**SIM1 to SIM5 PROGRAM DOCUMENTATION**

1. **SIM1 Documentation**

1.1 Global Declarations

The program **sim1.c** begins three comment lines followed by two #include statements.

/\* sim1.c – Implements the six instructions for the SIM1 computer \*/

/\* Last modified on july 1, 2012 \*/

/\* usage “sim1 < test.sim1”, up to 100 cycles with trace \*/

#include <stdio.h> /\* printf() function available \*/

#include <stdlib.h> /\* exit() function declared \*/

The C statement **#include <stdio.h>** is similar to the Java statement **import java.lang.System;** and

makes the prewritten standard input/output functions including **printf()** available to the C programmer. Similarly, including **stdlib.h** makes the standard library functions including **exit()** available.

Next, ten **#define** statements are used to define ten symbolic constants.

#define MEMSIZE 1000 /\* 1000 words of memory \*/

#define WORDSIZE 4 /\* each word is 4 digits \*/

#define WORDLIMIT 9999 /\* change this if you change WORDSIZE \*/

#define MAXCNT 100 /\* limit execution of 100 instructions \*/

/\* SIM1 operation codes with format naaa where \*/

/\* aaa is a 3-digit address and n is as follows: \*/

#define HALT 0 /\* halt processor \*/

#define LD 1 /\* load accumulator instruction \*/

#define ST 2 /\* store accumulator instruction \*/

#define ADD 3 /\* add (to accumulator) \*/

#define SUB 4 /\* subtract (from accumulator) \*/

#define LDA 5 /\* load address \*/

The C statement **#define MEMSIZE 1000** has the same effect as the Java statement

**static final int MEMSIZE = 1000;**. The C convention is to use capital letters for such symbolic constants. MEMSIZE, WORDSIZE, and WORDLIMIT describe the memory of the SIM1 computer and MAXCNT limits the number of machine language instructions executed to catch programming errors that result in infinite loops. The remaining lines define the six operation codes used by the SIM1 computer.

If you are a Java programmer, think of a C program as consisting of a single static class with an “invisible” class statement. The C program consists of a sequence of functions (methods) and variable declarations (field declarations). They are analogous to static class methods and class fields in Java. Like class fields in Java, the C variables declared outside of any function are “public” variables that are accessible by any function unless the function “overrides” the external declaration with its own declaration. Because there is only a single (invisible) static class, there is no nesting of classes or inheritance.

The next lines in the C program contain function prototypes for all of the functions in the C program except for the **main()** function (which is required and known to the C compiler).

Void panic(char \*pmessage); /\* found impossible condition, C code broken \*/

void trace(); /\* output a trace line \*/

void readcode(); /\* input a machine language program \*/

void fetch(); /\* fetch a machine language instruction \*/

void execute(); /\* execute a machine language instruction \*/

The declaration **void panic(char \* pmessage);** specifies that function **panic()** accepts a pointer to a character as its argument (really a pointer to a character string) and does not return a result. The primary function of these prototypes is documentation and error checking. In C they are optional but in CIS 2107, they are mandatory.

Next come the data declarations that occur outside of any function. These statements reserve space for **“public”** variables that are accessible from any function.

Int memory[MEMSIZE];

n tip, acc, inst, cnt;

int digit1, digit2, digit3, digit4, digit234;

The variables **memory**, **ip**, and **acc** represent the SIM1 memory and the processor registers. The variable **inst** contains the most recently fetched instruction, **cnt** is a running total of the number of instructions executed, and the “digit” variables contain the individual digits of an instruction.

1.2 The function main()

The **main()** function where execution begins follows the declaration of the external variables and functions. With the exception of the **printf()** and **exit()** statements, the C implementation is identical to a Java implementation.

