Introduction to the MIPS ISA

Overview

- Remember that the machine only understands very basic instructions (machine instructions)
- It is the compiler’s job to translate your high-level (e.g. C program) into machine instructions. In more detail (forgetting linking):

  - Source program (foo.c)
  - Compiler (cc -S foo.c)
  - Assembly program (foo.s)
  - Assembler (cc foo.s)
  - Executable program (a.out)

  - Assembly language is a thin veneer over machine language.

Overview (2)

- Think about a simple C program...
  ```c
  int array[100];
  void main () {
    int i;
    for (i=0; i<100; i++)
      array[i] = i;
  }
  ```
- What set of instructions (ISA) should the machine provide to execute it?
- What does your intuition tell you about trade-offs between the ISA and the size (length) of the resulting machine program?
- What kind of trade-offs exist between the ISA and the speed, cost, complexity of the hardware needed to execute the program?
- Tensions and contributing factors: ease of programming, ease of hardware design, program/memory size, compiler technology

MIPS ISA Overview

- MIPS is a “computer family”: R2000/3000 (32-bit), R4000/4400 (64-bit)
- New entries include R8000 (scientific/graphics) and R10000
- MIPS originated as a Stanford project: Microprocessor without Interlocked Pipe Stages
- H+P posit 4 principles of hardware design. Try to keep them in mind during our discussion of the MIPS ISA:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Compromise
  4. Make the common case fast
**MIPS is a RISC**

- RISC = Reduced (Regular/Restricted) Instruction Set Computer
- All arithmetic operations are of the form:
  \[ R_d \leftarrow R_s \text{ op } R_t \]  # the Rs are registers
- Important restriction: MIPS is a load store architecture: the ALU can only operate on registers. Why?
- Basic operations (really only a few kinds)
  1. Arithmetic (addition, substraction, etc)
  2. Logical (and, or, xor, etc)
  3. Comparison (less-than, greater-than, etc)
  4. Control (branches, jumps, etc)
  5. Memory access (load and store)
- All MIPS instructions are 32 bits long

**MIPS Registers**

- Provides thirty-two, 32-bit registers, named $0, $1, $2 .. $31 used for:
  - integer arithmetic
  - address calculations
  - special-purpose functions defined by convention
  - temporaries
- A 32-bit program counter (PC)
- Two 32-bit registers HI and LO used specifically for multiplication and division
- Thirty-two 32-bit registers $f0, f1, f2 .. f31 used for floating point arithmetic
- Other special-purpose registers (see later)
### MIPS Register Names and Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>zero</td>
<td>Always 0</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>Sat</td>
<td>reserved for assembler</td>
<td></td>
</tr>
<tr>
<td>S2-3</td>
<td></td>
<td>S0-O+1</td>
<td></td>
</tr>
<tr>
<td>S4-7</td>
<td>a3</td>
<td>expression eval./function return</td>
<td></td>
</tr>
<tr>
<td>S8-15</td>
<td>t7</td>
<td>volatile temporaries</td>
<td></td>
</tr>
<tr>
<td>S16-23</td>
<td>t7</td>
<td>temporaries (saved across calls)</td>
<td></td>
</tr>
<tr>
<td>S24-27</td>
<td>t9</td>
<td>volatile temporaries</td>
<td></td>
</tr>
<tr>
<td>S28</td>
<td></td>
<td>reserved kernel/OS</td>
<td></td>
</tr>
<tr>
<td>S29</td>
<td>Sop</td>
<td>stack pointer</td>
<td></td>
</tr>
<tr>
<td>S30</td>
<td>Sfp</td>
<td>frame pointer</td>
<td></td>
</tr>
<tr>
<td>S31</td>
<td>Sra</td>
<td>proc/funct return address</td>
<td></td>
</tr>
</tbody>
</table>

### MIPS Information Units

- Data types and size:
  - Byte
  - Half-word (2 bytes)
  - Word (4 bytes)
  - Float (4 bytes, single precision format)
  - Double (8 bytes, double precision format)
- Memory is byte addressable.
- A data type must start on an address divisible by its size (in bytes)
- The address of the data type is the address of its lowest byte (MIPS on DEC is little endian)

### MIPS Addressing

- In MIPS (and most byte addressable machines) every word should start at an address divisible by 4.
- Why?

