Instruction Set Architecture

Hung-Wei Tseng
Recap: von Neumann model

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!
Modern computers

The same spirit, but different implementations
How CPU handle instructions

- Instruction fetch: where?
  - Instruction memory

- Decode:
  - What’s the instruction? registers
  - Where are the operands? ALUs

- Execute

- Memory access data memory
  - Where is my data?

- Write back registers
  - Where to put the result

- Determine the next PC
Where is “computer architecture”
Outline

• What is an ISA (Instruction Set Architecture)?
• How source code becomes a running program
• Overview of MIPS ISA
What’s an Instruction Set Architecture (ISA)?
Where is “ISA”?
How to setup your i-clicker

• Set your channel to “CA”
• Register your i-clicker
  • You can do this through TritonEd
• It’s OK if you don’t have a clicker this week — but you should have it next week!
How do you know about ISAs?

• Which of the following is generally true about ISAs?
  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 Ryzen, AMD FX, AMD A-series
  
  *Almost every desktop/laptop/server is using this*

- **ARM**: Apple A-Series, Qualcomm Snapdragon, TI OMAP, nVidia Tegra

  *Almost every mobile phone/tablet is using this*

- **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 ...**
How do you know about ISAs?

- Which of the following is generally true about ISAs?
  
  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
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
Assembly language

- The human-readable representation of “instructions”/“machine language”
- Has a direct mapping between assembly code and instructions
  - Assembly may contain “pseudo instructions” for programmer to use
  - Each pseudo instruction still has its own mapping to a set of real machine instructions

```
add $v0, $a1, $a2
```

<table>
<thead>
<tr>
<th>000000</th>
<th>00101</th>
<th>00110</th>
<th>00010</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Instruction Set Architecture (ISA)
From C/C++ to Machine Code

- Intermediate Representation
  - compiler frontend (e.g. gcc/llvm)
  - compiler backend assembler/optimizer

- Object
  - linker (e.g. ld)
  - Executable
  - Library

- machine code/binary

- one time cost

- OS loader
From Java to Machine Code

- Java byte-code
  - Intermediate Representation
  - .class
  - compiler frontend (e.g. javac)
  - compiler backend
  - .class
  - JVM
  - Machine code
  - JVM
  - one time cost

- Java
From Script Languages to Machine Code

- **Python**
- **Perl**

### Diagram

- **Intermediate Representation**
- **Compiler**
- **Binary**
- **Executable**
- **Runtime**
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
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?

\[ y = a + b \]

```
add $s1, $s2, $s3
add $s1, $s2, 64
```
Overview of 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

MIPS ISA

<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>
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>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA</td>
<td>0x0000</td>
<td>0xAA15</td>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
<td>0x0002</td>
<td>0x13FF</td>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
<td>0x0004</td>
<td>.</td>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
<td>0x0006</td>
<td>.</td>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
<td>...</td>
<td>.</td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>.</td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
<td>...</td>
<td>.</td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
<td>0xFFF</td>
<td>.</td>
<td>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>
“Abstracted” MIPS Architecture

CPU

Registers
$zero
$at
$v0
$at
$s0
$s1
$s2
$s3
$s4
$s5
$s6
$s7
$s8
$s9
$s0
$gp
$fp
$ra

Program Counter
0x00000004

ALU

add
sub
and
or
bne
beq
jal

lw
sw

Memory

0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C

0xFFFFFFF0
0xFFFFFFF4
0xFFFFFFF8
0xFFFFFFFc

0xFFFFFFE0
0xFFFFFFE4
0xFFFFFFE8
0xFFFFFFEC

0xFFFFFFE0
0xFFFFFFE4
0xFFFFFFE8
0xFFFFFFEC

0xFFFFFFFC
0xFFFFFFFF
0xFFFFFFFF4
0xFFFFFFFF8
0xFFFFFFFFc
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>
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)`:
    \[ R[16] = \text{mem}[R[18]+4] \]

- Only two addressing modes

- `lw $s0, 0($s2)`
Data transfer instructions

- The ONLY type of instructions that can interact with memory in MIPS
- Two big categories
  - Load (e.g., lw): copy data from memory to a register
  - Store (e.g., sw): copy data from a register to memory
- Two parts of operands
  - A source or destination register
  - Target memory address = base address + offset
    - Register contains the “base address”
    - Constant as the “offset”
    - \(8(\$s0) = (\text{the content in } \$s0) + 8\)
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 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:

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

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
- $s0 = &A[0]$ 
- $v0 = \text{sum}$
- $t0 = \text{i}$

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

<table>
<thead>
<tr>
<th></th>
<th>A:</th>
<th>B:</th>
<th>C:</th>
<th>D:</th>
</tr>
</thead>
</table>
  $t2 = 1$ | slt $t2$, $t0$, $t1$ | slt $t2$, $t1$, $t0$ | slt $t2$, $t1$, $t0$ | slt $t2$, $t0$, $t1$ |
  $t2$ != 0 | bne $t2$, $zero$, false | bne $t2$, $zero$, true | bne $t2$, $zero$, false | bne $t2$, $zero$, true |
  to false | addi $t0$, $t0$, 1 | true: addi $t0$, $t0$, 1 | true: addi $t0$, $t0$, 1 | false: next instruction |
  false: next instruction |

  E: none of the above

Assume $t0 = i = 2$
$t1 = j = 4$
$\$t2$ = 0, execute addi, but it goes to true anyway...$
Function calls
```c
int main(int argc, char **argv)
{
    n = atoi(argv[0]);
    bar = rand();
    printf("%d\n", foo(n));
    return 0;
}

int foo(int n)
{
    int i, sum=0;
    for(i = 0; i < n; i++) {
        sum+=i;
    }
    return sum;
}
```
Function calls

- Parameters
- Transfer the control from the caller (the code calling the function) to the callee (the function being called)
- Prepare resource (registers/memory locations) for the function call
- Compute
- Return the value
- Return the control to the caller
How to manage the memory space?

Function A

Function B

Function C

Register values

Register values

Register values

Memory

stack pointer

stack pointer

stack pointer

stack pointer
```c
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 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)**

- addi $a0, $t1, $t0
- jal hanoi

**Callee (hanoi)**

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

**Where are we going now?**

We are supposed to go to PC1+4 not hanoi_1+4!

**Overwrite!**

- $a0 != $t1+$t0
- the current location of PC

**PC1:**

- jal hanoi
- sll $v0, $v0, 1
- add $v0, $v0, 1
- 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
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**

PC1:
- 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 $sra, 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
add $t0, $zero, $a0
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
```
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