EE 361

MIPS Introduction (MIPSa Notes)

ROADMAP: Our Computer Architecture

• MIPS overview
• Basic instruction overview
  – Operations
  – Operands (data) and memory organization
  – Format
• C language and machine language
  – Some basic C language instructions
  – How to implement them with machine instructions
• Design principles will be highlighted
MIPS Motivation

RISC Computer Architecture

Reduced Instruction Set Computer

Instructions are simple, means implementation can run fast

Instructions are carefully chosen, from experience and experiments
    The most commonly used ones were chosen

Instructions are structured so that they can be executed quickly
    Uniformity, avoid special cases/non-uniformity

We will first study Instruction Set Architecture of MIPS

Actually, we’ll focus on a subset of the instructions that does integer arithmetic

Instruction Components

Operations: What instructions do

Types
Arithmetic
Data Transfer
Program Control

Examples
add two numbers, subtract two numbers, etc.
copy data from a memory loc. to another loc.
if-else

Operands: Data

Examples: numbers, text, pointers

We’ll review elements of the C language and show how to implement it with assembly/machine language and introduce instruction concepts
Review of C

Some basic elements of C:
- assignments: `i = j;`
- arithmetic: `k = j+n;`
- data types: `int` (integers), `float`, `double` (real numbers), `char` (characters)
- arrays: `A[ ]` such as integer arrays or character (text) strings
- `struct` (structures)

Arithmetic (Data Manipulation)

Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add a,b,c</td>
<td>a = b+c</td>
</tr>
<tr>
<td>subtract</td>
<td>sub a,b,c</td>
<td>a = b-c</td>
</tr>
</tbody>
</table>

Note: Each instruction has 3 operands. *Principle 1: Simplicity favors regularity*

Example of hand compiling

C Statement | An implementation in (informal) MIPS assembly language: (here, "#" means comment to the right.)
---|---
`f = (g+h) - (i+j)` | `add t0,g,h  # t0 = g+h, t0 is temp. var.  
add t1,i,j  # t1 = i+j, t1 is temp. var.  
sub f,t0,t1  # f = t0-t1`
Example

C statement | Implementation in MIPS assembly language
---|---
f = 2*(g-h) + i

Operand Basics

How do we represent data (operands)?

Sizes

<table>
<thead>
<tr>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>word (integers, real numbers, machine instructions), standard operand size for MIPs. Memory cell size</td>
</tr>
<tr>
<td>16 bits</td>
<td>halfword</td>
</tr>
<tr>
<td>8 bits</td>
<td>byte (char)</td>
</tr>
</tbody>
</table>

How and where do we store data? Memory cells and arrays of memory cells
Memory Organization

Two types of memory cells

CPU

32 registers $0$ - $31$
Each cell stores a 32-bit word
Very fast access

Principle: Smaller is faster
Simpler hardware
Fast hardware is expensive

Main Memory

Array of memory cell, each cell holds a 32-bit word
Each cell is specified by a 32-bit address (index)
BUT the address must be a multiple of 4.
There are $2^{30}$ memory cells
Stores operands and instructions

Byte-addressable

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


"Big Endian"

Example: hand compiling a C statement

C Statement: \[ f = (g+h) - (i+j) \]
add \( \text{temp1} = g+h \)
add \( \text{temp2} = i+j \)
sub \( f = \text{temp1-temp2} \)

Assembly Language Implementation:

Assign C vars to mem. locs: \( f: $18 \quad g: $19 \quad h: $20 \quad i: $21 \quad j: $22 \quad k: $23 \)

add $23,$19,$20 # $23 = g+h
add $24,$21,$22 # $24 = i+j
sub $18,$23,$24 # $f = $23-$24
Instruction Type: Data Transfer

lw and sw are instructions that copies words between memory and the registers in the CPU

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>load word</td>
<td>lw</td>
<td>$1 = \text{Memory}[2 + 100] $</td>
</tr>
<tr>
<td>store word</td>
<td>sw</td>
<td>\text{Memory}[, 2 + 100] = 1</td>
</tr>
</tbody>
</table>

Note

*“100” is a constant, which doesn’t have to be 100.
*“100” and the value in $2$ are used to compute an address

lw does a memory read
sw does a memory write

Example

C Statement:

\[
f = A[i];
\]

MIPS assembly language implementation:

\[
f: \text{\$18}\quad i: \text{\$19} \\
\text{Astart: “Label” = start of } A[ \]
\]

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Astart}</td>
<td>A[0]</td>
</tr>
<tr>
<td>\text{Astart} + 4</td>
<td>A[1]</td>
</tr>
<tr>
<td>\text{Astart} + 8</td>
<td>A[2]</td>
</tr>
</tbody>
</table>

\[
\text{add } \text{\$20,\$19,\$19}\quad # \text{\$20 = 4*i} \\
\text{add } \text{\$20,\$20,\$20}\ |
\text{lw } \text{\$18,Astart(\$20)}\quad # f = A[i], \text{ here } \text{\$20 is “index register”}
\]
Example