### MIPS Instruction Types

- As we said earlier, there are very few basic operations:
  1. Memory access (load and store)
  2. Arithmetic (addition, subtraction, etc)
  3. Logical (and, or, xnor, etc)
  4. Comparison (less-than, greater-than, etc)
  5. Control (branches, jumps, etc)
- We'll use the following notation when describing instructions:
  - rd: destination register (modified by instruction)
  - rs: source register (read by instruction)
  - rt: source/destination register (read or read-modified)
  - immed: a 16-bit value
Running Example

Let's translate this simple C program into MIPS assembly code:

```c
int x, y;

void main() {
    ...
    x = x + y;
    if (x==y) {
        x = x + 3;
    }
    x = x + y + 42;
    ...
}
```

Load and Store Instructions

- Data is explicitly moved between memory and registers through load and store instructions.
- Each load or store must specify the memory address of the memory data to be read or written.
- Think of a MIPS address as a 32-bit, unsigned integer.
- Because a MIPS instruction is always 32 bits long, the address must be specified in a more compact way.
- We always use a base register to address memory
- The base register points somewhere in memory, and the instruction specifies the register number, and a 16-bit, signed offset
- A single base register can be used to access any byte within ??? bytes from where it points in memory.

Load and Store Examples

- Load a word from memory:
  ```
  lw rt, offset(base)  # rt <- memory[base+offset]
  ```
- Store a word into memory:
  ```
  sw rt, offset(base)  # memory[base+offset] <- rt
  ```
- For smaller units (bytes, half-words) only the lower bits of a register are accessible. Also, for loads, you need to specify whether to sign or zero extend the data.
  ```
  lb rt, offset(base)  # rt <- sign-extended byte
  lbu rt, offset(base)  # rt <- zero-extended byte
  sb rt, offset(base)  # store low order byte of rt
  ```

Arithmetic Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>rd, rs, rt</td>
<td># rd &lt;- rs + rt</td>
</tr>
<tr>
<td>ADDI</td>
<td>rt, rs, immed</td>
<td># rt &lt;- rs + immed</td>
</tr>
<tr>
<td>SUB</td>
<td>rd, rs, rt</td>
<td># rd &lt;- rs - rt</td>
</tr>
</tbody>
</table>

Examples:

- ADD $8, $8, $10  # r8 <- r9 + r10
- ADD $t0, $t1, $t2  # t0 <- t1 + t2
- SUB $s0, $s0, $s1  # s0 <- s0 - s1
- ADDI $t3, $t4, 5  # t3 <- t4 + 5
Multiply and Divide Instructions

- Multiplying two 32-bit numbers can yield a 64 bit number. Hence the use of HI and LO registers.
- Dividing two numbers yields a quotient and a remainder.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT</td>
<td>rs, rt</td>
<td># HI/LO &lt;- rs * rt</td>
</tr>
<tr>
<td>MULTU</td>
<td>rs, rt</td>
<td># HI/LO &lt;- rs * rt</td>
</tr>
<tr>
<td>DIV</td>
<td>rs, rt</td>
<td># LO &lt;- rs/rt</td>
</tr>
<tr>
<td>DIVU</td>
<td>rs, rt</td>
<td># HI &lt;- rs rem rt</td>
</tr>
</tbody>
</table>

- If an operand is negative, the remainder is not specified by the MIPS architecture.

