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

<table>
<thead>
<tr>
<th>Types</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add two numbers, subtract two numbers, etc.</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>copy data from a memory loc. to another loc.</td>
</tr>
<tr>
<td>Program Control</td>
<td>if-else</td>
</tr>
</tbody>
</table>

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: 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

<table>
<thead>
<tr>
<th>C Statement</th>
<th>An implementation in (informal) MIPS assembly language:</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = (g+h) - (i+j)</td>
<td>add t0,g,h # t0 = g+h, t0 is temp. var.</td>
</tr>
<tr>
<td></td>
<td>add t1,i,j # t1 = i+j, t1 is temp. var.</td>
</tr>
<tr>
<td></td>
<td>sub f,t0,t1 # f = t0-t1</td>
</tr>
</tbody>
</table>

Galen Sasaki  University of Hawaii
Example

<table>
<thead>
<tr>
<th>C statement</th>
<th>Implementation in MIPS assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = 2*(g-h) + i</td>
<td></td>
</tr>
</tbody>
</table>

Operand Basics

How do we represent data (operands)?

Sizes
- 32 bits = word (integers, real numbers, machine instructions),
  standard operand size for MIPS. Memory cell size
- 16 bits = halfword
- 8 bits = byte (char)

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

Two types of memory cells

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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Memory[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Principle: Smaller is faster
Simpler hardware
Fast hardware is expensive

Example: hand compiling a C statement

C Statement: \( f = (g+h) - (i+j) \)

add   temp1 = g+h
add   temp2 = i+j
sub   f = 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

Galen Sasaki        University of Hawaii
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 $1,100($2)</td>
<td>$1 = Memory[$2 + 100]</td>
</tr>
<tr>
<td>store word</td>
<td>sw $1,100($2)</td>
<td>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:

\[ f: \$18 \quad i: \$19 \]

Astart: “Label” = start of \[A[ ]\]

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

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

C statement:  \[ A[h] = 2^*A[i]; \]
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 = \text{Memory}[100+$2]$</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $1$,$100$($2$)</td>
<td>\text{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

#### Instruction Format

Different Instruction Types: Good designs demand compromise.

**R Type:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**I Type:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Example:

- `lw $8,1000($21)`
- `sw $8,1000($21)`

<table>
<thead>
<tr>
<th>I type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 21 8</td>
<td>1000</td>
</tr>
</tbody>
</table>

rs = address to memory
rt = register in CPU
sw $8,1000($21) is the same (except different opcode)

Simplicity favors regularity

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

Galen Sasaki University of Hawaii

---

### 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>10000</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

Galen Sasaki University of Hawaii
Example

R Type  | op | rs | rt | rd | shamt | funct
--|---|---|---|---|------|------
add | R  | 0 | 1st src | 2nd src | dst | 0 | 32 | NA
sub | R  | 0 | “” | “” | “” | “” | 34 | NA
lw  | I  | 35 | mem. | cpu | NA | NA | NA | NA
sw  | I  | 43 | “” | “” | “” | “” | “” | “”

Example:  What is machine code for \texttt{sub $3,$4, $5} in decimal notation?

Example:  What about \texttt{sw $7,2000($8)}?

Odds and Ends

Register $0$: special register that always equals zero

Pseudo-instructions:

These are instructions accepted by the \textit{assembler}, but are not instructions of the ISA. Example:

\texttt{move} $15,$16

is actually implemented by

\texttt{add} $15,0,$16

How would you implement \texttt{move} $15,$16 using the sub instruction?
Logic Operations

• MIPS support for the C operations:
  – shift left:  \( m = n \ll 2 \)
  – shift right:  \( m = n \gg 3 \)
  – Bit-wise AND:  \( m = a \& b \)
  – Bit-wise OR:  \( m = a \mid b \)
  – Bit-wise NOT:  \( m = \sim n \)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X*Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ Y \]

\[ X \]

\[ \cdot \]

OR Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X+Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Bit-wise operations between two strings of bits of equal length
Pairs of bits from each bit position are AND’d or OR’d together

<table>
<thead>
<tr>
<th>Bit-wise AND</th>
<th>Bit-wise OR</th>
<th>Also referred to as</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X = 001011 )</td>
<td>( X = 001011 )</td>
<td>“bit-by-bit” AND and OR</td>
</tr>
<tr>
<td>( Y = 101101 )</td>
<td>( Y = 101101 )</td>
<td></td>
</tr>
<tr>
<td>( 001001 )</td>
<td>( 101111 )</td>
<td></td>
</tr>
</tbody>
</table>

Galen Sasaki  
University of Hawaii
Logic Operation Instructions

\texttt{and} \ $r1,$r2,$r3$ \\
Similar to add or sub but bit-wise AND operation \\
Of course, overflow doesn’t make sense here. \\
The format is R type \\

