The MIPS Instruction Set Architecture

CPS 104
Lecture 5

Today's Lecture

Admin
• HW #2
Outline
• Review
• A specific ISA, we'll use it throughout semester
• Instruction categories
• Specific Instructions
Reading
Chapter 3, Appendix A

Review: Instruction Set Architecture

Five Primary Dimensions
• Operations add, sub, mul, ...
  How is it specified?
• Number of explicit operands (0, 1, 2, 3)
  Where besides memory?
• Operand Storage
  How is memory location specified?
• Type & Size of Operands
  byte, int, float, vector, ...
  How is it specified?
Other Aspects
• Successor instruction
  How is it specified?
• Conditions
  How are they determined?
• Encodings
  Fixed or variable? Wide?
• Parallelism

Review: Basic ISA Classes

Accumulator:
1 address add A
1+x address adds A

Stack:
0 address add

General Purpose Register:
2 address add A B
3 address add A B C

Load/Store:
3 address add Ra Rb Rc
load Ra Rb
store Ra Rb

Review: Design Space of ISA

Review: Making Instructions Machine Readable

Instructions are bits with well defined fields
• Like a floating point number has different fields
• Instruction Format
  establishes a mapping from “instruction” to binary values
  which bit positions correspond to which parts of the instruction
  (operation, operands, etc.)
A Program

```cpp
#include <iostream.h>

main()
{
    int *a = new int[100];
    int *p = a;
    int k;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }
    cout << "entry 3 = " << a[3] << endl;
}
```

LOAD / STORE ISA

• What must the computer do to accomplish the work of these instructions?

Example: \( a * b - (a+c*b) \) (assume in memory)

<table>
<thead>
<tr>
<th>cc</th>
<th>bits</th>
<th>Data</th>
<th>Text</th>
<th>reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cc file</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stored Program Computer

• Instructions: a fixed set of built-in operations
• Instructions and data are stored in the (same) computer memory
• Fetch-Execute Cycle
  ```
  while (!done)
  {
      fetch instruction
      execute instruction
  }
  ```

What Must be Specified?

• Instruction Format
  ◦ how do we tell what operation to perform?
• Location of operands and result
  ◦ where other than memory?
  ◦ how many explicit operands?
  ◦ which can or cannot be in memory?
• Data type and Size
• Operations
  ◦ what are supported
  ◦ Successor instruction
    ◦ jumps, conditions, branches
  ◦ fetch-decode-execute is implicit!

MIPS ISA Categories

• Arithmetic
  ◦ add, sub, mul, etc
• Logical
  ◦ and, or, shift
• Data Transfer
  ◦ load, store
  ◦ MIPS is LOAD/STORE architecture
• Conditional Branch
  ◦ implement if, for, while... statements
• Unconditional Jump
  ◦ support function call (procedure calls)

MIPS Instruction set Architecture

• 3-Address Load/Store Architecture.
• Register and Immediate addressing modes for operations.
• Immediate and Displacement addressing for Loads and Stores.
• Examples (Assembly Language):
  ```
  add $1, $2, $3 # $1 = $2 + $3
  addi $1, $1, 4 # $1 = $1 + 4
  lw $1, 100($2) # $1 = Memory[$2 + 100]
  sw $1, 100($2) # Memory[$2 + 100] = $1
  lui $1, 100 # $1 = 100 X 216
  addi $1, $3, 100 # $1 = $3 + 100
  ```
### MIPS Integer Registers

- Registers: fast memory, integral part of the CPU.
- Programmable storage: 2^32 bytes
- 31 x 32-bit GPRs (R0 = 0)
- 32 x 32-bit FP regs (paired DP)
- 32-bit HI, LO, PC

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>U</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td></td>
</tr>
<tr>
<td>lo</td>
<td></td>
</tr>
<tr>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

### MIPS Instruction Formats

**R-type: Register-Register**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>shmt</td>
<td>func</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**I-type: Register-Immediate**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>Rs</td>
<td>Rt</td>
<td>Immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**J-type: Jump/Call**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Terminology**
- **Op** = opcode
- **Rs, Rt, Rd** = register specifier

