The MIPS Instruction Set Architecture

CPS 104
Lecture 5

Review: Basic ISA Classes

Accumulator:
1 address add A acc ← acc + mem[A]
1+x address addx A acc ← acc + mem[A + x]

Stack:
0 address add tos ← tos + next (JAVA VM)

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

Load/Store:
3 address add Ra Rb Rc Ra ← Rb + Rc
load Ra Rb Ra ← mem[Rb]
store Ra Rb mem[Rb] ← Ra

Review: LOAD / STORE ISA

• Instruction set:
  add, sub, mult, div, ... only on operands in registers
  ld, st, to move data from and to memory, only way
to access memory

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

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>r2</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>mult</td>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>r3</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>add</td>
<td>r1</td>
<td>r3</td>
</tr>
<tr>
<td>r2</td>
<td>r3</td>
<td>10</td>
</tr>
<tr>
<td>sub</td>
<td>r3</td>
<td>r2</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

7 instructions

Using Registers to Access Memory

• Registers can hold memory addresses

```c
int *p;
p = &x;
*p = *p + 8;
```

Instructions

```c
ld r1, p  // r1 ← mem[p]
ld r2, r1  // r2 ← mem[r1]
add r2, r2, 0x8  // increment x by 8
st r1, r2  // mem[r1] ← r2
```

• Many different ways to address
operands
  > not all Instruction sets include all modes

Making Instructions Machine Readable

• So far, still too abstract
  > add r1, r2, r3
• Need to specify instructions in machine readable
  form
• Bunch of Bits
  > 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.)
### Example: MIPS

**Register-Register**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Register-Immediate**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Branch**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>15</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Jump / Call**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</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

  - This is done by the hardware for speed
  - This is what the NiosII Instruction Set Simulator does

### 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?
  - how are memory operands located?
  - 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 method invocation (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 = $2 + 100`
  - `sw $1, 100($2)`  # `$1 = Memory[$2 + 100]`
  - `lui $1, 100`     # `$1 = 100 \times 2^{16}$`
  - `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**

---

© Alvin R. Lebeck

CPS 104
MIPS Instruction Formats

R-type: Register-Register

Op 31 26 25 21 20 16 15 11 10 5 0
Rs  Rt  Rd  shmt  func

I-type: Register-Immediate

Op 31 26 25 21 20 16 15 0
Rs  Rt  immediate

J-type: Jump / Call

Op 31 26 25 21 20 0
target

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

NiosII Instruction Formats

R-type: Register-Register

Op 31 26 25 21 20 16 15 11 10 5 0
A  B  C  D  OPX

I-type: Register-Immediate

Op 31 26 25 21 20 16 15 0
A  B  IMMEDIa

J-type: Jump / Call

Op 31 26 25 21 20 16 15 0
IMMEDIa

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

Operand Addressing: Register Direct

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

I-Type <op> rt, rs, immediate

Op 31 26 25 21 20 16 15 0
Rs  Rt  immediate

Immediate: 16 bit value
Operand Addressing:
Register Direct and Immediate

Add Immediate Example
addi $1, $2, 100 # $1 = $2 + 100

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 \( PC = PC + ? \)
  \[ \text{Size of instruction?} \]

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

PC is always automatically incremented to next instruction

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

Understand branches

Successor Instruction

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

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
    return $3;}

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

R-Type: <OP> rd, rs, rt

Jump Register Example
dr $31  # PC <- $31

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
add add $1,$2,$3 $1 = $2 + $3 3 operands
subb 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
subb unsigned subu $1,$2,$3 $1 = $2 – $3 3 operands
add imm. unsign. addiu $1,$2,100 $1 = $2 + 100 + constant
mult mult $1,$2 $1 = $2 x $3 64-bit signed product
mult unsigned multu $1,$2 $1 = $2 x $3 64-bit unsigned product
divide div $1,$2 $1 = Hi, Lo = $2 ÷ $3, Hi = quotient, Lo = remainder
divide unsigned divu $1,$2 $1 = Hi, Lo = $2 ÷ $3, Hi = quotient, Lo = 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

MIPS Logical Instructions
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 ori $1,$2,10 $1 = $2 | 10 Bitwise OR reg, const
xor immediate xori $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 logical sra $1,$2,10 $1 = $2 >> 10 Shift right (sign extend)
shift left logical sllv $1,$2,$3 $1 = $2 << $3 Shift left by var
shift right logical srlv $1,$2,$3 $1 = $2 >> $3 Shift right by var

MIPS Data Transfer Instructions
SW R3, 500(R4) Store word
SH R3, 502(R2) Store half
SB R2, 41(R3) Store byte
LW R1, 30(R2) Load word
LH R1, 40(R2) Load halfword
LHU R1, 40(R3) Load halfword unsigned
LB R1, 40(R3) Load byte
LBU 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 >
- bltz rs, offset <
- bgez rs, offset >=
- bltzal rs, offset if R[rs] < 0 then branch and link (into R 31)
- bgeal rs, offset >=

- 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>if ($1 == $2) go to PC+4+100</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1!=$2) go to PC+4+100</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>Compare &lt; constant; 2's comp.</td>
</tr>
<tr>
<td>set on less than imm.</td>
<td>slt $1,$2,100</td>
<td>Compare &lt; constant, natural numbers</td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>slti $1,$2,100</td>
<td>Compare &lt; constant, natural numbers</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>go to 10000</td>
</tr>
</tbody>
</table>

Signed v.s. Unsigned Comparison

R1= 0000 0000 0000 0000 0001
R2= 0000 0000 0000 0000 0010
R3= 1111 1111 1111 1111 1111

- After executing these instructions:
  - slt r4,r2,r1
  - slt r5,r3,r1
  - sltu r6,r2,r1
  - sltu r7,r3,r1

- What are values of registers r4 - r7? Why?
  - r4 = ; r5 = ; r6 = ; r7 = ;

Summary

- MIPS has 5 categories of instructions
  - Arithmetic, Logical, Data Transfer, Conditional Branch, Unconditional Jump
- 3 Instruction Formats
- Nios II Soft Processor and ISA Reference

Next Time
- Assembly Programming

Reading
- Ch. 2, Appendix A, Nios II Soft Processor
- HW #2