Instruction Set Architecture (2)

Hung-Wei Tseng
Announcement

• Reading quizzes for 1.5-1.10 due tomorrow before class
  • Please check your csemoodle access
  • Will drop worst 2
• Homework #1 is online, please check
• Office hours: ThF 11a-12p @ CSE 2128
• Register your i-clicker!
• CSE141/CSE141L enrollment issues
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 PS3 games on your PC with PS3Emulator
  - Human: You will become one when you’re doing the homework!
- Programmer/Compiler/Assembler produces software consist 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 |
Outline

• MIPS ISA
  • How to perform function calls
• x86
• Other ISA designs
• 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
```

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
$s0 = &A[0]$
$v0 = sum$
$t0 = i$
Your turn...

- To implement the following C code, what are the instructions that we should put in the box?

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

**A:**
- `lw   $t0, 0($s2)`
- `addi $s2, $s2, 4`

**B:**
- `sw   $t1, 0($s2)`
- `addi $s2, $s2, 1`

**C:**
- `sw   $t0, 0($s2)`
- `addi $s2, $s2, 4`

**D:**
- `sw   $t1, 0($s2)`
- `addi $s2, $s2, 1`

**LOOP:**
- `add $t0, $zero, $zero`
- `addi $t1, $zero, 100`
- `addi $t0, $t0, 1`
- `beq $t0, $t1, LOOP`
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
Let’s write the hanoi()

```c
int hanoi(int n)
{
    if(n==1)
        return 1;
    else
        return 2*hanoi(n-1)+1;
}
```

```assembly
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 hanoi
    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)**
- Prepare argument for hanoi
- $a0 - a3$ for passing arguments

**Callee (hanoi)**
- 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

PC1: jal hanoi
- addi $a0, $t1, $t0
- sll $v0, $v0, 1
- addi $v0, $v0, 1
- add $t0, $zero, $a0
- li $v0, 4
- syscall

Where are we going now?

Overwrite!

Point to PC1+4

 registers

zero
at
v0
vl
a0
a1
a2
a3
t0
t1

 Callable (main) Calllee (hanoi)

Prepare argument for hanoi $a0 - a3$ for passing arguments
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)
        addi $a0, $a0, -1
        bne  $a0, $zero, hanoi_1
        addi $v0, $zero, 1
        jr   return
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
```

**Notes**

- Save shared registers to the stack, maintain the stack pointer.
- Restore shared registers from the stack, maintain the stack pointer.

---

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
</tr>
<tr>
<td>$at</td>
</tr>
<tr>
<td>$v0</td>
</tr>
<tr>
<td>$v1</td>
</tr>
<tr>
<td>$a0</td>
</tr>
<tr>
<td>$a1</td>
</tr>
<tr>
<td>$a2</td>
</tr>
<tr>
<td>$a3</td>
</tr>
<tr>
<td>$t0</td>
</tr>
<tr>
<td>$t1</td>
</tr>
</tbody>
</table>

**Memory**

```
sp
memory
```
Recursive calls

**Caller**

addi $a0, $zero, 2
addi $a0, $t1, $t0
PC1: 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
Function inlining

• The keyword inline in C can embed the callee code at the call site. For example:

<table>
<thead>
<tr>
<th>Original C Code</th>
<th>Equivalent code after inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for(i=0;i&lt;100;i++) sum += square(i);</code></td>
<td><code>for(i=0;i&lt;100;i++) sum+= i*i;</code></td>
</tr>
</tbody>
</table>

Which of the following is generally NOT correct regarding function inlining:

A. Help reducing the number of executed instructions
B. Help reducing the size of binary
C. Does not help if the function is called by function pointer
D. Potentially reduce the number of data memory accesses

Does NOT Help reducing the size of binary if the function is called multiple times
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
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
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>These can be used more or less interchangeably</td>
</tr>
<tr>
<td>Rn</td>
<td>RnD</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>
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] = \text{mem}[R[ebp]-4])</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl (%ecx,%eax,4), %eax</td>
<td>(R[eax] = \text{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] = \text{mem}[R[ebx]+R[edx]*4-4])</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>movl %ebx, -4(%ebp)</td>
<td>(\text{mem}[R[ebp]-4] = R[ebx])</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl $6, -4(%ebp)</td>
<td>(\text{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>subl $16, %esp</td>
<td>R[%esp] = R[%esp] - 16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subl %eax, %esp</td>
<td>R[%esp] = R[%esp] - R[%eax]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subl -4(%ebx), %eax</td>
<td>R[eax] = R[eax] - mem[R[ebx]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>subl (%ebx, %edx, 4), %eax</td>
<td>R[eax] = R[eax] - mem[R[ebx]+R[edx]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl -4(%ebx, %edx, 4), %eax</td>
<td>R[eax] = R[eax] - mem[R[ebx]+R[edx]*4-4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl %eax, -4(%ebx)</td>
<td>mem[R[ebx]-4] = 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>base+offset, base+index, scaled+index, scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
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)