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”?

Devices  Micro-architecture  Processors  Instruction Set Architectures  Operating Systems  Compilers Runtime Virtual Machines  Programming Languages  Programmers/ Users
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 ...
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
```
Instruction Set Architecture (ISA)

add
sub
mul
...

lw
sw
...

bne
jal
...

Instruction Set Architecture

Emulator/Virtual machine
From C/C++ to Machine Code

C

compiler frontend
(e.g. gcc/llvm)

Intermediate
Representation

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
(compiler frontend)
(e.g. javac)

 одной разовой стоимости

class

Machine code

JVM

class

JVM
From Script Languages to Machine Code

18
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?

Examples:

- \( y = a + b \)

In assembly language:

- `add $s1, $s2, $s3`
- `add $s1, $s2, 64`
Overview of MIPS ISA
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>
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>0xFFFFC</td>
<td>.</td>
<td>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>
“Abstracted” MIPS Architecture

CPU

Registers

$zero
$at
$v0
$v1
$s0
$s1
$s2
$s3
$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
$a0
$a1
$a2
$a3
$a4
$a5
$a6
$a7
$a8
$a9
$a10
$a11
$a12
$a13
$a14
$a15
$t0
$t1
$t2
$t3
$t4
$t5
$t6
$t7
$t8
$t9
$k0
$k1
$k2
$k3
$k4
$k5
$k6
$k7
$sp
$fp
$ra

Program Counter

0x00000004

ALU

add
sub
and
or
bne
beq
jal

lw
sw

Memory

2^32 Bytes

64-bit

32-bit

0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C
0xFFFFFFE0
0xFFFFFFE4
0xFFFFFFE8
0xFFFFFFEC
0xFFFFFFF0
0xFFFFFFF4
0xFFFFFFF8
0xFFFFFFFC
0xFFFFF000
0xFFFFF004
0xFFFFF008
0xFFFFF00C
0xFFFFF010
0xFFFFF014
0xFFFFF018
0xFFFFF01C
### 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

- \text{op $rt, $rs, immediate}
- \text{addi, addiu, andi, beq, bne, ori, slti, sltiu}
- \text{op $rt, offset($rs)}
- \text{lw, lbu, lhu, ll, lui, sw, sb, sc, sh}
- 1 arithmetic op, 1 I-memory and 1 D-memory access
- Example:
  - \text{lw $s0, 4($s2):}
    \begin{align*}
    \text{R[16] &= mem[R[18]+4]}
    \end{align*}
  - \text{add $s2, $s2, $s1}
  - \text{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
    \[ R[31] = PC + 4 \]
    \[ PC = \text{quicksort} \]
Practice

- Translate the C code into assembly:

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

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

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

There are many ways to translate the C code.
But efficiency may be differ among translations
Function calls
int main(int argc, char **argv)
{
    n = atoi(argv[0]);
    bar = rand();
    printf("%d\n", foo(n));
    return 0;
}

int foo(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?

Memory

Register values
Function A

Register values
Function B

Register values
Function C

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()

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

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)

Callee (hanoi)

Prepare argument for hanoi
$s0 - s3$ for passing arguments

where are we going now?
we are supposed to go to PC1+4 not hanoi_1+4!

PC1: jal hanoi

PC1: jal hanoi

jal hanoi

addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return

hanoi_1: jal hanoi

addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return

return: jr $ra

Overwrite!
$s0 != s1+s0$

the current location of PC
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**

```assembly
addi $a0, $t1, $t0
jal  hanoi
sll  $v0, $v0, 1
addi $v0, $v0, 1
li   $v0, 4
syscall
```

**Callee**

