

Appendix C

Mini Simulator

The Mini machine simulator is simply a C program stored in the file `mini.c`. It reads instructions and data in the form of hex characters from the standard input file, `stdin`. The instruction format is as specified in Section 6.5.1, and is specified with a structure called `fmt` in the header file, `mini.h`.

The simulator begins by calling the function `boot()`, which loads the Mini machine memory from the values in the standard input file, `stdin`, one memory location per line. These values are numeric constants, and zeroes for program variables and temporary locations. The `boot()` function also initializes the program counter, PC (register 1), to the starting instruction address.

The simulator fetches one instruction at a time into the instruction register, `ir`, decodes the instruction, and performs a switch operation on the operation code to execute the appropriate instruction. The user interface is designed to allow as many instruction cycles as the user wishes, before displaying the machine registers and memory locations. The display is accomplished with the `dump()` function, which sends the Mini CPU registers, and the first sixteen memory locations to `stdout` so that the user can observe the operation of the simulated machine. The memory locations displayed can be changed easily by setting the two arguments to the `dumppmem()` function. The displays include both hex and decimal displays of the indicated locations.

As the user observes a program executing on the simulated machine, it is probably helpful to watch the memory locations associated with program variables in order to trace the behavior of the original MiniC program. Though the compiler produces no memory location map for variables, their locations can be determined easily because they are stored in the order in which they are declared, beginning at memory location 3. For example, the program that computes the cosine function begins as shown here:

```
... public static void main (String [] args)
{ float cos, x, n, term, eps, alt;
```

In this case, the variables `cos`, `x`, `n`, `term`, `eps`, and `alt` will be stored in that order in memory locations 3 through 8.

The source code for the Mini machine is in the file `mini.c` and is shown below:

```
/* simulate the Mini architecture */
/* 32-bit word addressable machine, with 16 general regis-
ters and
   16 floating-point registers.
r1: program counter
ir: instruction register
r0-r15: general registers (32 bits)
fpr0-fpr15: floating-point registers (32 bits)

instruction format:
  bits  function
  0-3  opcode      1    r1 = r1+s2
                    2    r1 = r1-s2
                    4    r1 = r1*s2
                    5    r1 = r1/s2
                    7    pc = S2 if flag      JMP
                    8    flag = r1 cmp s2    CMP
                    9    r1 = s2             Load
                   10    s2 = r1             Store
                   11    r1 = 0              Clear

  4      mode  0    s2 is 20 bit address
              1    s2 is 4 bit reg (r2) and 16 bit
                  offset (o2)

  5-7  cmp    0    always true
              1    ==
              2    <
              3    >
              4    <=
              5    >=
              6    !=

  8-11 r1      register address for first operand
  12-31 s2     storage address if mode=0
  12-15 r2     part of storage address if mode=1
  16-31 o2     rest of storage address if mode=1
```

```

if mode=1, s2 = c(r2) + o2 */

#include <stdio.h>
#include "mini.h"
#define PC reg[1]

FILE * tty; /* read from keyboard */

unsigned long addr;
unsigned int flag, r2, o2;

main ()
{
int n = 1, count;

boot(); /* load memory from stdin */

tty = fopen ("/dev/tty", "r"); /* read from keyboard
even if stdin is
redirected */

while (n>0)
{
for (count = 1; count<=n; count++)
{ /* fetch */
ir.full32 = memory[PC++].instr;
if (ir.instr.mode==1)
{ o2 = ir.instr.s2 & 0x0ffff;
r2 = ir.instr.s2 & 0xf0000;
addr = reg[r2] + o2;}
else addr = ir.instr.s2;

switch (ir.instr.op)
{ case ADD: fpreg[ir.instr.r1].data =
fpreg[ir.instr.r1].data +
memory[addr].data;
break;
case SUB: fpreg[ir.instr.r1].data =
fpreg[ir.instr.r1].data -
memory[addr].data;
break;
case MUL: fpreg[ir.instr.r1].data =
fpreg[ir.instr.r1].data *
memory[addr].data;
break;
}
}
}
}