C statement: \[ A[h] = 2\times A[i]; \]

MIPS assembly language implementation: \[ h:\$18 \quad i:\$19 \]

Odds and Ends

**Spilling Registers:**

Best storage locations for C language variables are in the registers (because they're fast). But often there are more variables than 32. So variables can be moved between registers and main memory (compiler takes care of this).
Review

Review of Instructions so far:

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $1, $2, $3</td>
<td>$1 = $2 + $3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $1, $2, $3</td>
<td>$1 = $2 - $3</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>load word</td>
<td>lw $1,100($2)</td>
<td>$1 = Memory[100 + $2]</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $1,100($2)</td>
<td>Memory[100 + $2] = $1</td>
</tr>
</tbody>
</table>

Memory Cells

Main memory is an array of 32-bit cells, each containing 4 bytes
It stores machine instructions (programs) and operands (data)

Registers are 32-bit memory cells in the CPU. Registers are faster than main memory.

Design principles: Smaller is faster, and uniform means simpler and faster

Machine Instruction Format

We’ve got other types of instructions to go but we’ll take a detour to introduce

MACHINE INSTRUCTIONS:

The instructions that the machine understands

An instruction is a 32-bit word -- difficult for us to read

The machine instructions are of different formats.
format = instruction is organized into fields
field = tells something about the instruction

Example: Dates are sometimes formatted YYMMDD

870612
Example

add $8,$17,$18

<table>
<thead>
<tr>
<th>0</th>
<th>17</th>
<th>18</th>
<th>8</th>
<th>0</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

fields:

- **op**: operation of the instruction (opcode)
- **rs**: first register source operand
- **rt**: 2nd register source operand
- **rd**: destination register, where result is stored
- **shamt**: shift amount -- to be explained later
- **funct**: function -- works with op to determine instruction

Simplicity favors regularity

But “lw” and “sw” require two registers and one address-constant. E.g., lw $1,100($2)
## Example

### R Type

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0</td>
<td>1st src</td>
<td>2nd src</td>
<td>dst</td>
<td>0</td>
<td>32</td>
<td>NA</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0</td>
<td>“”</td>
<td>“”</td>
<td>“”</td>
<td>34</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>35</td>
<td>mem.</td>
<td>cpu</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>43</td>
<td>“”</td>
<td>“”</td>
<td>“”</td>
<td>“”</td>
<td>“”</td>
<td>“”</td>
</tr>
</tbody>
</table>

**Example:** What is machine code for `sub $3, $4, $5` in decimal notation?

**Example:** What about `sw $7, 2000($8)`?

## Odds and Ends

**Register $0:** special register that always equals zero

**Pseudo-instructions:**

These are instructions accepted by the assembler, but are not instructions of the ISA. Example:

- `move $15,$16`

  is actually implemented by

  - `add $15,$0,$16`

**How would you implement `move $15,$16` using the sub instruction?**
Review of C: Changing Program Flow

Programs are a list of instructions

We typically follow instructions going *down* the list

BUT may jump around --- and the jumping may depend on *decisions*

```
main()
{
  ...
  i = 20;
  j = k+30;
  if (n < m) {
    k = n;
  }
  j = j + k;
  ...
}
```

We'll select some C program fragments
Hand compile them and introduce instructions and concepts

Useful C Instructions

- **if** *(statement)* {
  }

- **else** {
  }

- **while** *(statement)* {
  }

- **for** *( , , )* {
  }

- **switch**(value) {
  case 0:  {      }  break;
  case 1:  {      }  break;
  }

*How can we implement them?*

Step 1. Implement them using goto and if
Step 2. Translate goto and if into machine or assembly language instructions
if and goto

We can implement many C instructions using goto and if

To jump: goto label

You're allowed to place labels in your program. goto tells you to jump to the label to execute the next instruction

goto is generally forbidden since it leads to bad programming style.

To make decisions:

```c
if (statement) {
    Bunch-of-Code A
}
```

if-else and while

if (statement) {
    Bunch-of-Code A
} else {
    Bunch-of-Code B
}

while (statement) {
    Bunch-of-Code
}

if (statement is false) goto Else
    Bunch-of-Code A
else:
    goto Skip
Else:
    Bunch-of-Code B
Skip:

Loop: if (statement is false) goto Skip
    Bunch-of-Code
    goto Loop
Skip:
for

for (statement-1, statement-2, statement-3) {
  Bunch-of-Code
}

How do we implement if and goto?