### Operands Addressing: Register Direct

**Example:** ADD $1, $2, $3  \# $1 = $2 + $3

<table>
<thead>
<tr>
<th>op</th>
<th>Rs</th>
<th>Rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>001010</td>
<td>00010</td>
<td>00001</td>
<td>000001000100</td>
</tr>
</tbody>
</table>

### Immediate: 16 bit value

**Operand Addressing:**
- Register Direct and Immediate

**Add Immediate Example**

```
addi $1, $2, 100  \# $1 = $2 + 100
```

<table>
<thead>
<tr>
<th>op</th>
<th>Rs</th>
<th>Rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>000011</td>
<td>00010</td>
<td>00001</td>
<td>0000000011100100</td>
</tr>
</tbody>
</table>

### Successor Instruction

```
main()
{
    int x, y, same;  // $0 == 0 always
    x = 43;         // addi $1, $0, 43
    y = 2;          // addi $2, $0, 2
    same = 0;       // addi $3, $0, 0
    if (x == y)     // execute only if x == y
        same = 1;   // addi $3, $0, 1
    ...}
```
The Program Counter (PC)

• Special register (PC) that points to instructions
• Contains memory address (like a pointer)
• Instruction fetch is
  \[ \text{inst} = \text{mem}[\text{pc}] \]
• To fetch next sequential instruction \( \text{PC} = \text{PC} + \) ?
  \( \text{Size of instruction?} \)

The Program Counter

\[
x = 43; \quad \text{// addi $1$, $0$, 43}
y = 2; \quad \text{// addi $2$, $0$, 2}
same = 0; \quad \text{// addi $3$, $0$, 0}
\quad \text{if} (x == y)
\quad \quad \text{same} = 1; \quad \text{// addi $3$, $0$, 1 execute if } x == y
\]

PC is always automatically incremented to next instruction

\[
0x10000 \quad addi \quad 0x10004 \quad addi \quad 0x10008 \quad addi \quad 0x1000c \quad addi
\]

Clearly, this is not correct
We cannot always execute both 0x10008 and 0x1000c

\[
0x10000 \quad addi \quad 0x10004 \quad addi \quad 0x10008 \quad addi \quad 0x1000c \quad addi
\]

Understand branches

Successor Instruction

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)  
        tsame = 1;  // only if a1 == a2
    return(tsame);
}
```

```c
main() {
    int x,y,same; // r0 == 0 always
    x = 43; // addi $1$, $0$, 43
    y = 2; // addi $2$, $0$, 2
    same = equal(x,y); // need to call function
    // other computation
    }```

The Program Counter

• Branches are limited to 16 bit immediate
• Big programs?
Jump and Link Example

JAL 1000  # PC<- 1000, $31<-PC+4

$31 set as side effect, used for returning, implicit operand

J-Type: <op> target

PC

31 26

Op  Target Address

000011 00 0000 0000 0000 0000 1110 1000

Jump Register Example

jr $31  # PC <- $31

R Type: <OP> rd, rs, rt

Op  Rs  Rt  Rd  shamt  func

000000 00010 10000 00001 00000 00100

Instructions for Procedure Call and Return

int equal(int a1, int a2) {
  int tsame;
  tsame = 0;
  if (a1 == a2)
    tsame = 1;
  return(tsame);
}

main() {
  int x,y,same;
  x = 43;
  y = 2;
  same = equal(x,y);
  // other computation
}