Int main()

{

int i; /\* local variable known only in main() \*/

/\* Initialize memory and processor registers \*/

for (i = 0; i < MEMSIZE; i++) {

memory[i]=0;

}

ip = 0; /\* instruction pointer contains address of next instruction \*/

acc = 0; /\* accumulator is where all arithmetic happens \*/

cnt = 0; /\* number of instructions executed so far \*/

readcode(); /\* input a SIM2 machine language program \*/

/\* Main loop – fetch next instruction, decode, and execute \*/

while (cnt++ < MAXCNT) { /\* limit execution to MAXCNT instructions \*/

fetch();

execute();

} /\* end of while loop \*/

/\* Normal exit is via HALT instruction, only reach here if cnt >= MAXCNT \*/

printf(“Processor executed more than %d instructions\n”, MAXCNT);

trace();

exit(1);

} /\* end of main \*/

The main program initializes the SIM1 memory and processor registers (**memory**, **ip**, **acc**) and sets the count of instructions executed (**cnt**) to zero. Next function **readcode()** is called to read a machine language program into memory and to set **ip** to the starting address of the program just read.

The **while** statement implements the processor “fetch-execute” cycle in which successive machine language instructions are fetched from memory and then executed. Usually, the **while** loop never terminates because a SIM **halt** instruction (opcode 0000) is executed and the C program terminates inside function **execute()**. However, if more than **cnt** instructions are executed, the program drops out of the while loop, an error

message is printed, function **trace()** prints the processor state, and library function **exit(1)** terminates execution with a return code of 1.

1.3 The function readcode()

Function **readcode()** reads successive line that are assumed to contain an address (000 to 999) followed by the contents of the address (0000 to 9999). If a line contains only an address, it is assumed to be the starting address of the SIM1 program.

/\* function to input a machine language program \*/

void readcode()

{

#define MAXLINE 80

int addr, value, items;

char line[MAXLINE];

while (1)

{

if ( (fgets(line, sizeof(line), stdin) ) <= 0) { /\* read until EOF \*/

printf(“End of file encountered, exiting\n”);

exit(1);

}

printf(“%s”, line); /\* print all lines \*/

if (line[0] == ‘#’) { /\* ignore comment \*/

continue;

}

items = sscanf(line, “%d %d” , &addr, &value); /\* up to two ints \*/

switch (items) {

case 0: /\* illegal line, print and ignore \*/

printf(“illegal input ignored\n”);

continue;

case 1: /\* one number is starting address \*/

if ( (addr < 000) || (addr > 999) ) {

printf(“illegal starting address %d, exiting\n”, addr);

exit(1);

}

else {

ip = addr; /\* so initialize ip and return \*/

printf(“Starting execution of SIM program at address %3.3d\n”, ip);

return;

}

case 2: /\* two numbers, address and value \*/

if ( (addr < 000) || (addr > 999) ) {

printf(“illegal memory address %d, exiting\n”, addr);

exit(1);

}

if ( (value < 000) || (value > 9999) ) {

printf(“illegal memory value %d, exiting\n”, value);

exit(1);

}

memory[addr]=value; /\* place value in memory \*/

} /\* end switch \*/

} /\* end of while \*/

} /\* end of readcode() \*/

The library function fgets() reads the next line of input into the **char** (**byte**) array **line**. If **line()** returns -1, it signifies “end of file” and the program exits. Comment lines that begin with “#” are ignored. Library function **sscanf()** reads up to two integers from **line,** placing them in addresses **addr** and **value**, and returning the number of integers successfully read. Lines containing no integers (**case 0**☺ are ignored. A single integer on a line should be the starting address of the program, so the instruction pointer (**ip**) is initialized and **readcode()** returns. If a line contains two integers, the first integer should be an address and the second should be the value, and memory is initialized (**memory[addr]=value**;).

1.4 Functions panic() and trace()

Function **trace()** simply the current values of the variables **cnt**, **ip**, **inst**, **acc** and returns. Function **panic()** displays an error message, calls **trace()**, and terminates execution by calling the library function **exit()**.