```assembly
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
        j    return
hanoi_1:jal  hanoi
        sll  $v0, $v0, 1
        addi $v0, $v0, 1
return: 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**

```
PC1:

    addi $sp, $sp, -8
    sw $ra, 0($sp)
    sw $a0, 4($sp)

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, $zero, 1
        j return

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
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
  • http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
The abstracted x86 machine architecture

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

ALU:
- ADD
- SUB
- IMUL
- AND
- OR
- XOR

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

These can be used more or less interchangeably.
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 $16, %esp</td>
<td>$\text{R}[$%esp$] = $\text{R}[$%esp$] - 16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subl %eax, %esp</td>
<td>$\text{R}[$%esp$] = $\text{R}[$%esp$] - $\text{R}[$%eax$]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subl -4(%ebx), %eax</td>
<td>$\text{R}[$eax$] = $\text{R}[$eax$] - \text{mem}[\text{R}[\text{ebx}]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>subl (%ebx, %edx, 4), %eax</td>
<td>$\text{R}[$eax$] = $\text{R}[$eax$] - \text{mem}[\text{R}[\text{ebx}]+\text{R}[$edx$]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl -4(%ebx, %edx, 4), %eax</td>
<td>$\text{R}[$eax$] = $\text{R}[$eax$] - \text{mem}[\text{R}[\text{ebx}]+\text{R}[$edx$]*4-4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl %eax, -4(%ebx)</td>
<td>$\text{mem}[\text{R}[\text{ebx}]-4] = \text{mem}[\text{R}[\text{ebx}]-4]-\text{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];
}

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>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

```c
struct node {
    int data;
    struct node *next;
};
```
Taxonomy of ISAs
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
    • Each instruction takes more cycles to execute

• RISC (Reduced Instruction Set Computer)
  • Examples: ARMv8, MIPS (the first RISC instruction, invented by the authors of our textbook)
  • Each instruction only performs simple tasks
  • Easy to decode
  • Each instruction takes less cycles to execute
How many operands: accumulator, stack, memory-register, load-store

<table>
<thead>
<tr>
<th></th>
<th>stack</th>
<th>accumulator</th>
<th>register-memory</th>
<th>load-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>operands</td>
<td>0</td>
<td>1</td>
<td>2 or 3</td>
<td>3</td>
</tr>
<tr>
<td>operations</td>
<td>work on top elements of the stack</td>
<td>work on one accumulator once</td>
<td>work on registers and memory addresses</td>
<td>work only on several registers, only load/store instructions can interact with memory</td>
</tr>
<tr>
<td>A=X<em>Y-B</em>C</td>
<td>push B</td>
<td>load B</td>
<td>mul R1, mem[X], mem[Y]</td>
<td>load t1, X</td>
</tr>
<tr>
<td></td>
<td>push C</td>
<td>mul C</td>
<td>mul R2, mem[B], mem[C]</td>
<td>load t2, Y</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>store temp</td>
<td>sub A, R1, R2</td>
<td>mul t2, t1, t2</td>
</tr>
<tr>
<td></td>
<td>push X</td>
<td>load X</td>
<td></td>
<td>load t3, B</td>
</tr>
<tr>
<td></td>
<td>push Y</td>
<td>mul Y</td>
<td></td>
<td>load t4, C</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>sub temp</td>
<td></td>
<td>mul t4, t4, t3</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>store A</td>
<td></td>
<td>sub t4, t3, t4</td>
</tr>
<tr>
<td></td>
<td>pop A</td>
<td></td>
<td></td>
<td>store t4, A</td>
</tr>
</tbody>
</table>
public static int fibonacci(int n) {
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}

Most instructions doesn’t have an argument!
## How many operands: accumulator, stack, memory-register, load-store

<table>
<thead>
<tr>
<th>Operands</th>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-memory</th>
<th>Load-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operands</td>
<td>0</td>
<td>1</td>
<td>2 or 3</td>
<td>3</td>
</tr>
<tr>
<td>Operations</td>
<td>work on top elements of the stack</td>
<td>work on one accumulator once</td>
<td>work on registers and memory addresses</td>
<td>work only on several registers, only load/store instructions can interact with memory</td>
</tr>
<tr>
<td>A=X<em>Y-B</em>C</td>
<td>push B, push C, mul X, push Y, mul sub pop A</td>
<td>load B, mul C, store temp, load X, mul Y, sub temp, store A</td>
<td>mul R1, mem[X], mem[Y], mul R2, mem[B], mem[C], sub A, R1, R2</td>
<td>load t1, X, load t2, Y, load t3, B, load t4, C, mul t4, t4, t3, sub t4, t3, t4, store t4, A</td>
</tr>
</tbody>
</table>

### Additional Notes:
- **A=X*Y-B*C**
  - High code density
  - Easy to compile
  - Fewest instructions
  - Simple hardware
  - Fewest memory access
  - Hardware stack design
  - Most memory accesses
  - Complex hardware design
  - Code size
  - Instruction count