• Branch instructions (conditional)
  – beq : branch equal
  – bne : branch not equal
• Jump instruction (unconditional)
  – j: jump
  – jr: jump register
Branching

Conditional branch instructions: primitive if

beq $reg1,$reg2,Label branch if equal I Type (same one as before)
if ($reg1 == $reg2) goto Label
bne $reg1,$reg2,Label branch if not equal I Type
if ($reg1 != $reg2) goto Label

Example:
if (i == j) {
    i = 0;
    j = 0;
}

Jumping

Jumping (unconditional): goto Label
j Label jump J Type
jr $reg jump-register R Type

J Type: | op | address |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
</tr>
</tbody>
</table>

Comments
# i: $10  j: $11
bne $10,$11,Skip # if i!= j Skip
move $10,$0  # i = 0;
move $11,$0  # j = 0;
Skip:  # ":" suffix identifies # label
Implementing if

if (statement) {
    Block-of-Code
}

Find a temporary variable k
Evaluate statement
Set $k = 0$ if false
or $k \neq 0$ if true

if ($k \neq 0$) {
    Block-of-Code
}

Comments:

• Evaluating statement means arithmetic, perhaps lots of it

• Note that many evaluations are inequalities (e.g., if ($x < y$))

A useful instruction:

<table>
<thead>
<tr>
<th>slt</th>
<th>$8,19,20$</th>
<th>set-on-less-than</th>
<th>$8 = 1$ if $19 &lt; 20$</th>
<th>$0$ otherwise</th>
</tr>
</thead>
</table>

Example of slt

if ($x < y$) {
    # Let
    # $4 = x$
    # $5 = y$
    # $6$ be a temporary variable

    slt $6,4,5$  # $6 = 1$ if $x < y$
    beq $6,0,\text{Skip}$

    \text{Block-of-Code}
}

Skip:
Example

C language

/* Compute $m = \min(i,j)$ */
\[
\text{if } (i \geq j) \quad m = j; \\
\text{else} \quad m = i;
\]

Assembly Language

# m: $10$
# i: $11$
# j: $12$

\[
\text{slt}$1,$11,$12 \quad # \text{ $1 = 1$ if } i < j; 0, \text{otherwise}
\]
\[
\text{b__}$0,$1,Else \\
\text{move}$10,$12 \quad # \text{ $m = j$;}
\]
\[
\text{jExit}
\]

Else:
\[
\text{move}$10,$11
\]

Exit:

Comments on Design

Other CPUs have the following types of instructions

- \text{blt} \quad \text{branch if less than}
- \text{bgt} \quad \text{branch if greater than}

Why doesn't MIPS have these instructions?

\text{It's better to to have two fast instructions than one slow one.}

Regularity and simplicity is the theme of MIPS
An application of jr

Recall: jr $5
jump to the memory location pointed to by $5

switch(k) {
  case 0: f = i+j; break;
  case 1: f = g+h; break;
  case 2: f = g-h; break;
}

Calculate $5 = \text{where to jump to}
jr $5

Main Memory

Case0:
  f = i+j
  j Exit

Case1:
  f = g+h
  j Exit

Case2:
  f = g-h
  j Exit

Exit:

mult $8,$19,$20 multiplying $8 = \text{$19 \times $20}

Calculating where to jump

$6 = \text{JTable + 4 x k}$
$5 = \text{Memory[$6$]}
jr $5$

Main Memory

switch(k) {
  case 0: f = i+j; break;
  case 1: f = g+h; break;
  case 2: f = g-h; break;
}

JTable:

Case0: $f = i+j$
  j Exit

Case1: $f = g+h$
  j Exit

Case2: $f = g-h$
  j Exit

Exit:

JTable:

Jump table
Jump Tables

```c
switch(k) {
  case 0:  f = i+j;   break;
  case 1:  f = g+h; break;
  case 2:  f = g-h;  break;
}
```

# Assume $10 = 4
# $16.  g: $17.  h: $18.  i: $19.  j: $20.  k: $21
Switch:
mult $9,$10,$21  # $9 = 4*k
lw $8,JumpTable($9)  # $8 = JumpTable[k]
jr $8

<table>
<thead>
<tr>
<th>JumpTable:</th>
<th>Case0</th>
<th>JumpTable+4:</th>
<th>Case1</th>
<th>JumpTable+8:</th>
<th>Case2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case0:</td>
<td></td>
<td>Case1:</td>
<td></td>
<td>Case2:</td>
<td></td>
</tr>
<tr>
<td>add $16,$19,$20</td>
<td># f = i+j</td>
<td>add $16,$17,$18</td>
<td># f = g+h</td>
<td>sub $16,$17,$18</td>
<td># f = g-h</td>
</tr>
<tr>
<td>j</td>
<td>Exit</td>
<td>j</td>
<td>Exit</td>
<td>j</td>
<td>Exit</td>
</tr>
<tr>
<td>Exit:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Could be somewhere else in memory

Review

- Machine instruction formats: R, I, and J
  - Sometimes design compromises are necessary
  - You can find format information in the back cover of the textbook
- Realizing if-else branching, loops, and case statements using branching and jump instructions