Definition of DISM (Diminished Instruction Set Machine)

version 1.1 Jay Ligatti

1 Introduction

A DISM is a virtual machine with a simple, RISC-like instruction set.

A DISM has 8 *general-purpose registers* (numbered 0 to 7) and a *program-counter register* (PC). As usual, the PC stores the address of the next instruction to execute. A DISM also has a block of *data memory* from address 0 to address 65535. In our DISM simulator, these memory components are implemented with the following data types:

```
#define NUM_REGS 8
#define DATA_SIZE 65536
unsigned int PC;           // program counter
unsigned int R[NUM_REGS];  // registers
unsigned int M[DATA_SIZE]; // data memory
```

2 Initialization

All DISM registers (including the PC) and the data memory get initialized to 0.

At startup, a DISM program's instructions are sequentially stored in a *code memory* beginning at address 0. Hence, the PC initially points to the first program instruction.

DISM code memory and data memory are completely separate. Instructions in DISM programs can only load from and store to (i.e., read and write) *data* memory.

3 Instructions

A DISM can execute only twelve types of instructions:

Instruction	Meaning
add d s1 s2	R[d] <- R[s1] + R[s2]
sub d s1 s2	R[d] <- R[s1] - R[s2] (R[d]<-0 when R[s2]>R[s1])
mul d s1 s2	R[d] <- R[s1] * R[s2]
mov d n	R[d] <- n
lod d s i	R[d] <- M[R[s]+i]
str d i s	M[R[d]+i] <- R[s]
jmp s i	PC <- R[s] + i
beq s1 s2 n	If R[s1] = R[s2] then PC <- n
bgt s1 s2 n	If R[s1] > R[s2] then PC <- n
rdn d	Read natural number from screen into R[d]
ptn s	Print natural number R[s] to screen
hlt s	Halt the DISM with code R[s]

In the instruction definitions above, n denotes a natural number (0, 1, 2, ...), while i denotes an integer (which may be negative). Opcodes (add, sub, etc.) must be in lower case.

After executing any instruction that does not otherwise set the PC register, a DISM increments the PC by 1 so that it points to the next instruction to execute. The last instruction executed must be a hlt instruction to prevent a DISM from attempting to execute a nonexistent instruction.

DISM programs may also contain comments. A comment begins with a semicolon; the rest of a line is ignored after a semicolon.

Finally, an instruction may optionally be preceded by a *symbolic label* and then a colon. A symbolic label contains a # and then a string of ASCII letters (a-z and A-Z) and digits (0-9). When a label precedes an instruction in this way, other instructions may reference the labeled instruction's location in code memory by using the label itself. The following example should clarify this use of symbolic labels.

4 An Example

For example, consider the following DISM program.

```
            rdn 1          ;read n into register 1
            rdn 2          ;read m into register 2
            mov 3 1        ;move value 1 into register 3
    #LOOP:  beq 2 0 #END   ;if m==0 then goto end
            ptn 1          ;print n
            sub 2 2 3      ;decrement m
            jmp 0 #LOOP    ;goto loop beginning
    #END:   hlt 0          ;halt with code 0
```

This program reads two numbers n and m and then prints n to the screen m times. The program is equivalent to the following:

```
            rdn 1          ;read n into register 1
            rdn 2          ;read m into register 2
            mov 3 1        ;move value 1 into register 3
            beq 2 0 7      ;if m==0 then goto end
            ptn 1          ;print n
            sub 2 2 3      ;decrement m
            jmp 0 3        ;goto loop beginning
            hlt 0          ;halt with code 0
```

The #LOOP and #END labels have been resolved to 3 and 7 because they labeled instructions stored at addresses 3 and 7 in code memory (recall that instructions get stored in code memory at consecutive addresses, with the first instruction at address 0).

A few additional examples of DISM programs appear at http://www.cse.usf.edu/~ligatti/compilers/23/a1/dism/.

5 A DISM Simulator

You can download source code implementing a DISM simulator at: http://www.cse.usf.edu/~ligatti/compilers/23/a1/dism/sim-dism/. The following files comprise the simulator's source code: ast.c, ast.h, dism.l, dism.y, interp.c, and interp.h.

Having copied all these source-code files into the same directory, you can compile the simulator on the C4 machines with the following commands.

```
> flex dism.l
> bison dism.y
> sed -i '/extern YYSTYPE yylval/d' dism.tab.c
> gcc dism.tab.c ast.c interp.c –osim-dism
```

These commands produce an executable program called sim-dism, which is our DISM simulator.

After copying the test program nm.dism into the same folder, you can execute nm.dism in the simulator as follows.

```
> ./sim-dism nm.dism
Enter a natural number: 8
Enter a natural number: 4
8
8
8
8
Simulation completed with code 0 at PC=7.
>
```

Obtaining useful debugging information

The sim-dism simulator can also be executed in a *debug mode*. This mode of execution outputs additional information regarding (1) which instructions are being executed and (2) what the registers and memory contain after every instruction executes.

For example, consider the following simple DISM program:

```
     ptn 0
     hlt 1
```

When we execute this program in debug mode, sim-dism behaves as shown on the following page.

```
> ./sim-dism simple.dism D
****** begin abstract syntax tree for DISM program ********
0:INSTR_LIST
1:  PTN_AST
2:    INT_AST(0)
1:  HLT_AST
2:    INT_AST(1)
******* end abstract syntax tree for DISM program *********

******interpreting the following instruction at location 0:
0:PTN_AST
1:  INT_AST(0)
0
Register contents after executing this instruction:
  0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0 PC:1
Nonzero values currently stored in memory:
  <none>

******interpreting the following instruction at location 1:
0:HLT_AST
1:  INT_AST(1)
Simulation completed with code 0 at PC=1.
>
```