Instruction Set Architecture

Hung-Wei Tseng
Recap: computer architecture

• The art or science of building computers
  • Processors and memories are everywhere in computers
  • We care about performance, power, cost, and etc. when building computers
• Moore’s law still allows us to shrink the die size, but power consumption limits the single chip/core performance
• We are now in the era of multicore processors, but parallelizing a program to utilize multithreaded processors is not easy
Announcement

- Reading quizzes due next Thursday before class
  - Will drop the grades of the lowest two quizzes
- We don’t have discussion sessions on Fridays
  - We will hold both sessions concurrently
  - You should go to the one that you registered
- CSE141 enrollment issues
  - We will open up more seats
  - One reason why we are losing the Friday sessions
Setup your i-c Hunger

• Register your i-c lick through TritonEd
• Set your channel to “CA”
  • Press on/off button for 2 seconds
  • Press C and then press A
Outline

• How we talk to computers
• What is an ISA (instruction set architecture)
• MIPS ISA
How we talk to computers
Von Neumann architecture

What are in these colored boxes?
Which of the following is the language that processors speak?

- **A**: Assembly language
- **B**: Machine language!
- **C**: Control signal spec
- **D**: High-level language

One-to-one mapping:

- `lw $15, 0($2)` <-> `10001100011000100000000000000000`
- `lw $16, 4($2)` <-> `10001100111100100000000000001000`
- `sw $16, 0($2)` <-> `10101100111100100000000000000000`
- `sw $15, 4($2)` <-> `10101100011000100000000000001000`

ALUOP[0:3] <= InstReg[9:11] & MASK

C

D

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
The stored program computer

- The program is data
  - a series of bits
    - these bits are “instructions”!
  - lives in memory

- Program counter
  - points to the current instruction
  - processor “fetches” instructions from where PC points.
  - advances/changes after instruction execution

```
120007a30:  0f00bb27  ldah gp,15(t12)
120007a34:  509cbd23  lda gp,-25520(gp)
120007a38:  00005d24  ldah t1,0(gp)
120007a3c:  0000bd24  ldah t4,0(gp)
120007a40:  2ca422a0  ldl t0,-23508(t1)
120007a44:  130020e4  beq t0,120007a94
120007a48:  00003d24  ldah t0,0(gp)
120007a4c:  2ca4e2b3  stl zero,-23508(t1)
120007a50:  0004ff47  clr v0
120007a54:  28a4e5b3  stl zero,-23512(t4)
120007a58:  20a421a4  ldq t0,-23520(t0)
120007a5c:  0e0020e4  beq t0,120007a98
120007a60:  0204e147  mov t0,t1
120007a64:  0304ff47  clr t2
120007a68:  0500e0c3  br 120007a80
```
From C/C++ to Machine Code

1. Intermediate Representation (e.g. gcc/llvm)
2. Compiler frontend
3. Compiler backend/optimizer
4. Object
5. Linker (e.g. ld)
6. Executable
7. Library
8. Machine code/binary
9. OS loader

One time cost
From Java to Machine Code

Java byte-code

Intermediate Representation

compiler frontend (e.g. javac)

compiler backend

Java byte-code

.one time cost

.JVM

Machine code

.class
From Script Languages to Machine Code

- **Intermediate Representation**
  - **compiler**
  - **binary**
  - **executable**
  - **Machine Code**

- **Runtime**
  - **interpreter (python, perl)**
What’s an Instruction Set Architecture (ISA)?
Which of the following is the language that processors speak?

**Assembly Language**

lw $15, 0($2) → 10001100011000100000000000000000
lw $16, 4($2) → 10001100111100100000000000001000
sw $16, 0($2) → 10101100111100100000000000000000
sw $15, 4($2) → 10101100011000100000000000001000

**Instruction Set Architecture**

ALUOP[0:3] <= InstReg[9:11] & MASK

**Machine Language!**

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

**One-to-One Mapping**

A

B

C

D
Instruction Set Architecture (ISA)

• The contract between the hardware and software
• Defines the set of operations that a computer/processor can execute
• Programs are combinations of these instructions
  • Abstraction to programmers/compilers
• The hardware implements these instructions in any way it choose.
  • Directly in hardware circuit. e.g. CPU
  • Software virtual machine. e.g. VirtualPC
  • Simulator/Emulator. e.g. DeSmuME
  • Trained monkey with pen and paper
How do you know about ISA?