MIPS Arithmetic Instructions

Instruction  Example  Meaning  Comments
add  add $1,$2,$3  $1 = $2 + $3  3 operands
subtract  sub $1,$2,$3  $1 = $2 - $3  3 operands
add immediate  addi $1,$2,100  $1 = $2 + 100 + constant
add unsigned  addu $1,$2,$3  $1 = $2 + $3  3 operands
subtract unsigned  subu $1,$2,$3  $1 = $2 - $3  3 operands
add imm. unsigned  addiu $1,$2,100  $1 = $2 + 100 + constant
multiply  mult $2,$3  Hi, Lo = $2 x $3  64-bit signed product
multiply unsigned  multu $2,$3  Hi, Lo = $2 x $3  64-bit unsigned product
divide  div $2,$3  Lo = $2 ÷ $3, Hi = quotient, Remainder
divide unsigned  divu $2,$3  Lo = $2 ÷ $3, Hi = quotient, Remainder
Move from Hi  mfhi $1  $1 = Hi  Used to get copy of Hi
Move from Lo  mflo $1  $1 = Lo  Used to get copy of Lo

Which add for address arithmetic? Which for integers?

MIPS Logical Instructions

Instruction  Example  Meaning  Comment
and  and $1,$2,$3  $1 = $2 & $3  Bitwise AND
or  or $1,$2,$3  $1 = $2 | $3  Bitwise OR
xor  xor $1,$2,$3  $1 = $2 ⊕ $3  Bitwise XOR
nor  nor $1,$2,$3  $1 = ~($2 | $3)  Bitwise NOR
and immediate  andi $1,$2,10  $1 = $2 & 10  Bitwise AND reg, const
or immediate  or $1,$2,10  $1 = $2 | 10  Bitwise OR reg, const
xor immediate  xor $1,$2,10  $1 = $2 ⊕ 10  Bitwise XOR reg, const
shift left logical  sll $1,$2,10  $1 = $2 << 10  Shift left by constant
shift right logical  srl $1,$2,10  $1 = $2 >> 10  Shift right by constant
shift right arithm.  sra $1,$2,10  $1 = $2 >> 10  Shift right (sign extend)
shift right logical  srl $1,$2,3  $1 = $2 >> 3  Shift right by var
shift right arithm.  sraw $1,$2,3  $1 = $2 >> 3  Shift right arith. by var

Why do we need LUI?

LUI  RS

0000 ... 0000

MIPS Data Transfer Instructions

Instruction  Comment
SW  R3, 500(R4)  Store word
SH  R3, 502(R2)  Store halfword
SB  R2, 41(R3)  Store byte
LW  R1, 30(R2)  Load word
LH  R1, 40(R3)  Load halfword
LBU  R1, 40(R3)  Load byte
LH  R1, 40(R3)  Load byte unsigned
LB  R1, 40(R3)  Load byte unsigned
LUI  R1, 40  Load Upper Immediate (16 bits shifted left by 16)

Why do we need LUI?
MIPS Compare and Branch

**Compare and Branch**

- **beq** `rs, rt, offset` if `R[rs] == R[rt]` then PC-relative branch
- **bne** `rs, rt, offset` <>

**Compare to zero and Branch**

- **blez** `rs, offset` if `R[rs] <= 0` then PC-relative branch
- **bgtz** `rs, offset` >
- **bgez** `rs, offset` >=
- **bltz** `rs, offset` <
- **bgezal** `rs, offset` if `R[rs] < 0` then branch and link (into R 31)

• Remaining set of compare and branch take two instructions
• Almost all comparisons are against zero!

---

MIPS jump, branch, compare instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>Equal test; PC relative branch</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>Not equal test; PC relative</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,1</td>
<td>Compare less than, 2's comp.</td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>Compare less than; 2's comp.</td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltiu $1,$2,100</td>
<td>Compare less than; natural numbers</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>

---

Signed v.s. Unsigned Comparison

- `R1= 00 0000 0000 0000 0001`
- `R2= 00 0000 0000 0000 0010`
- `R3= 11 1111 1111 1111 1111`

• After executing these instructions:
  - `slt r4,r2,z1`
  - `slt r5,r2,z1`
  - `sltu r6,r2,z1`
  - `sltu r7,r2,z1`

• What are values of registers `r4` - `r7`? Why?

---

Summary

- MIPS has 5 categories of instructions
  - Arithmetic, Logical, Data Transfer, Conditional Branch, Unconditional Jump
- 3 Instruction Formats

Next Time

- Assembly Programming

Reading

- Ch. 3, Appendix A