Multiply and Divide Instructions (2)

- There are instructions to move between HI/LO registers.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFHI</td>
<td>rd</td>
<td># rd &lt;- HI</td>
</tr>
<tr>
<td>MTHI</td>
<td>rs</td>
<td># HI &lt;- rs</td>
</tr>
<tr>
<td>MFLO</td>
<td>rd</td>
<td># rd &lt;- LO</td>
</tr>
<tr>
<td>MTLO</td>
<td>rs</td>
<td># LO &lt;- rs</td>
</tr>
</tbody>
</table>

Integer Arithmetic

- Numbers can be either signed or unsigned
- The above instructions all check for, and signal overflow should it occur.
- MIPS ISA provides instructions that don’t care about overflows:
  - ADDU
  - ADDIU
  - SUBU, etc.
- For add and subtract, the computation is the same for both, but the machine will signal an overflow when one occurs for signed numbers.

Overflows in 2’s Complement

- Overflow occurs when the addition of two numbers of the same sign results in a sum of the opposite sign
- Overflow cannot occur when adding operands of different signs

- Example 1: Assume a 4-bit machine. Register 9 contains 7 and register 10 contains 3
  - What happens when we use ADD? ADDU?

- Example 2: Assume a 4-bit machine. Register 9 contains 7 and register 10 contains -3
  - What happens when we use ADD? ADDU?
Flow of Control: Conditional Branches

- You can compare on...
  - equality or inequality of two registers
  - comparison of register to zero (>, <, <=, >=)
  - ... and branch to a target that is a signed displacement (expressed in number of instructions [words not bytes!]) from the instruction following the branch.

Branches (2)

- In assembly language, it’s easiest to just use the target address (from the label), rather than trying to figure out the number of instructions.
  - `BEQ rs, rt, target` # branch if rs == rt
  - `BNE rs, rt, target` # branch if rs != rt
  - `BGTZ rs, target` # branch if rs > 0
  - `BGEZ rs, target` # branch if rs >= 0
  - `BLTZ rs, target` # branch if rs < 0
  - `BLEZ rs, target` # branch if rs <= 0

Comparison Between Registers

- What if you want to branch if R6 is greater than R7?
- We can use the SLT instruction:
  ```
  SLT rd, rs, rt # if rs<rt then rd <- 1
                  # else rd <- 0
  SLTU rd, rs, rt # same, but rs,rt unsigned
  ```
- Example: Branch to L1 if $5 > $6
  ```
  SLT $7, $6, $5 # $7 = 1, if $6 < $5
  BNE $7, $0, L1
  ```

Jump Instructions

- Jump instructions allow for unconditional transfer of control:
  ```
  J target      # go to specified target
  JR rs         # jump to addr stored in rs
  ```
- Jump and link is used for procedure calls:
  ```
  JAL target    # jump to target, $31 <- PC
  JALR rs, rd   # jump to addr in rs
                  # rd <- PC
  ```
- When calling a procedure, use JAL; to return, use JR $31
Logic Instructions

- Used to manipulate bits within words, set up masks, etc.

<table>
<thead>
<tr>
<th></th>
<th>Opcode Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>rd, rs, rt</td>
<td># rd &lt;- AND(rs, rt)</td>
</tr>
<tr>
<td>ANDI</td>
<td>rt, rs, immed</td>
<td># rt &lt;- AND(rs, immed)</td>
</tr>
<tr>
<td>OR</td>
<td>rd, rs, rt</td>
<td></td>
</tr>
<tr>
<td>ORI</td>
<td>rt, rs, immed</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>rd, rs, rt</td>
<td></td>
</tr>
<tr>
<td>XORI</td>
<td>rt, rs, immed</td>
<td></td>
</tr>
</tbody>
</table>

- The immediate constant is limited to 16 bits
- To load a constant in the 16 upper bits of a register we use LUI:

<table>
<thead>
<tr>
<th></th>
<th>Opcode Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>rt, immed</td>
<td># rt&lt;31,16&gt; &lt;- immed</td>
</tr>
<tr>
<td></td>
<td># rt&lt;15,0&gt; &lt;- 0</td>
<td></td>
</tr>
</tbody>
</table>

Logic Instruction Examples

1. Turn on the bits in the low order byte of R6:
   ORI $6, $6, 0x00ff # set r6<7,0> to 1s

2. Turn off the bits in the low order byte of R6:
   LUI $5, 0xffff # set r5<31,16> to 1s
   ORI $5, 0xff00 # zero low order byte
   AND $6, $6, $5 # zap low order byte in R6

3. Flip the 16 bits in the high order byte of R6: (check this one)
   LUI $5, 0xff00 # 1s in upper byte
   ANDI $5, 0x0000 # 0s everywhere else
   XOR $6, $6, $5 # flip upper bits...

