Instruction Set Architecture

Prof. Usagi
Outline

• What’s an “instruction set architecture (ISA)”
• Overview of MIPS ISA
• How to express a program in MIPS ISA
Instruction Set Architecture (ISA)
Popular ISAs

- x86
- ARM
- MIPS
- RISC-V
- SWERV Core
- Apple A13 Bionic
- NVIDIA Tegra X1
- Qualcomm Snapdragon
- Sony
- Broadcom
- Intel Core i7
- AMD Ryzen
Instruction Set Architecture (ISA)

- The **contract** between the hardware and software
- Provide an **abstraction** of the underlying processor
  - Defines the set of operations that a computer/processor can execute
  - Defines the **memory** space that a program can use
- Programs are combinations of these instructions
- 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

We’re abstracting a von Neumann machine!
Instruction Set Architecture (ISA)
How my “C code” becomes a “program”

Source Code

Compiler (e.g., gcc)

Program

Linker

Objects, Libraries

Program

Processor

Memory

Instructions

Data

Storage

509cbd23

00c2e800

0f00bb27

509cbd23

00005d24

00000d24

2ca422a0

13002a40

00003d24

2ca4e2b3

00c2e800

00000008

00c2f000

00c2f800

00c3000

00c2e800

00000008

00c2f000

00c2f800

00c3000

509cbd23

00005d24

00000d24

2ca422a0

13002a40

00003d24

2ca4e2b3

00c2e800

00000008

00c2f000

00c2f800

00c3000

00c2e800

00000008

00c2f000

00c2f800

00c3000

00c2e800

00000008

00c2f000

00c2f800

00c3000

00c2e800

00000008

00c2f000

00c2f800

00c3000

00c2e800

00000008

00c2f000

00c2f800

00c3000

00c2e800

00000008

00c2f000

00c2f800

00c3000
How my "Java code" becomes a "program"
How my “Python code” becomes a “program”
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>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>Shift Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110</td>
<td>000000</td>
<td>00101</td>
<td>00110</td>
<td>00010</td>
</tr>
</tbody>
</table>
Elements in an ISA

• Operations
  • Types of operations: Arithmetic/Logical, memory access, control-flow (e.g., branch, function calls)
  • How many operations should the processor support?
• Operands
  • Types of operands — register, constant, memory addresses
  • Number of operands — 0, 1, 2, 3 or ?
  • Sizes of operands — byte, 16-bit, 32-bit, 64-bit
• Memory space
  • The size of memory that programs can use
  • The addressing of each memory locations
  • The modes to represent those addresses
An Overview of MIPS ISA
The abstracted “MIPS” machine

- **Registers**: $zero, $at, $v0, $v1, $a0, $a1, $a2, $a3, $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $t8, $t9, $k0, $k1, $gp, $sp, $fp, $ra

- **Program Counter**: 0x0000000000000004

- **ALU Operations**: add, addi, and, andi, or, ori, xor, beq, blt, jal, jr, lw, sw

- **Memory**: 2^32 bytes

- **32-bit Addresses**: 0x00000000, 0x00000008, 0x00000010, 0x00000018, 0x00000020, 0x00000028, 0x00000030, 0x00000038, 0xFFFFFFC0, 0xFFFFFFC8, 0xFFFFFFD0, 0xFFFFFFD8, 0xFFFFFFE0, 0xFFFFFFE8, 0xFFFFFFF0, 0xFFFFFFF8, 0xFFFFFFFF0, 0xFFFFFFFF8
MIPS ISA

- All instructions are 32 bits
- 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>Reg. Name</th>
<th>Reg. Num</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</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>
## Subset of MIPS instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></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><strong>Logical</strong></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><strong>Data Transfer</strong></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><strong>Branch</strong></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 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>

*The only type of instructions can access memory*
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 l-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)

Only two addressing modes:
- lw $s0, $s2($s1)
- add $s2, $s2, $s1
- 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
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 = sum;
- $t0 = i;

There are many ways to translate the C code. But efficiency may be differ among translations.
Supporting function calls
## Subset of MIPS instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></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><strong>Logical</strong></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><strong>Data Transfer</strong></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><strong>Branch</strong></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 != $s2), PC = PC + 100</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>jal</td>
<td>jal 25</td>
<td>$ra = PC + 4, PC = 100</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>

