CSCI 410
8 – Assembly and Machine Language

Instruction Set Architecture

ISA is a description of a computer platform.
• Machine language is the binary representation of the ISA’s instructions
  • ISA and machine language are 1:1
  • ISA describes the instructions, machine language assigns specific values to each instruction
• The computer is the hardware realization of the ISA
  • ISA and computer are 1:many
  • Computer “recognizes” machine language of ISA
• Examples: x86, ARM, SPARC

CISC vs RISC

• Reduced Instruction Set Computing (MIPS, ARM)
  • Few, simple instructions
  • Fast execution of instructions (e.g., 1 clock cycle)
  • Hardware design simpler, but more work for compiler
• Complex Instruction Set Computing (x86)
  • Do more with each instruction
  • Many instructions to choose from
  • Slower execution of instructions (many cycles)
  • Makes hardware harder, compiler easier
  • Can be implemented on top of RISC

Assembly Language

Symbolic notation for machine language programs
• Thinly overlays machine language
• Much easier for programmers
• Requires a translator (assembler)

Most assembly languages provide facilities beyond just symbolic representation of machine language:
• Labels instead of addresses
• High-level constants (e.g., strings)
• Macro languages

Typical Assembly Instructions

Copy data from source to destination:
   mov %r1, addr       // copy register 1 to memory location addr
   mov %r2, %r1        // copy r2 to r1

Math and logical:
   add %r1, %r2, %r3   // add r1 and r2 and store in r3
   add %r1, %r2, %r3   // add 7 to r1 and store in r3
   also xor, or, etc.

Branching/jumping:
   jmp addr           // jump to addr
   beq %r1, %r2, addr // jump to addr if r1 == r2
   bne, blt, bgt, etc.

Many variations, addressing modes, etc., plus floating point instructions, other extensions.

Example: Hello, World!

If using gcc:
   gcc -S hello.c

.copy "Hello, World!"
.text
.globl main
.type main, @function
main:
   pushq %rbp
   movq %rsp, %rbp
   subq $.lclob, %rdi
   call puts
   movl $0, %eax
   popq %rbp
   ret
**Example: Sum 1 to 100**

```
Example: Sum 1 to 100

.globl main
.type main, @function
main:
pushq %rbp
movq %rsp, %rbp
movl $0, -4(%rbp)
movl $0, -8(%rbp)
jmp .L2
.L3:
movl -8(%rbp), %eax
addl %eax, -4(%rbp)
addl $1, -8(%rbp)
 jmp .L3
.L2:
```

**Stored Program Computer**

**Hack Computer**

**Hack Registers**

A: general purpose; only register that can be set with a constant value (immediately)

D: general purpose

M: not really a register, just an alias for RAM[A]

**Two Instruction Types**

- **A**: Loads a constant value into register A

- **C**: All the heavy lifting:
  - Calculates values
  - Stores results
  - Performs jumps

**A-Instructions**

@value

Sets A equal to value
- Value can be constant, e.g., @42
- Value can be a symbol (label), e.g., @foo
C-Instructions - Destination

Three parts:

destination = computation; jump

destination and jump are optional.

destination is one of: {A, D, M, AD, MD, AM, AMD}

The destination indicates where to store the result of computation (no destination means result not stored).

C-Instructions - Computation

Three parts:

destination = computation; jump

computation calculates functions:

- \( x + y \)
- \( x - y \)
- \( x \& y \)
- \( x | y \)
- \( x \neg \)
- \( 1 \)
- \( 0 \)

\( x \) is in \{A, D, M\}

\( y \) is in \{A, D, M, 1\}

Some restrictions apply: e.g., cannot use both A and M.

C-Instructions - Jump

Three parts:

destination = computation; jump

Jump is one of: {JMP, JEQ, JNE, JLT, JLE, JGT, JGE}

The jump instruction looks at result of computation (except for jmp):

- JEQ means “jump if equal (to zero)”
- JNE means “jump if not equal (to zero)”
- JLT means “jump if less than (zero)”
- JMP means “jump. Just do it.”

Result of successful jump is to set PC to contents of A.

Labels

Not part of machine language, just assembly:

(NAME) in program fixes NAME as an alias for next instruction location in ROM.

E.g.,

```
D=0
(LOOP)
D=D+1
@LOOP
0;JMP
```

Variables

Using any symbol in an A-instruction not declared somewhere else using (NAME) defines a variable.

Essentially, if you do

```
@i
```

You are declaring a variable, which gets a unique location in RAM (starting at address 16), this address is loaded into the A register.