1.5 function **fetch()**

The **fetch()** function fetches the next instruction from the memory cell specified by the instruction pointer (**ip**). For debugging, a **printf()** function displays the current values of the instruction pointer (**ip**), the accumulator (**acc**), the instruction just fetched (**inst**) along with the total number of machine language instructions executed to this point (**cnt**).

The next statement increments the instruction pointer so that it points to the next location in memory. However, if **ip** were 999, **ip** would be incremented to 1000 (and there is no memory cell 1000). The statement **ip=ip%MEMSIZE** uses the C modulo operator (%) to compute **ip** modulo 1000.

Next, the four digits of the instruction are broken into parts (**digit1**, **digit2**, **digit3**, **digit4**, and **digit234**) using integer division and the modulo (**%**) operator. If the instruction 5678 is fetched from memory, **digit1** through **digit4** will equal 5, 6, 7, and 8 respectively and **digit234** will equal 678.

1.6 function **execute()**

A switch statement is used to process the operation code (the value in **digit1**). Recall that the **#define** statement was used to define the symbolic constant HALT as 0 so that the statement **case HALT:** is equivalent to

**case 0:**.

When the HALT case is executed, the C code prints a terminating message and the program exits. The code for the remaining cases (LD, ST, ADD, SUB, and LDA) is the same code that would be used for a Java implementation. With an ADD instruction, it is possible for the resulting sum to exceed the largest number that will fit in a SIM1 memory cell (9999). Subtracting 10,000 corrects this condition. Similarly, a subtract instruction could produce a number less than zero and adding 10,000 corrects this condition.

/\* Process opcodes 0 to 7 with switch statement \*/

switch ( digit1 ) {

case HALT:

printf(“Processor executed HALT instruction\n”);

printf(“cnt = %4.4i, ip = %4.4i, inst = %4.4i, acc = %4.4i\n”,

cnt, ip, inst, acc);

exit(0);

case LD:

acc = memory[digit234];

break;

case ST:

memory[digit234] = acc;

break;

case ADD:

acc = acc + memory[digit234];

if (acc > WORDLIMIT) /\* wrap if acc > 9999 \*/

acc = acc – (WORDLIMIT + 1); /\* by subtracting 10,000 \*/

break;

case SUB:

acc = acc – memory[digit234];

if (acc < 0) /\* wrap if acc < 0 \*/

acc += (WORDLIMIT+1);

break;

case LDA:

acc = digit234;

break;

default: /\* 6xxx to 9xxx instructions are not implemented yet \*/

printf(“Illegal operation code, instruction %i \n”, inst);

exit(1);

} /\* end of switch statement \*/

**2.0 SIM2 Documentation**

The new SKIP and JUMP instructions are implemented by adding two cases to the **switch** statement in function execute()..

/\* Process opcodes 0 to 7 with switch statement \*/

switch ( digit1 ) {

case HALT:

…

case JMP:

ip = digit234;

break;

case SKIPSET:

skipop(); /\* call skipop() to implement skip instructions \*/

break;

default: /\* 8xxx and 9xxx instructions are not implemented \*/

…

} /\* end of switch ( digit1 ) \*/

The implementation of the **jmp** instruction simply sets **ip** equal to digit234. The function **skipop()** (SKIP Operation) implements the various skip operation codes with a **switch** statement.

Void skipop()

{

…

switch ( digit2 ) {

case SKIP: /\* unconditional skip \*/

ip = ip + 1;

break;

case SEQ: /\* skip if acc equals 0 \*/

if (acc == 0) ip = ip + 1;

break;

case SNE: /\* skip if acc not equal 0 \*/

if (acc != 0) ip = ip + 1;

break;

case SGT: /\* skip if acc greater than 0 \*/

if (acc > 0 && acc < 5000) ip = ip + 1;

break;

…

}

ip = ip % MEMSIZE; /\* so that address 000 follows address 999 \*/

return; /\* finished processing legal skip instruction \*/

} /\* end of skipop() \*/

Recall that **ip** is incremented every time an instruction is fetched. If a skip instruction increments **ip** a second time, the instruction following the skip instruction will be “skipped”. Notice the use of the modulo operator (%) to insure that IP never exceeds 999.

