Instruction Set Architecture (2)

Hung-Wei Tseng
Recap: Instruction Set Architecture

• An abstraction between hardware and software
• Defines the functions that the underlying processor should perform
• Hardware should implement the functions of these instructions
  • Virtual machine: running MIPS programs/OSs using VMWare
  • Emulator: running NDS games on your PC with an emulator
  • Human: You will become one when you’re doing the homework!
• Programmer/Compiler/Assembler/Interpreter produces software that is a collection of instructions
Recap: MIPS ISA

- All instructions are 32 bits
- 32 32-bit registers
  - All registers are the same
  - $zero is always 0
- 50 opcodes
- Only load and store instructions can access memory
- Memory is “byte addressable”
  - Most modern ISAs are byte addressable, too
  - byte, half words, words are aligned

<table>
<thead>
<tr>
<th>name</th>
<th>number</th>
<th>usage</th>
<th>saved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
<td>no</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Recap: MIPS ISA (cont.)

- **3 instruction formats**
  - **R-type:** all operands are registers
    - 6 bits  5 bits  5 bits  5 bits  5 bits  6 bits
    - opcode  rs  rt  rd  shift amount  funct
  - **I-type:** one of the operands is an immediate value
    - 6 bits  5 bits  5 bits  16 bits
    - opcode  rs  rt  immediate / offset
  - **J-type:** non-conditional, non-relative branches
    - 6 bits  26 bits
    - opcode  target
Practice

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
label

and $t0, $t0, $zero #let i = 0
addi $t1, $zero, 100 #temp = 100
lw  $t3, 0($s0)     #temp1 = A[i]  
add  $v0, $v0, $t3   #sum += temp1
addi $s0, $s0, 4     #addr of A[i+1] 
addi $t0, $t0, 1     #i = i+1
bne $t1, $t0, LOOP  #if i < 100

LOOP:

1. Initialization
2. Load A[i] from memory to register
3. Add the value of A[i] to sum
4. Increase by 1
5. Check if i still < 100

Assume

int is 32 bits
$s0 = &A[0]$
$v0 = sum;
$t0 = i;

There are many ways to translate the C code.
But efficiency may be differ among translations
Announcement

• Reading quizzes due this Thursday
• Homework #1 due next Tuesday
• Rizwen’s office hour change
  • Please check the Google calendar in our website
• Tutor hours
  • Mingshan Wang is having CSE141 tutor tomorrow 2p-4p
• Things I learned from your responses
  • Piazza for discussion only
  • Apologize for and will avoid inappropriate wordings
  • Inconsistency with pre-release slides
Outline

• MIPS ISA
  • How to perform function calls
• x86
• Other ISA designs
Be a trained MIPS simulator

• What happens when you execute the next instruction?

A. PC = 10004
B. $v1 = 80000
C. $v1 = 57
D. Both A and B
E. Both A and C
Your turn...

- For the C code below:

```c
if(i < j)
    i++;
```

Which of the following is the correct translation in MIPS?

Assume that $t0$ has $i$ and $t1$ has $j$.

(slt rd,rs,rt does: R[rd]=1 if R[rs] < R[rt], else R[rd]=0)

$\text{Assume } t0 = i = 2 \quad t1 = j = 4$

A: $t2 = 1$

$t2 \neq 0$, jump to false

```mips
slt $t2, t0, t1
bne $t2, $zero, false
addi $t0, $t0, 1
false: next instruction
```

B: $t2 = 0$

addi, but it goes to true anyway...

```mips
slt $t2, t1, t0
bne $t2, $zero, true
true: addi $t0, $t0, 1
next instruction
```

C: $t2 = 0$

$t2 == 0$, jump to false

```mips
slt $t2, t1, t0
beg $t2, $zero, false
addi $t0, $t0, 1
false: next instruction
```

D: $t2 = 1$

$t2 == 1$, execute addi

```mips
slt $t2, t0, t1
beg $t2, $zero, false
addi $t0, $t0, 1
false: next instruction
```

E: none of the above
Addressing mode of branch instructions

- What type of addressing is used by branch instructions?
  A. Absolute. Branch instructions require a full 32-bit address to know the branch target
  B. Absolute. A 32-bit immediate gives us enough bits to specify a full address
  C. Relative. Branches tend to be backward branches which requires a negative immediate
  D. Relative. Branch targets tend to be close to the branch instruction
  E. Register Indirect