E.g.,

```
@i  //loads address of i into A
M=0  //sets i to zero
[LOOP]
@i
M=M+1
@LOOP
0;JMP
```

General Notes

- Typically, need to use frequent A-instructions between C-instructions
  - Load an address or value (A)
  - Perform a calculation and/or do a jump (C)

- Note you never want to use M in a calculation and also do a jump:
  - M uses A for a RAM address
  - Jumps use A for a ROM address

- By convention, end a Hack program with infinite loop:
  - \( \text{(END)} \)
  - \( @\text{END} \)
  - \( 0;\text{JMP} \)
Summary

Hack commands:

A-command: @value  // set A to value
C-command: comp = jump  // dest = and jump
  // are optional

Where:
  comp = 0, 1, D, A, D, A, D-A, 1, D+1, D-A, D+1, M, A-M, D+M, D-M,
  M=M, D=M, M=M+1, D=M+1, and

  dest = +, D, A, AM, AD, AMD, or null

In the command dest = comp; jump, the jump materializes if (comp jump 0) is true. For example, in D=D+1,JLT, we jump if D+1 < 0.

Source: www.nand2tetris.org

If Statements in Hack

C-like language:

if condition {
  code block 1
} else {
  code block 2
}

code block 3

Hack:

D <- condition

@IF_FALSE
D;JEQ
code block 1
@END
0;JMP
(IF_FALSE)
code block 2
(END)
code block 3

Challenge Problem

Sum 3 numbers stored in RAM[0], RAM[1], RAM[2].

If the result is odd, add 1.

Store the final result in RAM[3].

While Statements in Hack

C-like language:

while condition {
  code block 1
}

code block 2

Hack:

(LOOP)
D <- condition

@END
D;JEQ
code block 1
@LOOP
0;JMP
(IF_FALSE)
code block 2
(END)
code block 3

Example: Sum 1 to 100

C-like language:

sum = 0
i = 1
while (i <= 100) {
  sum += i
  i++
}

Hack:

// i refers to some RAM location
sum @0
// i refers to some RAM location
i @1

(LOOP)
sum // i = 0
i+1 // i = 1 - 100
(D:Int) // i <= 100 ? goto END
sum // i = 0
i+1 // i = 1

(END)
sum // sum += i

i+1 // i++
(Loop)

@sum // code block 1

@END // code block 2

Hack Demo
I/O

- **Screen:**
  - 256 x 512 black & white screen
  - Memory mapped to 8K region starting at 0x4000
  - 0x4000 aliased to special symbol "SCREEN"
  - Read/write: 1 = black pixel, 0 = white pixel

- **Keyboard:**
  - Memory mapped to address 0x6000
  - 0x6000 aliased to special symbol "KBD"
  - 0 = no key pressed, otherwise ASCII code appears
  - Read only

Project 4

Write two programs in Hack assembly:

1. **Mult.asm** — multiply two numbers and store the result. Note — no multiply instruction available. How will you proceed?

2. **Fill.asm** — make the screen entirely black when a key is pressed, or white when no key is pressed.

Hack Machine Language: A-instructions

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>Where value is either a non-negative decimal number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: 15 bit addresses at most 32K memory.

Source: www.nand2tetris.org

Hack Machine Language: C-instructions

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>dest = comp</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Source: www.nand2tetris.org

Hack Machine Language: C-instructions (comp)

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>comp</th>
<th>dest</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1</td>
<td>1</td>
<td>d1</td>
</tr>
</tbody>
</table>

Source: www.nand2tetris.org

Hack Machine Language: C-instructions (dest)

<table>
<thead>
<tr>
<th>Symbolic</th>
<th>comp</th>
<th>dest</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1</td>
<td>1</td>
<td>d1</td>
</tr>
</tbody>
</table>

Source: www.nand2tetris.org
## Hack Machine Language: C-instructions (jump)

<table>
<thead>
<tr>
<th>comp</th>
<th>dest</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
<th>7</th>
<th>Mnemonic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>null</td>
<td>No jump</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>eqt</td>
<td>If out &gt; 0 jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>eqq</td>
<td>If out = 0 jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>eqe</td>
<td>If out &lt; 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>lot</td>
<td>If out &gt; 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>lte</td>
<td>If out = 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>lqe</td>
<td>If out &lt; 0 jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>jmp</td>
<td>Jump</td>
</tr>
</tbody>
</table>

Source: www.nand2tetris.org