• Which of the following is generally true about ISA?
  A. Many models of processors can support one ISA
  B. An ISA is unique to one model of processor
  C. Every processor can support multiple ISAs
  D. Each processor manufacturer has its own ISA
  E. None of the above
Example ISAs

- x86: intel Xeon, intel Core i7/i5/i3, intel atom, AMD Athlon/Opteron, AMD FX, AMD A-series
- ARM: Apple A-Series, Qualcomm Snapdragon, TI OMAP, nVidia Tegra
- MIPS: Sony/Toshiba Emotion Engine, MIPS R-4000(PSP)
- DEC Alpha: 21064, 21164, 21264
- PowerPC: Motorola PowerPC G4, Power 6
- IA-64: Itanium
- SPARC and many more ...
What should an instruction look like?

• Operations
  • What operations?
    e.g. add, sub, mul, and etc.
  • How many operations?

• Operands
  • How many operand?
  • What type of operands?
    • Memory/register/label/number (immediate value)

• Format
  • Length? How many bits? Equal length?
  • Formats?
What ISA includes?

- Instructions: what programmers want processors to do?
  - Math: add, subtract, multiply, divide, bitwise operations
  - Control: if, jump, function call
  - Data access: load and store
- Architectural states: the current execution result of a program
  - Registers: a few named data storage that instructions can work on
  - Memory: a much larger data storage array that is available for storing data
  - Program Counter (PC): the number/address of the current instruction
How to encode an instruction into binary

- Assuming that the ISA has 36 operations and 32 registers, how many of the following machine code/instruction formats are valid?

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>6 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>II.</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>V.</td>
<td>opcode</td>
<td>register</td>
<td>register</td>
<td>register</td>
<td>V.</td>
<td>opcode</td>
<td>register</td>
</tr>
<tr>
<td>III.</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V.</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. 1  
B. 2  
C. 3  
D. 4  
E. 5
We will study two ISAs

• MIPS
  • Simple, elegant, easy to implement
    • That’s why we want to implement it in CSE141L
  • Designed with many-year ISA design experience
  • The prototype of a lot of modern ISAs
    • MIPS itself is not widely used, though

• x86
  • Ugly, messy, inelegant, hard to implement, ...
  • Designed for 1970s technology
  • The dominant ISA in modern computer systems

You should know how to write MIPS code after this class

You should know how to read x86 code after this class
MIPS
MIPS ISA

- All instructions are 32 bits
- 32 32-bit registers
  - All registers are the same
  - $zero is always 0
- 50 opcodes
- 3 instruction formats
  - R-type: all operands are registers
  - I-type: one of the operands is an immediate value
  - J-type: non-conditional, non-relative branches

<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>
### MIPS ISA (cont.)

- 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>Byte addresses</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td></td>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td></td>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td></td>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0xFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Half Word Addresses</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td></td>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0x0006</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word Addresses</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>
We have the following C code:
```c
for (int i = 0; i < 10 ; i++) {
    A[i] = i
}
```
where A is a double (64 bits) array. Let’s suppose that the base address of A is 0x1000. What byte address(es) correspond to A[5].

A. 0x1028
B. 0x1040
C. 0x1028-0x102F
D. 0x1040-0x1047
E. None of the above
**R-type**

- op $rd, $rs, $rt
  - 3 regs.: add, addu, and, nor, or, sltu, sub, subu
  - 2 regs.: sll, srl
  - 1 reg.: jr

- 1 arithmetic operation, 1 I-memory access

**Example:**
  - opcode = 0x0, funct = 0x20
- `sll $t0, $t1, 8: R[8] = R[9] << 8`
  - opcode = 0x0, shamt = 0x8, funct = 0x0
I-type

- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory access
- Example:
  - lw $s0, 4($s2):
    \[ R[16] = \text{mem}[R[18]+4] \]
I-type (cont.)

- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory memory access
- Example:
  - beq $t0, $t1, -40
    if (R[8] == R[9]) PC = PC + 4 + 4*(-40)
J-type

- op immediate
  - j, jal
- 1 instruction memory access, 1 arithmetic op
- Example:
  - jal quicksort
    - \( R[31] = PC + 4 \)
    - PC = quicksort
Practice

- Translate the C code into assembly:

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

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

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
$s0 = &A[0]$
$v0 = sum;$
$t0 = i;$

```
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:
```