The only type of instructions can access memory. We use them to support function calls!
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;
}
```

- **Arguments/Parameters**: `main` takes `argc` and `argv`, `foo` takes `n`.
- **Local Variables**: `n`, `bar`, `i`, `sum`.
- **Return Value**: `foo` returns `sum`.

Change the PC/control to the callee.
How these two help to achieve function calls

<table>
<thead>
<tr>
<th>Jump</th>
<th>jal</th>
<th>jal 25</th>
<th>$ra = PC + 4, PC = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>PC = $ra</td>
</tr>
</tbody>
</table>

- Passing arguments
  - $a0-$a3
  - more to go using the **memory stack**
- Invoking the function
  - jal <label>
    - label is the location of the target function
  - store the PC of jal +4 in $ra — why +4?
- Return value in $v0
- Return to caller
  - jr $ra
#include <stdio.h>

int sum(int n);

int main() {
    int number, result;

    printf("Enter an integer > 0: ");
    scanf("%d", &number);

    result = sum(number);
    printf("sum = %d", result);
    return 0;
}

int sum(int n) {
    if (n == 1)
        return 1;
    else
        return n + sum(n-1);
}

……
add $a0, $v0, $zero
jal sum

PC1:
add $a1, $zero, $v0
li $a0, 4
syscall

……
add $a0, $v0, $zero, 1
beq $a0, $zero, return
jal sum
add $v0, $v0, $a0
// ret = sum(n-1)+n-1
addi $v0, $v0, 1
// ret += 1

sum:

return:
jr $ra
// return to caller
What happens when we execute the code

Say we input 2!

Is that correct?
Where are we going?
Is $ra pointing to the right place?

Registers

What happens when we execute the code

Say we input 2!

Is that correct?
Where are we going?
Is $ra pointing to the right place?
What happens when we execute the code

Say we input 2!

```
main:    ……
……
……
……
add $a0, $v0, $zero
jal sum
……
……
……
……
……
……
……
……
add $sp, $sp, -8
sw $a0, 0($sp)
sw $ra, 4($sp)
……
……
……
add $a0, $a0, -1
add $v0, $zero, 1
beq $a0, $zero, return
jal sum
……
……
……
add $sp, $sp, 8
jr $ra
……
……
……
add $sp, $sp, 8
jr $ra
```

```
PC1:
……
add $a1, $zero, $v0
li $a0, 4
syscall
……
……
……
addi $sp, $sp, -8
sw $a0, 0($sp)
sw $ra, 4($sp)
```

```
sum:     addi $a0, $a0, -1
addi $v0, $zero, 1
beq $a0, $zero, return
jal sum
……
……
……
add $sp, $sp, 8
jr $ra
```

```
return:  lw $a0, 0($sp)
lw $ra, 4($sp)
```

Registers

Memory
Other ISAs
The abstracted x86 machine

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

CPU

Memory

64-bit

ALU

- ADD
- SUB
- IMUL
- AND
- OR
- XOR

JMP
JE
CALL
RET

MOV

Memory:
- 0x0000000000000000
- 0x0000000000000008
- 0x0000000000000010
- 0x0000000000000018
- 0x0000000000000020
- 0x0000000000000028
- 0x0000000000000030
- 0x0000000000000038
- 0xFFFFFFFFFFFFFFC0
- 0xFFFFFFFFFFFFFFC8
- 0xFFFFFFFFFFFFFFD0
- 0xFFFFFFFFFFFFFFD8
- 0xFFFFFFFFFFFFFFE0
- 0xFFFFFFFFFFFFFFE8
- 0xFFFFFFFFFFFFFFF0
- 0xFFFFFFFFFFFFFFF8

MOV

2^64 Bytes

64-bit
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
  • http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
# 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></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] - \text{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] - \text{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] - \text{mem}[R[%ebx]+R[%edx]__4__4])</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl %eax, -4(%ebx)</td>
<td>(\text{mem}[R[%ebx]__4] = \text{mem}[R[%ebx]__4__4]-R[%eax])</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Summation for x86

• Translate the C code into assembly:

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

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

```assembly
xorl %eax, %eax
.L2:  addl (%ecx,%eax,4), %edx
     addl $1, %eax
     cmpl $100, %eax
     jne .L2
```
## MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
Popular ISAs

Reduced Instruction Set Computers (RISC)

Complex Instruction Set Computers (CISC)
<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
<tr>
<td>ISA type</td>
<td>Reduced Instruction Set</td>
<td>Complex Instruction Set</td>
</tr>
<tr>
<td></td>
<td>Computers (RISC)</td>
<td>Computers (CISC)</td>
</tr>
</tbody>
</table>
How many operations: CISC v.s. RISC

• CISC (Complex Instruction Set Computing)
  • Examples: x86, Motorola 68K
  • Provide many powerful/complex instructions
    • Many: more than 1503 instructions since 2016
    • Powerful/complex: an instruction can perform both ALU and memory operations

• RISC (Reduced Instruction Set Computer)
  • Examples: ARMv8, RISC-V, MIPS (the first RISC instruction, invented by the authors of our textbook)
  • Each instruction only performs simple tasks
  • Easy to decode
Announcements

• Login piazza, Canvas
• Check our website
• Reading quiz due tomorrow before class
Computer Science & Engineering