Virtual machines

Abstractions for computers

<table>
<thead>
<tr>
<th>Level 5</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 4</td>
<td>High-Level Language</td>
</tr>
<tr>
<td>Level 3</td>
<td>Operating System</td>
</tr>
<tr>
<td>Level 2</td>
<td>Instruction Set</td>
</tr>
<tr>
<td>Level 1</td>
<td>Architecture</td>
</tr>
<tr>
<td>Level 0</td>
<td>Digital Logic</td>
</tr>
</tbody>
</table>

Problems with programming using machine code

- Difficult to remember instructions
- Difficult to remember variables
- Hard to calculate addresses/relocate variables or functions
- Need to handle instruction encoding (e.g., jr Rt)
Virtual machines

Abstractions for computers

TOY assembly

<table>
<thead>
<tr>
<th>opcode</th>
<th>mnemonic</th>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hlt</td>
<td>hlt</td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>add rd, rs, rt</td>
</tr>
<tr>
<td>2</td>
<td>sub</td>
<td>sub rd, rs, rt</td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>and rd, rs, rt</td>
</tr>
<tr>
<td>4</td>
<td>xor</td>
<td>xor rd, rs, rt</td>
</tr>
<tr>
<td>5</td>
<td>shl</td>
<td>shl rd, rs, rt</td>
</tr>
<tr>
<td>6</td>
<td>shr</td>
<td>shr rd, rs, rt</td>
</tr>
<tr>
<td>7</td>
<td>lda</td>
<td>lda rd, addr</td>
</tr>
<tr>
<td>8</td>
<td>ld</td>
<td>ld rd, addr</td>
</tr>
<tr>
<td>9</td>
<td>st</td>
<td>st rd, addr</td>
</tr>
<tr>
<td>A</td>
<td>ldi</td>
<td>ldi rd, rt</td>
</tr>
<tr>
<td>B</td>
<td>sti</td>
<td>sti rd, rt</td>
</tr>
<tr>
<td>C</td>
<td>bz</td>
<td>bz rd, addr</td>
</tr>
<tr>
<td>D</td>
<td>bp</td>
<td>bp rd, addr</td>
</tr>
<tr>
<td>E</td>
<td>jr</td>
<td>jr rd (rt)</td>
</tr>
<tr>
<td>F</td>
<td>jl</td>
<td>jl rd, addr</td>
</tr>
</tbody>
</table>

Assembler's task:

- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Build machine instructions in proper format
- Convert data constants into internal machine representations (data formats)
- Write object program and the assembly listing
Forward Reference

Definition

- A reference to a label that is defined later in the program

Solution

- Two passes: scan the source program for label definition, address accumulation, and address assignment
- Second pass: perform most of the actual instruction translation

Assembly version of REVERSE

```assembly
int A[32];
A DUP 32
10: C020
  lda R1, 1
  lda RA, A
20: 7101
  lda RC, 0
21: 7A00
i=0;
22: 7C00
Do {
  RD=stdin;
  read ld RD, 0xFF
23: BDF
  if (RD==0) break.
  bz RD, exit
24: CD29
  add R2, RA, RC
25: 12AC
  A[i]=RD;
26: BD02
  add RC, R2
27: 1CC1
  i=i+1;
28: C023
} while (1);
29: FF2B
printr():
  exit jl RF, printr
30: FF2B
  hlt
31: 0000
```

Assembly version of REVERSE

```assembly
printr()
{
  : print reverse
  : array address (RA)
  : number of elements (RC)
  do {
    : i=0-31
    i=i-1;
    printr sub RC, RC, R1
    add R2, RA, RC
    ldi RD, R2
    st RD, 0xFF
  } while (i>=0);
  print A[i];
  bp RC, printr
  bz RC, printr
  hlt RF, printr
}
```

toyasm < reverse.asm > reverse.toy

Function Call: A Failed Attempt

Goal: $x \times y \times z$.

- Need two multiplications: $x \times y$, $(x \times y) \times z$.
  - Solution 1: write multiply code 2 times.
  - Solution 2: write a TOY function.

A failed attempt:

- Write multiply loop at 30-36.
- Calling program agrees to store arguments in registers A and B.
- Function agrees to leave result in register C.
- Call function with jump absolute to 30.
- Return from function with jump absolute.

