Instruction Set Architecture (II) & Performance (I)

Hung-Wei Tseng
Recap: 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
Recap: Instruction Set Architecture (ISA)
 Recap: MIPS ISA

• All instructions are 32 bits
• 32 32-bit registers
  • All registers are the same
  • $zero is always 0
• 50 opcodes
  • Arithmetic/Logic operations
  • Load/store operations
  • Branch/jump operations
• 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>$k0-$k1</td>
<td>26-27</td>
<td>OS kernel</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>
“Abstracted” MIPS Architecture

CPU

Program Counter
0x00000004

Registers
$zero
$at
$v0
$at
$v1
$s0
$a1
$s2
$a3
$s4
$s5
$s6
$s7
$s8
$s9
$s10
$s11
$s12
$s13
$s14
$s15
$s16
$s17
$s18
$s19
$s20
$s21
$s22
$s23
$s24
$s25
$s26
$s27
$s28
$s29
$s30
$s31
$t0
$t1
$t2
$t3
$t4
$t5
$t6
$t7
$k0
$k1
$gp
$sp
$fp
$ra

ALU

add
sub
and
or
bne
beq
jal

lw
sw

Memory

0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C
0xFFFFFFE0
0xFFFFFFE4
0xFFFFFFE8
0xFFFFFFEC
0xFFFFFFF0
0xFFFFFFF4
0xFFFFFFF8
0xFFFFFFFc

0xFFFFFEE0
0xFFFFFEE4
0xFFFFFEE8
0xFFFFFEC
0xFFFFF0
0xFFFFF4
0xFFFFFF8
0xFFFFFFFC

64-bit
32-bit
2^32 Bytes

Registers

Memory

ALU

add
sub
and
or
bne
beq
jal

lw
sw
Frequently used MIPS instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>addi $s1, $s2, 20</td>
<td>$s1 = $s2 + 20</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>and $s1, $s2, $s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>or $s1, $s2, $s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td></td>
<td>andi</td>
<td>andi $s1, $s2, 20</td>
<td>$s1 = $s2 &amp; 20</td>
</tr>
<tr>
<td></td>
<td>sll</td>
<td>sll $s1, $s2, 10</td>
<td>$s1 = $s2 * 2^10</td>
</tr>
<tr>
<td></td>
<td>srl</td>
<td>srl $s1, $s2, 10</td>
<td>$s1 = $s2 / 2^10</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>lw</td>
<td>lw $s1, 4($s2)</td>
<td>$s1 = mem[$s2+4]</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>lw $s1, 4($s2)</td>
<td>mem[$s2+4] = $s1</td>
</tr>
<tr>
<td>Branch</td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>if($s1 == $s2), PC = PC + 100</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>if($s1 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td>Jump</td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>
Recap: 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 (if i = 0, it must < 100)
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
Recursive calls

**Caller**
- addi $a0, $zero, 2
- addi $a0, $t1, $t0
- jal hanoi
- sll $v0, $v0, 1
- add $t0, $zero, $a0
- li $v0, 4
- syscall

**Callee**

```assembly
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
```

**Registers**
- zero
- at
- v0
- v1
- a0
- a1
- a2
- a3
- t0
- t1

**Memory**
- hanoi_0+4
- hanoi_0+4
- sp
Overview of x86 ISA
x86 ISA

- 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
The abstracted x86 machine architecture

CPU

Registers
- RAX
- RBX
- RCX
- RDX
- RSP
- RBP
- RSI
- RDI
- R8
- R9
- R10
- R11
- R12
- R13
- R14
- R15
- RIP
- FLAGS
  - CS
  - SS
  - DS
  - ES
  - FS
  - GS

64-bit

Memory

2^64 Bytes

0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFF8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8

ALU
- ADD
- SUB
- IMUL
- AND
- OR
- XOR
- JMP
- JE
- CALL
- RET

MOV
# 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>Rn</td>
<td>RnD</td>
<td>General purpose registers (8-15)</td>
<td></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>These can be used more or less interchangeably</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] = 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>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:
    \texttt{cmp \%eax, \%ebx} #computes \%eax-%ebx, sets the flag
    \texttt{je <location>} #jump to location if equal flag is set

• Unconditional branches
  • Example:
    \texttt{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)
Demo

• The magic of compiler optimization!
• Without optimization
• After compiled with -O3
User-defined data structure

- Programming languages allow user to define their own data types
- In C, programmers can use `struct` to define new data structure

```c
struct node {
    int data;
    struct node *next;
};
```

How many bytes each “struct node” will occupy?
Generate an x86 assembly file on your PC!

