divisible by its size.

MIPS Instruction Types

• Remember these instruction classes:
  • Memory access (load/store)
  • Arithmetic/Logical (add, and, or, sub, etc)
  • Comparison (less-than)
  • Control (branches and jumps)

• We’ll use this notation when describing instructions:

  ```
  rd: destination register (modified by instruction)
  rs: source registers (read by instruction)
  rt: source/destination register (read or read & modified)
  immed: 16 bit immediate value encoded in instruction
  ```

Running Example

• Let’s translate this example into MIPS assembly:

  ```
  int x, y;
  void main() {
    x = x + y;
    if (x == y) {
      x = x + 3;
    }
    x = 42 + x * y;
    ...
  }
  ```
Loading and Storing

- Data is moved explicitly from memory to registers
- Each load/store must specify the address of the memory data to be read/written
- A MIPS address is just a 32-bit, unsigned integer
- Loads/Stores always use a base register (that holds an address) together with a 16-bit signed offset.

Load/Store Examples

- Load a word from memory:
  ```
  lw rt, offset(rs) # regs[rt] = memory[regs[rs] + offset]
  ```
- Store a word to memory:
  ```
  sw rt, offset(rs) # memory[regs[rs] + offset] = regs[rt]
  ```
- Real examples:
  ```
  lw $t6, 4($gp)
  sw $t3, -16($fp)
  ```

Arithmetic Instructions

- Two basic forms:
  ```
  OP rd, rs, rt
  OPI rt, rs, immed
  ```
- Examples:
  ```
  ADD $t3, $t3, $t5
  ADDI $t4, $sp, 4
  SUB $t1, $0, $a0
  ```
- Instructions to know and love:
  • ADD, SUB, ADDI

Multiplication & Division

- These are "special". Multiplying two 32 bit numbers can yield a result larger than 32 bits, hence:
- MULTI/DIV use the HI and LO registers for their results:
  ```
  MULT rs, rt # HI/LO <- rs * rt
  DIV rs, rt # LO <- rs/rt
  HI <- rs rem rt
  ```
- Talking to the HI/LO registers:
  ```
  MFHI rd # rd <- HI
  MTHI rs # HI <- rs
  MFLO rd # rd <- LO
  MTLO rs # LO <- rs
  ```

Control Flow: Branches

- MIPS lets us compare on...
  • equality or inequality of two registers (=== or !==)
  • comparison of a register to zero (>, <, >=, <=)
  • ... and then branch to a target that is a signed displacement (expressed in the number of words) from the instruction following the branch.

Branches (2)

- Here are examples of the main branch instructions:
  ```
  BEQ $t0, $t3, 12
  BNE $t1, $t4, -132
  BEQZ $t7, -200
  BNEZ $t7, 12
  BLTZ $s0, 24
  BLTZ $s1, 2
  ```
Comparing 2 Registers

What if you want to branch if register 6 is greater than register 7? Use the SLT instruction:

```
SLT $t0, $6, $7  # $t0 <- 1 if regs[$6] < regs[$7]
# else $t0 <- 0
```

Jump Instructions

Jump instructions allow for unconditional control flow change:

```
J target  # PC <- target
JR rs     # PC <- regs[rs]
JAL target # regs[32] <- PC; PC <- target
```

Examples:

```
J 100
JR $t4
JR $ra
JAL 440
```

JAL is used to implement procedure call...

Logic Instructions

Used to manipulate bits within words.

Have the same form as arithmetic instructions:

```
OP rd, rs, rt
OPI rt, rs, immed
```

- OP can be: AND, OR, XOR.
- Examples:
  - ORI $6, $6, 0x00FF
  - ORI $7, $0, 0xFF00
  - AND $8, $8, $7

Shift Instructions

Used to move bits around within registers.

Logical shifts (zeros are shifted in from the end):

```
SLL rd, rt, immed  # regs[rd] = regs[rt] << immed
SLLV rd, rt, rs    # regs[rd] = regs[rt] << regs[rs]
SRL rd, rt, immed  # regs[rd] = regs[rt] >>> immed
SRLV rd, rt, rs    # regs[rd] = regs[rt] >>> regs[rs]
```

Arithmetic shift (sign extend from left bit):

```
SRA rd, rt, immed  
SRAV rd, rt, rs
```

Back to Our Example

We’ll put x into location 0($gp) and y into location 4($gp)

Here’s the assembly code:

```
lw $t1, 0($gp)  # t1 holds x
lw $t2, 4($gp)  # t2 holds y
add $t1, $t1, $t2  # x = x + y
bne $t1, $t2, 2  # branch if t1 != t2
add $t1, $t1, 3  # x = x + 3
sw $t1, 0($gp)  # update x
```

Discussion

We’re going to great lengths to preserve the original semantics of the C program.

- We store the values back to their memory locations after computing them.
- Why might this be a good idea?
- A bad idea?
- Rewrite the previous example and eliminate as many of the loads/stores as is reasonable.
An Optimized Example

We eliminate unnecessary loads and stores...

lw $t1, 0(gp)  # t1 holds x
lw $t2, 4(gp)  # t2 holds y
add $t1, $t1, $t2  # x = x + y
bne $t1, $t2, 2  # branch if t1 != t2
addi $t1, $t1, 3  # x = x + 3
mult $t1, $t2  # lo = x * y
mflo $t3  # get the result
add $t1, $t3, 42
lw $t1, 0(gp)  # update x