Reason for failure.

- Need to return to a VARIABLE memory address.
Multiplication Function

Calling convention.
- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result c = a \times b in register C.
- Register 1 is scratch.
- Overwrites registers A and B.

Client program to compute x \times y \times z.
- Read x, y, z from standard input.
- Note: PC is incremented before instruction is executed.
  - value stored in register F is correct return address

Function Call

Contract between calling program and function:
- Calling program stores function parameters in specific registers.
- Calling program stores return address in a specific register.
  - jump-and-link
- Calling program sets PC to address of function.
- Function stores return value in specific register.
- Function sets PC to return address when finished.
  - jump register

What if you want a function to call another function?
- Use a different register for return address.
- More general: store return addresses on a stack.
### Procedure prototype

With a stack, the procedure prototype is changed. It allows us to have a deeper call graph by using the stack.

```
mul RE, push

mul

jl RE, pop

jr RF
```

### Assembly programming flow

```
asm source

assembler

object

• Combine separate object codes

linker

• Supply necessary information for references between them

executable

loader

Bring the object program into memory for execution

Target machine
```

### Linking

Many programs will need multiply. Since multiply will be used by many applications, could we make multiply a library?

Toyasm has an option to generate an object file so that it can be later linked with other object files.

That is why we need linking. Write a subroutine `mul3` which multiplies three numbers in RA, RB, RC together and place the result in RD.

Three files:
- stack.obj: implementation of stack, needed for procedure
- mul.obj: implementation of multiplication.
- multest.obj: main program and procedure of mul3

```
toylink multest.obj mul.obj stack.obj > multest.toy
```
object file (multest.asm)

main:
  ld RA, A
  ld RB, B
  ld RC, C
  st RD, RF
  mul3
  jl R, mul

  lda RC, 0
  lda R1, 1
  id R, SIXTEEN

  m_loop:
    sub R2, R2, R1
    shl R3, RA, R2
    shr R4, RB, R2
    and R4, R4, R1
    bz R4, m_end
    add RC, RC, R3
    m_end:
      bp R2, m_loop
      jl R, pop

; calculate A*B*C
main:
  ld RA, A
  ld RB, B
  ld RC, C
  st RD, RF
  mul3
  jl R, mul
  lda RD, 0
  add RD, RC, R0
  add RA, RC, R0
  add RB, RD, R0
  add RC, RD, RC, R0
  jl R, mul
  jr RF

object file (mul.obj)

SIXTEEN DW 16
; multiply RC=RA*RB
; return address is in RF
; it will modify R2, R3 and R4 as well
mul:
  jl R, push
  add RA, RC, R0
  add RD, RC, R0
  add RB, RD, R0
  add RC, RD, RC, R0
  jl RF, mul
  add RB, RD, R0
  add RC, RD, R0
  add RA, RC, R0
  add RB, RD, R0
  add RC, RD, RC, R0
  jl RF, mul

import table

push 1 0x10
pop 1 0x1B

start address=0

stack.obj

push 0x10
pop 0x16

mul.obj

push 0x10
pop 0x16

multest.obj

Linking

start address=0

Resolve external symbols

These are literals. No need to relocate

need to fill in address of push once we know it

need to fill in address of pop once we know it
Virtual machines

Abstractions for computers

Operating system is a resource allocator
- Managing all resources (memory, I/O, execution)
- Resolving requests for efficient and fair usage

Operating system is a control program
- Controlling execution of programs to prevent errors and improper use of the computer
Abstractions for computers

Assignment #2

Assigned: 11/03/2008
Due: 11:59pm 11/16/2008

Part 1 (50%): write a procedure BCD to convert a hexadecimal number into a BCD (Binary-Coded Decimal). The input number is placed in RA. The result should be placed in RB. The return address is in RF. (Hint: you need to implement division)

Part 2 (30%): write a procedure CNT0 to count 0’s in an array. The address of the array is placed at RA. The size of the array is specified by RC. The result should be placed in RB. The return address is in RF.

Part 3 (20%): write a program to read a series of numbers specified by the user from stdin until the input is 0x0000. Count the number of 0-bits in the input array and display this number using BCD in stdout.