Branch is an I-type instruction:
int hanoi(int n)
{
    if(n==1)
        return 1;
    else
        return 2*hanoi(n-1)+1;
}

int main(int argc, char **argv)
{
    int n, result;
    n = atoi(argv[0]);
    result = hanoi(n);
    printf("%d\n", result);
}
Function calls

• Passing arguments
  • $a0-$a3
  • more to go using the memory stack
• Invoking the function
  • jal <label>
  • store the PC of jal +4 in $ra
• Return value in $v0
• Return to caller
  • jr $ra
int hanoi(int n)
{
    if(n==1)
        return 1;
    else
        return 2*hanoi(n-1)+1;
}

hanoi:   addi $a0, $a0, -1           // n = n-1
bne  $a0, $zero, hanoi_1            // if(n == 0) goto: hanoi_1
addi $v0, $zero, 1                  // return_value = 0 + 1 = 1
j    return                           // return
hanoi_1: jal  hanoi                  // call honai
sll  $v0, $v0, 1                     // return_value=return_value*2
addi $v0, $v0, 1                    // return_value = return_value+1
return:  jr   $ra                    // return to caller
Function calls

Caller (main)  Callee (hanoi)

Prepare argument for hanoi
$a0 - $a3 for passing arguments

addi $a0, $t1, $t0
jal  hanoi
sll  $v0, $v0, 1
addi $v0, $v0, 1
add  $t0, $zero, $a0
li   $v0, 4
syscall

hanoi:  addi $a0, $a0, -1
bne  $a0, $zero, hanoi_1
addi $v0, $zero, 1
j    return

hanoi_1:jal  hanoi
sll  $v0, $v0, 1
addi $v0, $v0, 1
return: jr   $ra

hanoi:  addi $a0, $a0, -1
bne  $a0, $zero, hanoi_1
addi $v0, $zero, 1
j    return

hanoi_1:jal  hanoi
sll  $v0, $v0, 1
addi $v0, $v0, 1
return: jr   $ra

Where are we going now?
Overwrite!

PC1: jal  hanoi
sll  $v0, $v0, 1
addi $v0, $zero, 1
li   $v0, 4
syscall

the current location of PC

Point to PC1+4

zero
at
v0
vl
a0
a1
a2
a3
t0
tl

14
Manage registers

• Sharing registers
  • A called function will modified registers
  • The caller may use these values later

• Using memory stack
  • The stack provides local storage for function calls
  • FILO (first-in-last-out)
  • For historical reasons, the stack grows from high memory address to low memory address
  • The stack pointer ($sp) should point to the top of the stack
Function calls

Caller

addi $a0, $t1, $t0
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
li $v0, 4
syscall

Callee

hanoi: addi $sp, $sp, -8
sw $ra, 0($sp)
sw $a0, 4($sp)
hanoi_0: addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return
hanoi_1: jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
return: lw $a0, 4(sp)
lw $ra, 0(sp)
addi $sp, $sp, 8
jr $ra

save shared registers to the stack, maintain the stack pointer
restore shared registers from the stack, maintain the stack pointer
Recursive calls

**Caller**

```
addi $a0, $zero, 2
addi $a0, $t1, $t0
jal hanoi
sll $v0, $v0, 1
addi $v0, $zero, 1
li $v0, 4
syscall
```

**Callee**

```
hanoi:  addi $sp, $sp, -8
          sw $ra, 0($sp)
          sw $a0, 4($sp)
          hanoi_0: addi $a0, $a0, -1
                    bne $a0, $zero, hanoi_1
                    addi $v0, $zero, 1
                    j return
hanoi_1: jal hanoi
          sll $v0, $v0, 1
          addi $v0, $zero, 1
          return: lw $a0, 4(sp)
                    lw $ra, 0(sp)
                    addi $sp, $sp, 8
                    jr $ra
```
Demo

- The overhead of function calls
- The keyword `inline` in C can embed the callee code at the call site
  - Eliminates function call overhead
- Does not work if it’s called using a function pointer
x86
x86

• The most widely used ISA
• A poorly-designed ISA
  • It breaks almost every rule of a good ISA
    • variable length of instructions
    • the work of each instruction is not equal
    • makes the hardware become very complex
  • It’s popular != It’s good
• You don’t have to know how to write it, but you need to be able to read them and compare x86 with other ISAs
• Reference
  • http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
# x86 Registers