```

```
        case DIV:      fpreg[ir.instr.r1].data =
                        fpreg[ir.instr.r1].data /
                        memory[addr].data;
                        break;
        case JMP:      if (flag) PC = addr; /* conditional
                        jump */
                        break;
        case CMP:      switch (ir.instr.cmp)
                        {case 0:      flag = TRUE;      /* uncondi-
                        tional */
                        break;
                        case 1:      flag = fpreg[ir.instr.r1].data
                        == memory[addr].data;
                        break;
                        case 2:      flag = fpreg[ir.instr.r1].data
                        < memory[addr].data;
                        break;
                        case 3:      flag = fpreg[ir.instr.r1].data
                        > memory[addr].data;
                        break;
                        case 4:      flag = fpreg[ir.instr.r1].data
                        <= memory[addr].data;
                        break;
                        case 5:      flag = fpreg[ir.instr.r1].data
                        >= memory[addr].data;
                        break;
                        case 6:      flag = fpreg[ir.instr.r1].data
                        != memory[addr].data;
                        }
        case LOD:      fpreg[ir.instr.r1].data =
                        memory[addr].data;
                        break;
        case STO:      memory[addr].data = fpreg[ir.instr.r1].data;
                        break;
        case CLR:      fpreg[ir.instr.r1].data = 0.0;
                        break;
        case HLT:      n = -1;
        }
}
```



```

void boot()
/* load memory from stdin */
{ int i = 0;

    scanf ("%8lx%*[\n]\n", &PC);          /* starting ad-
                                         dress of instructions */

    while (EOF!=scanf ("%8lx%*[\n]\n", &memory[i++].instr));
}

```

The only source files that have not been displayed are the header files. The file `miniC.h` contains declarations, macros, and includes which are needed by the compiler but not by the simulator. The file `mini.h` contains information needed by the simulator.

The header file `miniC.h` is shown below:

```

/* Size of hash table for identifier symbol table */
#define HashMax 100

/* Size of table of compiler generated address labels */
#define MAXL 1024

/* memory address type on the simulated machine */
typedef unsigned long ADDRESS;

/* Symbol table entry */
struct Ident
{char * name;
 struct Ident * link;
 int type; /* program name = 1,
           integer = 2,
           real = 3 */
 ADDRESS memloc;};

/* Symbol table */
struct Ident * HashTable[HashMax];

/* Linked list for declared identifiers */
struct idptr
{struct Ident * ptr;

```

```

        struct idptr * next;
    };
struct idptr * head = NULL;
int dcl = TRUE; /* processing the declarations section */

/* Binary search tree for numeric constants */
struct nums
    {ADDRESS memloc;
      struct nums * left;
      struct nums * right;};
struct nums * numsBST = NULL;

/* Record for file of atoms */
struct atom
    {int op; /* atom classes are shown below */
      ADDRESS left;
      ADDRESS right;
      ADDRESS result;
      int cmp; /* comparison codes are 1-6 */
      int dest;
    };

/* ADD, SUB, MUL, DIV, and JMP are also atom classes */
/* The following atom classes are not op codes */
#define NEG 10
#define LBL 11
#define TST 12
#define MOV 13

FILE * atom_file_ptr;
ADDRESS avail = 0, end_data = 0;
int err_flag = FALSE; /* has an error been detected? */

```

The header file `mini.h` is shown below:

```

#define MaxMem 0xffff
#define TRUE 1
#define FALSE 0

/* Op codes are defined here: */
#define CLR 0
#define ADD 1
#define SUB 2
#define MUL 3

```

```
#define DIV 4
#define JMP 5
#define CMP 6
#define LOD 7
#define STO 8
#define HLT 9

/* Memory word on the simulated machine may be treated as
   numeric data or as an instruction */
union { float data;
        unsigned long instr;
        } memory [MaxMem];

/* careful!  this structure is machine dependent! */
struct fmt
{ unsigned int s2:    20;
  unsigned int r1:    4;
  unsigned int cmp:   3;
  unsigned int mode:  1;
  unsigned int op:    4;
}
;

union {
  struct fmt instr;
  unsigned long full32;
} ir;

unsigned long reg[8];
union { float data;
        unsigned long instr;
        } fpreg[8];
```