Shift Instructions

- Used to move bits around within registers.
- Logical shifts (zeros are shifted in from end).

<table>
<thead>
<tr>
<th></th>
<th>Opcode Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>rd, rt, shamt</td>
<td># rd = rt shifted left by # shamt</td>
</tr>
<tr>
<td>SRL</td>
<td>rd, rt, shamt</td>
<td># right shift</td>
</tr>
</tbody>
</table>

- Arithmetic shift right (sign extend from left)

| SRA | rd, rt, shamt | # rd = rt shifted right by # shamt, and sign extended |

- shamt is a 5-bit shift amount

Back to our example

```assembly
.data # start of data segment
x: .word # data layout directive
y: .word # allocate two words
text # start of text segment
.globl main
main:
   la $t0, x # t0 holds &x
   lw $t1, 0($t0) # t1 holds x
   la $t2, y # t2 holds &y
   lw $t3, 0($t2) # t3 holds y
   add $t1, $t1, $t3 # t1 = t1 + t3
   j main: # if x == y
   addi $t4, $t1, 17 # t4 = y + 17
   lw $t1, 0($t0)
   lw $t2, 0($t0)
   add $t1, $t1, $t2 # if x == y
   sw $t1, 0($t0)
   sw $t1, 0($t0)
L1: addi $t4, $t3, 17 # t4 = y + 17
   add $t1, $t1, $t4
   sw $t1, 0($t0)
```

CSE378 WINTER, 2001
Discussion

- Note that we're going to great lengths to preserve the semantics of the original C program.
- We're storing back values to their memory locations immediately after computing them.
- Why might this be a good idea?
- Why might this be a bad idea?

An optimized example

- We eliminated “unnecessary” stores.

```
.data # start of data segment
x: .word # data layout directive
y: .word # allocate two words
text # start of text segment
globl main
main:
la $t0, x # t0 holds &x
lw $t1, 0($t0) # t1 holds x
la $t2, y # t2 holds &y
lw $t3, 0($t2) # t3 holds y
add $t1, $t1, $t3 # x = x+y
bne $t1, $t3, L1 # if x == y
add $t1, $t1, 3 # x = x+3
L1: addi $t4, $t3, 17 # t4 = y + 17
add $t1, $t1, $t4
sw $t1, 0($t0)
```

Example C Program

```
#include <stdio.h>
int array[100];

void main ()
{
    int i;
    for (i=0; i<100; i++)
        array[i] = i;
}
```

Assembly Version (Hand coded)

```
data # begin data segment
array: .space 400 # allocate 400 bytes
text # begin code segment
globl main # entry point must be global
main:
move $t0, $0 # $t0 is used as counter
la $t1, array # $t1 is pointer into array
start: bge $t0, 100, exit # more than 99 iterations?
sw $t0, 0($t1) # store zero into array
addi $t0, $t0, 1 # increment counter
addi $t1, $t1, 4 # increment pointer into array
j start # goto top of loop
exit: j $ra # return to caller of main...
```
Assembly Version (Compiler Generated)

.data
array: .space 400 # the comments are obviously
.globl main # NOT generated by the compiler!

.text
.globl main
main:
    subu $sp, 8 # make room on stack
    lw $0, 4($sp) # i lives at 4($sp)...
    $32: # ...initialize it to zero
    lw $14, 4($sp) # load i into $14
    mul $15, $14, 4 # $15 is used as base reg
    sw $14, array($15) # store i into array[i]
    lw $24, 4($sp) # load i into $24
    addu $25, $24, 1 # increment $24
    sw $25, 4($sp) # store new val into i
    blt $25, 100, $32 # if i<100 goto top
    move $2, $0 # set $2 to 0
    addu $sp, 8 # set stack
.end main