<table>
<thead>
<tr>
<th>16bit</th>
<th>32bit</th>
<th>64bit</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>EAX</td>
<td>RAX</td>
<td>The accumulator register</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>EBX</td>
<td>RBX</td>
<td>The base register</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td>ECX</td>
<td>RCX</td>
<td>The counter</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>EDX</td>
<td>RDX</td>
<td>The data register</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
<td>Pointer to the base of stack frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rn</td>
<td>RnD</td>
<td>General purpose registers (8-15)</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
<td>Source index for string operations</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
<td>Destination index for string operations</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>EIP</td>
<td>RIP</td>
<td>Instruction pointer</td>
<td></td>
</tr>
<tr>
<td>FLAGS</td>
<td></td>
<td></td>
<td>Condition codes</td>
<td></td>
</tr>
</tbody>
</table>

These can be used more or less interchangeably.
MOV and addressing modes

- MOV instruction can perform load/store as in MIPS
- MOV instruction has many address modes
  - an example of non-uniformity

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>arithmetic op</th>
<th>memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $6, %eax</td>
<td>R[eax] = 0x6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl .L0, %eax</td>
<td>R[eax] = .L0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl %ebx, %eax</td>
<td>R[ebx] = R[eax]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl -4(%ebp), %ebx</td>
<td>R[ebx] = mem[R[ebp]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl (%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>movl -4(%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4-4]</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>movl %ebx, -4(%ebp)</td>
<td>mem[R[ebp]-4] = R[ebx]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl $6, -4(%ebp)</td>
<td>mem[R[ebp]-4] = 0x6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
## Arithmetic Instructions

- Accepts memory addresses as operands
- Register-memory ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Arithmetic op</th>
<th>Memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subl $16, %esp</code></td>
<td>$R[%esp] = R[%esp] - 16$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>subl %eax, %esp</code></td>
<td>$R[%esp] = R[%esp] - R[%eax]$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>subl -4(%ebx), %eax</code></td>
<td>$R[eax] = R[eax] - \text{mem}[R[ebx]-4]$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><code>subl (%ebx, %edx, 4), %eax</code></td>
<td>$R[eax] = R[eax] - \text{mem}[R[ebx]+R[edx]*4]$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><code>subl -4(%ebx, %edx, 4), %eax</code></td>
<td>$R[eax] = R[eax] - \text{mem}[R[ebx]+R[edx]*4-4]$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><code>subl %eax, -4(%ebx)</code></td>
<td>$\text{mem}[R[ebx]-4] = \text{mem}[R[ebx]-4]-R[eax]$</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Branch instructions

• x86 use condition codes for branches
  • Arithmetic instruction sets the flags
  • Example:
    cmp %eax, %ebx  #computes %eax-%ebx, sets the flag
    je <location>  #jump to location if equal flag is set

• Unconditional branches
  • Example:
    jmp <location>  #jump to location
Summation for x86

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
xorl %eax, %eax
.L2:
    addl (%ecx,%eax,4), %edx
    addl $1, %eax
    cmpl $100, %eax
    jne .L2
```

Assume

- int is 32 bytes
- %ecx = &A[0]
- %edx = sum;
- %eax = i;


MIPS v.s. x86

• Which of the following is NOT correct about these two ISAs?

A. x86 provides more instructions than MIPS
B. x86 usually needs more instructions to express a program
C. An x86 instruction may access memory for 3 times
D. An x86 instruction may be shorter than a MIPS instruction
E. An x86 instruction may be longer than a MIPS instruction
# MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA type</td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td>instruction width</td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td>code size</td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td>registers</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>addressing modes</td>
<td>reg+offset</td>
<td>scale+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
Uniformity of MIPS

- Only 3 instruction formats
  - opcodes, rs, rt, immediate are always at the same place
- Similar amounts of work per instruction
  - only 1 read from instruction memory
  - <= 1 arithmetic operations
  - <= 2 register reads, <= 1 register write
  - <= 1 data memory access
- Fixed instruction length
- Relatively large register file: 32 registers
- Reasonably large immediate field: 16 bits
- Wise use of opcode space: only 6 bit, R-type get another 6
Translate from C to Assembly

- gcc: gcc [options] [src_file]
  - compile to binary
    - gcc -o foo foo.c
  - compile to assembly (assembly in foo.s)
    - gcc -S foo.c
  - compile with debugging message
    - gcc -g -S foo.c
  - optimization
    - gcc -On -S foo.c
      - n from 0 to 3 (0 is no optimization)
Demo

• The magic of compiler optimization!
• Without optimization
• After compiled with -O3