**3.0 SIM3 Program Documentation**

The SIM3 machine language simulator adds instructions that use or modify the contents of the accumulator. IN and OUT instructions allow four digits numbers to be read or printed. CLR, INC, DEC, and NEG instructions perform the obvious functions of setting to zero, incrementing, decrementing, or negating the accumulator. The SHFTL (SHiFT Left) instruction multiplies the number in the accumulator by 10, while SHIFTR divides the accumulator by 10. As shown below, these accumulator instructions have the format 8n00, where "8" indicates an instruction in the accumulator group, and **n** specifies a particular accumulator instruction.

#define ACCSET 8 /\* accumulator instructions \*/

/\* accumulator operation codes (format 8n00, where n is as follows) \*/

#define IN 0 /\* input a 4-digit number into acc \*/

#define OUT 1 /\* output the 4-digit number from acc \*/

#define CLR 2 /\* clear the acc (acc = 0) \*/

#define INC 3 /\* increment (add 1) to the acc \*/

#define DEC 4 /\* decrement (subtract 1) from the acc \*/

#define NEG 5 /\* negate the number in acc \*/

#define SHFTL 6 /\* shift acc left (acc = acc \* 10) \*/

#define SHFTR 7 /\* shift acc right (acc = acc / 10) \*/

The implementation of these instructions is similar to the implementation of the skip instructions. A separate function (**accop()** - accumulator operation) is declared to handle the accumulator operations.

/\* function declarations \*/ 15

/\* function declarations \*/

void panic(char \*pmessage); /\* found impossible condition, C code broken \*/

void trace(); /\* output a trace line \*/

void readcode(); /\* input a machine language program \*/

void fetch(); /\* fetch a machine language instruction \*/

void execute(); /\* execute a machine language instruction \*/

void skipop(); /\* process skip operation codes \*/

void accop(); /\* process accumulator operation codes \*/

The **switch** statement in the execute() function is extended to test for an operation code that begins with 8 (ACCSET) and then call the "accumulator operation" function.

/\* Process opcodes 0 to 8 with switch statement \*/

switch ( digit1 ) {

case HALT:

...

case SKIPSET:

skipop();

break;

case ACCSET:

accop();

break;

default: /\* should never get here \*/

...

} /\* end of switch ( digit1 ) \*/

The implementation of the **accop()** function is mostly straight forward. The IN, INC, and DEC instructions must insure that the result is in the range 0 to 9999 and the IN instruction must test for end-of-file.

/\* accumulator operation codes (format 8n00) where n specifies the \*/

/\* particular instruction. \*/

void accop()

{

if (digit3 != 0 || digit4 != 0) {

printf("Illegal accumulator instruction %i \n", inst);

trace();

exit(1);

}

switch ( digit2 ) {

case IN: /\* input a four digit number to acc \*/

printf("Input a 4-digit number - ");

if (scanf("%i", &acc) < 1) {

printf("\nEOF on input, exiting\n");

trace();

exit(1);

}

if ( (acc < 0) || (acc > WORDLIMIT) ) {

printf("\nIllegal input value of %d, exiting\n", acc);

trace();

exit(1);