- You can do this on your PC with Linux or MacOS
  - You need to have gcc/Xcode installed on your Linux/MacOS machine
  - You will need to complete your 3rd project under this environment — if you’re using **Windows** or **MacOS**, you need to install VMWare/VirtualBox to host a linux system
- gcc -S source_file
  - Using “gcc -S hello_world.c”, you can get “hello_world.s”
  - Demo
Addressing and accessing the data structure

• Memory allocation
  • Each object-instance of the data structure occupies consecutive memory locations that can accommodate all members in this object-instance
  • The starting address of each object-instance must be aligned with the multiple of 8
    • Try to have as many members aligned with address multiplied by 8 using the smallest amount of space, but also maintains the member order
    • Although ARM supports unaligned access — they are slow

• Memory access:
  • The base address register points to the beginning of the accessing object-instance
  • The offset points to the member — one of the reason why we have an offset field
Memory layout of data structures

```
struct node {
    int data;
    struct node *next;
};
```
The result of `sizeof(struct student)`

- Consider the following data structure:

```c
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What’s the output of

```c
printf("%lu\n", sizeof(struct student));
```

A. 20  
B. 28  
C. 32  
D. 36  
E. 40
Performance

Hung-Wei Tseng
Outline

• Definition of performance
• Execution time
• What affects your performance
Performance
What do you want for a computer?

- Latency/Execution time
- Frame rate
- Responsiveness
- Real-time
- Throughput
- Cost
- Volume
- Weight
- Battery life
- Low power/low temperature
- Reliability
Match (Best) Performance Metric to Domain

- Consider the following performance metrics
  1. Network Bandwidth (data/sec)
  2. End-to-end Latency (ms)
  3. Frame Rate (frames/sec)
  4. Throughput (ops/sec)

Which option contains the best match of the most important performance metric for each application?

<table>
<thead>
<tr>
<th></th>
<th>WoW or LOL (Online gaming)</th>
<th>Halo (FPS)</th>
<th>Torrent download</th>
<th>Google Server Farm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>None of the above</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How about running a single program

- Latency/Execution time
- Frame rate
- Responsiveness
- Real-time
- Throughput
- Cost
- Volume
- Weight
- Battery life
- Low power/low temperature
- Reliability

The most direct measurement of performance
Evaluating the execution time of a program
Recap: Von Neumann architecture

CPU is a dominant factor of performance since we heavily rely on it to execute programs.

By pointing “PC” to different part of your memory, we can perform different functions!
Recap: Clock — synchronizing hardware components

- A hardware signal defines when data for any specific component is ready to use by others
  - Think about the clock in real life!
- We use edge-triggered clocking
  - Values stored in the sequential logic is updated only on a clock edge
The simplest kind of performance
- Shorter execution time means better performance
- Usually measured in seconds

**Execution Time**

1. How many of these?
2. Instruction Count!
3. How long is it take to execution each of these?
4. Cycles per instruction * cycle time

```
Processor

instruction memory

120007a30: 0f00bb27 ldah gp,15(t12)
120007a34: 509bd23 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
```
Execution Time = \( \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \)

- How many instructions are executed?
- How long does it take to execute each instruction?

- \( ET = IC \times CPI \times CT \)
- IC (Instruction Count)
- CPI (Cycles Per Instruction)
- CT (Seconds Per Cycle)
- 1 Hz = 1 second per cycle; 1 GHz = 1 ns per cycle
Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle. If the processor runs at 2 GHz, how long is the execution time?

A. 500000 ns
B. 1000000 ns
C. 2000000 ns
D. 3500000 ns
E. None of the above

\[
\text{Execution Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

\[
500000 \times (0.8 \times 1 + 0.2 \times 6) \times 0.5 \text{ ns} = 500000 \text{ ns}
\]
Relative performance

- Can be confusing
  - A runs in 12 seconds
  - B runs in 20 seconds
  - We know A is faster, but
    - \( A/B = 0.6 \), so A is 40% faster, or 1.4X faster, or B is 60% slower
    - \( B/A = 1.67 \), so A is 67% faster, or 1.67X faster, or B is 67% slower

- Needs a precise definition
Speedup

- Compare the relative performance of the baseline system and the improved system
- Definition

\[
\text{Speedup} = \frac{\text{Execution time} \text{ baseline}}{\text{Execution time} \text{ improved system}}
\]