\texttt{andi} \ $r1,$r2,constant \\
Similar to addi or subi but bit-wise AND operation \\
The format is I type with a 16-bit constant. \\
The constant is \textit{zero-extended} to 32 bits, \\
which means the first 16 bits are set to zero. \\

or and \texttt{ori} \ are similar to \texttt{and} and \texttt{andi} except with a bit-wise OR. \\

There is also a \texttt{nor} operation too \\

\texttt{and} \ implements \ $m = a \ & \ b$ \\
\texttt{or} \ implements \ $m = a \ | \ b$ \\
\texttt{nor} \ implements \ $a = \sim b$ \\

---

Applications

\texttt{and} is used to clear bits \\

\begin{align*}
\text{Data} &= 00010011011 \\
\text{Mask} &= 10110110001 0 \Rightarrow \text{clear and 1} \Rightarrow \text{leave alone} \\
\text{Result} &= 00010010001
\end{align*}

Application: check if the value of a particular bit \\

\begin{align*}
\text{Data} &= 00010011011 \\
\text{Mask} &= 00000100000 \\
\text{Result} &= 000000b0000 \quad \text{Result} = 0 \text{ if and only if } b = 0
\end{align*}

\texttt{or} is used to set bits \\

\begin{align*}
\text{Data} &= 00010011011 \\
\text{Mask} &= 10110110001 1 \Rightarrow \text{set to 1 and 0} \Rightarrow \text{leave alone} \\
\text{Result} &= 10110111011
\end{align*}
Shifting

Logical shift
- Left
- Right

Rotate
- Left
- Right

Multiply and divide
Overflow and underflow are possible

Unsigned
- Multiply by 2
- Divide by 2

Signed
- Multiply by 2
- Divide by 2

Arithmetic shift

MIPS Instructions

- `sll $r1,$r2,constant`
  - constant = shift amount (shamt)
  - $r1 = $r2 << constant

- `slr $r1,$r2,constant`
  - constant = shift amount (shamt)
  - $r1 = $r2 << constant
Example

```
sll $10, $16, 4    # $10 = $16 << 4
sll $t2, $s0, 4    # $t2 = $s0 << 4
```

R Format

```
op    rs    rt    rd    shamt    funct
0      0     16    10    4       0
```

always 0

Indicates “sll”

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) {
}

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
    if (statement) {
    }

if-else and while

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

while (statement) {
  Bunch-of-Code
}

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;
}
```

```
#     i: $10  j: $11  ← Comments
bne   $10,$11,Skip        # if i!= j Skip
move  $10,50             # i = 0;
move  $11,50             # j = 0;
Skip:
```

# **":" suffix identifies
# label
Jumping

Jumping (unconditional):

goto Label

j Label jump J Type
jr $reg jump-register R Type

J Type: op address
6 26

Implementing if

if (statement) {
    Block-of-Code
}

Find a temporary variable k
Evaluate statement
Set k = 0 if false
or k != 0 if true

if (k != 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:

```
slt $8,$19,$20
```

set-on-less-than $8 = 1 if $19 < $20
otherwise 0

Galen Sasaki University of Hawaii
Example of slt

```c
if (x < y) {
    // Block-of-Code
}
```

# Let
#     $4 = x
#     $5 = y
#     $6 be a temporary variable

```
slt $6,$4,$5  # $6 = 1 if x < y
beq $6,$0,Skip
```

Block-of-Code

Skip:

Example

C language

```c
/* Compute m = min(i,j) */
if (i >= j) m = j;
else    m = i;
```

Assembly Language

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

slt $1,$11,$12  # $1 = 1 if i < j; 0, otherwise
b__ $0,$1,Else
move $10,$12    # m = j;
j Exit
Else:
    move $10,$11
Exit:
```

Galen Sasaki  University of Hawaii
Comments on Design

Other CPUs have the following types of instructions:

- `blt` : branch if less than
- `bgt` : branch if greater than

Why doesn’t MIPS have these instructions?

It’s better 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;
}
```

- **Case 0:**
  - `f = i+j`
  - `j Exit`

- **Case 1:**
  - `f = g+h`
  - `j Exit`

- **Case 2:**
  - `f = g-h`
  - `j Exit`

- **Exit:**
  - `jr $5`

**Main Memory**

- **Calculate $5:**
  - Where to jump to

- **$8 = $19*$20**

```
mult $8,$19,$20
```
Calculating where to jump

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

Jump Tables

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

Could be somewhere else in memory

# Assume $10 = 4
# f: $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

JumpTable:
    Case0:
    JumpTable+4:
    JumpTable+8:

Case0:
    add $16,$19,$20  # f = i+j
    j Exit

Case1:
    add $16,$17,$18  # f = g+h
    j Exit

Case2:
    sub $16,$17,$18  # f = g-h
    j Exit

Exit:
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