}

printf("%d\n", acc);

break;

case OUT: /\* output a four digit number from acc \*/

printf("output from program - %4.4i\n", acc);

break;

case CLR: /\* clear acc \*/

acc = 0;

break;

case INC: /\* increment acc \*/

acc = (acc + 1) % (WORDLIMIT+1); /\* increment and wrap \*/

break; /\* if acc > 9999 \*/

case DEC: /\* acc--, e.g. (1234+9999) mod 1000 = 1233 \*/

acc = (acc + WORDLIMIT) % (WORDLIMIT + 1);

break;

case NEG: /\*--acc, e.g. (10000-9999) mod 10000 = 0001 \*/

acc = (WORDLIMIT+1) - acc; /\*(10000-0000) mod 10000 = 0000 \*/

acc = acc % (WORDLIMIT + 1); /\*(10000-0001) mod 10000 = 9999 \*/

break; /\*(10000-4999) mod 10000 = 5001 \*/

case SHFTL: /\* shift left (or multiply by 10) \*/

acc = (acc \*10) % (WORDLIMIT+1);

break;

case SHFTR: /\* shift right (or multiply by 10) \*/

acc = acc / 10;

break;

default:

printf("Illegal accumulator instruction %i \n", inst);

trace();

exit(1);

} /\* end of "8n00" switch \*/

return; /\* finished processing legal acc instruction \*/

} /\* end of accop()

The NEG instruction simply subtracts the value in the accumulator from 10,000. As the following table shows, this gives the correct signed result for all valid accumulator values except 0. The instruction **acc=acc%(WORDLIMIT+1)** fixes this problem. Notice that if one negates 5000 (which is interpreted as the signed number -5000), the result is 5000 (still interpreted as the signed number -5000).This, of course, is incorrect. The largest possible signed number is 4999, so negating 5000 (interpreted as -5000) produces an incorrect result (which the SIM computer just ignores).

acc signed value 10,000 - acc signed result

0 0 10000 illegal

1 1 9999 -1

2 2 9998 -2

3 3 9997 -3

... ... ... ...

4998 4998 5002 -4998

4999 4999 5001 -4999

5000 -5000 5000 -5000 (error)

5001 -4999 4999 4999

5002 -4998 4998 4998

... ... ... ...

9998 -2 2 2

9999 -1 1 1

**4.0 SIM4 Program Documentation**

Creating the SIM4 general register processor from the SIM3 processor is straight forward. Instead the variables **ip** and **acc**, SIM4 uses an array of ten integers **regs[10]** where **regs[0]** now serves as the accumulator **regs[9]** is now the instruction pointer. All 10 elements of the array are initialized to zero. Throughout SIM4, all occurrence of the variables **acc** and **ip** in the earlier SIM versions are replaced by **regs[0]** and **regs[9].**

int regs[10];

...

/\* Initialize memory and processor registers \*/

for (i = 0; i < MEMSIZE; i++) {

memory[i]=0;

}

for (i = 0; i < 10; i++) {

regs[i]=0;

}

In the **execute()** function, in addition to replacing **acc** and **ip,** the accumulator group of instructions is renamed as the one-register (ONEREG) family of instructions, and a case is added to the switch statement for the new two-register (TWOREG) family of instructions.

switch ( digit1 ) {

...

case LD: /\* load %r0 \*/

regs[0] = memory[digit234];

break;

case ST: /\* store %r0 \*/

memory[digit234] = regs[0];

break;

case ONEREG: /\* one register inst \*/

onereg();

break;

case TWOREG: /\* two register inst \*/

tworeg();

break;

default: /\* should never get here \*/

printf("Illegal operation code, instruction %i \n", inst);

trace();

exit(1);

} /\* end of switch ( digit1 ) \*/

In function **skipop()**, the **if** statements must be changed to test a general register rather than just the accumulator. If the test is successful, the variable **regs[9]** rather than **ip** is incremented.

void skipop()

{

...

reg = regs[digit4]; /\* register we are testing \*/

switch ( digit2 ) {

case SKIP: /\* unconditional skip \*/

break;

case SKEQ: /\* skip if acc equals 0 \*/

if (reg == 0) regs[9] = regs[9] + 1;

break;

...

} /\* end of "8n0x" switch \*/

...

