**THE SIMPLE COMPUTER (SIM)**

**0.0 The SIM PROCESSOR**

**0.1 INTRODUCTION**

The C program described here simulates a very simple computer (SIM) that is designed to illustrate features of computer architecture and machine language. SIM consists of a central processing unit (CPU) or processor, a memory, and a console of some kind that allows an operator to change and display the content of memory locations and processor registers.

Memory consists of 1000 memory cells, each of which contains a 4-digit decimal number called a word. The processor also contains storage locations called processor registers or simply registers. The programmer of the SIM computer must be aware of two registers The first is called the accumulator (abbreviated **acc**). It contains a 4-digit decimal number and is analogous to the display window in a calculator. The second register is called the instruction pointer (abbreviated **ip**). **ip** contains a 3-digit decimal number that specifies the address of the next instruction to be executed.

+-------------CPU-------------+

| |

| +------+ +-----+ |

| acc | 0000 | ip | 000 | |

| +------+ +-----+ |

| |

+-----------------------------+

/ \

/| |\

| |

| |

\| |/

\ /

+----------------+

| |

| addr cont |

| +------+ |

| 000 | 1004 | |

| +------+ |

| 001 | 3005 | |

| +------+ |

| 002 | 2006 | |

| +------+ |

| 003 | 0000 | |

| +------+ |

| 004 | 5555 | |

| +------+ |

| 005 | 0234 | |

| +------+ |

| 006 | 0000 | |

| +------+ |

| ... ... |

| +------+ |

| 998 | 0000 | |

| +------+ |

| 999 | 0000 | |

| +------+ |

| |

+-----MEMORY-----+

Memory is a passive device that responds to fetch and store requests from the processor. To perform a fetch, the processor sends memory a 3-digit address (e.g. 005) and memory responds by sending the word at that address (e.g. 0234) back to the processor. The content of memory is not changed. To perform a store, the processor sends memory an address (e.g. 006) and a new contents (e.g. 5789). The new contents (5789) replaces the old contents (0000).

**0.2 PROGRAM EXECUTION**

The processor executes programs by blindly performing the following steps.

1. Using the instruction pointer (**ip**), fetch the next instruction from memory.
2. Add 1 to **ip** (so that instructions are fetched from consecutive memory cells).
3. Execute the instruction fetched in step 1.
4. Go back to step 1 to repeat the process.

Graphically:

+-------------------+ +---------------+ +----------------+

+--->| inst = memory[ip] |---> | ip = ip + 1 |--->| execute inst |--->+

^ +-------------------+ +---------------+ +----------------+ |

| v

+<------------------------------------------------------------------------+

The processor continues fetching and executing instructions until it encounters a **halt** instruction or an illegal (unimplemented) instruction.

In the example above, **ip** initially contains 000, and if the computer is started, the processor will execute the instruction in address 000 (1004) followed by the instructions in addresses 001 (3005), 002 (2006) and 003 (0000). Because 0000 is the **halt** instruction, the processor will halt after executing the instruction in address 003.

We will trace the execution of this program in more detail beginning at address 000. The instruction in address 000, namely 1004, is an example of a **ld** (for load accumulator from memory) instruction. Load instructions have the format 1xxx, where xxx specifies the address containing the word to be loaded into the accumulator. In this case, the address is 004, so the instruction 1004 will load the number in address 004, namely 5555, into the accumulator. At this point, the processor registers appear as follows. (Note that **ip** has been incremented to 001.)

+------+ +-----+

acc | 5555 | ip | 001 |

+------+ +-----+

Since **ip** now contains 001, the processor will fetch the instruction (3005) from address 001 and increment **ip** to 002. The instruction 3005 is an example of an **add** instruction with format 3xxx. In this case, xxx is 005, so the processor will add the contents of address 005 (namely 0234) to the accumulator, producing:

+------+ +-----+

acc | 5789 | ip | 002 |

+------+ +-----+

Since **ip** now contains 002, the processor fetches the instruction 2006 from address 002 and increments **ip** to 003. 2006 is an example of a **st** (for store accumulator into memory) instruction with format 2xxx. The processor stores the word 5789 into location 006, erasing the previous contents (0000). The processor registers now contain:

+------+ +-----+

acc | 5789 | ip | 003 |

+------+ +-----+

Next the processor fetches the **halt** instruction (0000) from address 003, increments **ip** to 004, and halts.