} /\* end of skipop() \*/

A similar modification must be made in function **onereg()** (the new name for function **accop())** to accommodate the new processor registers.

reg = regs[digit4];

switch ( digit2 ) {

...

case CLR: /\* clear (zero) reg \*/

regs[digit4] = 0;

break;

case INC: /\* increment reg \*/

regs[digit4] = (reg + 1) % (WORDLIMIT+1);

break;

case DEC: /\* reg--, e.g. (1234+9999) mod 1000 = 1233 \*/

regs[digit4] = (reg + WORDLIMIT) % (WORDLIMIT + 1);

break;

case NEG: /\* -reg, e.g. (10000-9999) mod 10000 = 0001 \*/

reg = (WORDLIMIT+1) - reg; /\* (10000-0000) mod 10000 = 0000 \*/

regs[digit4] = reg % (WORDLIMIT + 1);

break;

...

} /\* end of "8n0r" switch \*/

In SIM4 (as well as SIM5) the registers **regs[7]**, **regs[8], and** **regs[9]** hold 3-digit addresses rather than 4-digit numbers. Since these registers may have been modified in **onereg()**, statements are included keep the values within range.

regs[7] = regs[7] % MEMSIZE; /\* make sure %sp has legal address \*/

regs[8] = regs[8] % MEMSIZE; /\* make sure %lk has legal address \*/

regs[9] = regs[9] % MEMSIZE; /\* make sure %ip has legal address \*/

The function **regop()** is added to process operation codes in the two-register family. The variables **sreg** and **dreg** specify the source and destination register. For the MVMR and MVRM instructions, a validity check is performed on the register containing an address. The implementation of the SUBR (SUBtract Register) instruction implements the subtraction without allowing **dreg** to go negative although the resulting value may exceed 9999. The mod operator “%” corrects this situation.

Finally, the values in registers **regs[7]**, **regs[8], and** **regs[9]** are checked just as they were in **onereg()**.

/\* Two register operation codes (format 9nsd) \*/

void tworeg()

{

int sreg, dreg, temp; /\* local variables \*/

sreg = regs[digit3];

dreg = regs[digit4];

switch ( digit2 ) {

case MVRR: /\* Rd = Rs \*/

regs[digit4] = sreg;

break;

case MVMR: /\* Rd = memory[Rs] \*/

if (sreg > 999) {

printf("Illegal memory reference via register %d, exiting\n", digit3);

trace();

exit;

}

regs[digit4] = memory[sreg];

break;

case MVRM: /\* memory[Rd] = Rs \*/

if (dreg > 999) {

printf("Illegal memory reference via register %d, exiting\n", digit3);

trace();

exit;

}

memory[dreg] = sreg;

break;

case EXCH: /\* exchange Rs and Rd \*/

regs[digit3] = dreg;

regs[digit4] = sreg;

break;

case ADDR: /\* Ry = Ry + Rx \*/

dreg = dreg + sreg;

regs[digit4] = dreg % (WORDLIMIT + 1); /\* wrap if reg > 9999 \*/

break;

case SUBR: /\* Ry = Ry - Rx \*/

temp = (WORDLIMIT+1) - sreg; /\* temp = -sreg \*/

dreg = dreg + temp; /\* \*/

regs[digit4] = dreg % (WORDLIMIT+1); /\* wrap if regy > 9999 \*/

break;

default:

printf("Illegal register instruction %i \n", inst);

trace();

exit(1);

} /\* end of "9nxy" switch \*/

regs[7] = regs[7] % MEMSIZE; /\* make sure %sp has legal address \*/

regs[8] = regs[8] % MEMSIZE; /\* make sure %lk has legal address \*/

regs[9] = regs[9] % MEMSIZE; /\* make sure %ip has legal address \*/

return; /\* finished processing two register instructions \*/

} /\* end of tworeg() \*/

**5.0 SIM5 Program Documentation**

Your assignment.