**0.3 FILE DESCRIPTIONS**

The SIM computer is introduced using a series of four C programs contained in the four files **sim1.c**, **sim2.c**, **sim3.c**, and **sim4.c**. The SIM1 program only implements simple data movement and arithmetic instructions, so only straight-line code can be implemented. SIM2 introduces jump and conditional branch instructions to implement loops and conditional statements. SIM3 introduces input and output instructions as well as additional data manipulation instructions. Finally, SIM4 replaces **ip** and **acc** with ten general registers and adds instructions to manipulate these registers. Each successive SIM version is backward compatible, so that SIM4 will execute any program written for the earlier SIM's. In summary, the C program files are:

* sim1.c In-line data movement and arithmetic
* sim2.c Adds jump and conditional branch instruction
* sim3.c Adds input, output, and additional data manipulation instructions
* sim4.c Converts sim3.c to a general register machine.

The file extensions .sim1, .sim2, .sim3, and .sim4 are used to identify SIM machine language programs designed to run on the corresponding SIM computer. The files are:

* addv1.sim1 Computes 20+30
* addv2.sim1 Computer 20+30 with assembly and high level language comments
* addv3.sim1 Initializes variables a and b and computes c = a + b
* addv4.sim1 Same as addv3.sim1 with program relocated to different addresses.
* loopv1.sim2 Computes 4+3+2+1 using a loop
* readv1.sim3 Inputs two numbers and prints the sum.
* readv2.sim3 Inputs n numbers and prints the sum.
* sum1to10.sim4 Sums numbers from 10 to 1 to illustrate use of the general registers
* reverse.sim4 Inputs n numbers and outputs in reverse order to illustrate arrays
* mult.sim inputs 2 numbers and outputs their product to illustrate functions
* addn.sim4 Uses registers to implement readv2.sim3 more efficiently.

The file **card.html** contains a 3-page SIM reference card that describes all of the features of the SIM1 through SIM4 computers. The file **card.docx** is a Microsoft Word version of the same document that fits on a single page. This document can be used during any quiz or examination. Finally, the current Microsoft Word document is available as simwritup.docx or as four smaller text files **sim1.txt**, **sim2.txt**, **sim3.txt**, and **sim4.txt**.

* [sim1.txt](http://develop.temple.edu/stafford/cis2107f11/sim/sim1/sim1.txt) - text file describing the SIM1 computer
* [sim2.txt](http://develop.temple.edu/stafford/cis2107f11/sim/sim2/sim2.txt) - text file describing the SIM2 computer
* [sim3.txt](http://develop.temple.edu/stafford/cis2107f11/sim/sim3/sim3.txt) - text file describing the SIM3 computer
* [sim4.txt](http://develop.temple.edu/stafford/cis2107f11/sim/sim4/sim4.txt) - text file describing the SIM4 computer
* [simwriteup.docx](http://develop.temple.edu/stafford/cis2107f11/sim/simwriteup.docx) - Microsoft word description of SIM1 to SIM4
* [card.html](http://develop.temple.edu/stafford/cis2107f11/sim/card.html) - SIM reference card, HTML version
* [card.docx](http://develop.temple.edu/stafford/cis2107f11/sim/card.docx) - SIM reference card, Microsoft Word version

**1.0 THE SIMPLE COMPUTER VERSION 1 (SIM1)**

**1.1 A SAMPLE PROGRAM**

The SIM1 simulator is a C program that executes programs written in the SIM1 machine language. To create a machine language program, the SIM1 programmer creates a data file such as the following file **addv1.sim1** (add version 1, sim1 machine language program).

addv1.sim1 - Add 20 to 30 and store the sum in address 006

000 1004 Load the accumulator with the word at 004

001 3005 Add to the accumulator the word at 005

002 2006 Store the accumulator in address 006

003 0000 Halt

004 0020 Address 004 initialized to 20

005 0030 Address 005 initialized to 30

006 0000 Address 006 initialized to 0

000 Execution starts at 000

Because the first line does not begin with a number, it is assumed to be a comment. The second line specifies that address 000 should be initialized to 1004. The next six lines specify the initial contents of addresses 001 to 006. The comments following the numbers are ignored when the program is loaded into the SIM1 memory. Because the last line only begins a single number, it specifies the address where execution is to begin and marks the end of the file (and the program).

The file **sim1.c** is the C program that simulates a SIM1 computer. This source file is compiled into an executable file called **a.out** using the C compiler driver **gcc** (for GNU C Compiler). The following Unix command is used to compile **sim1.c**. The string **cis-lclient02:~>** is the prompt used by the Unix computer called **cis-lclient02** and the command **gcc sim1.c** requests that the **gcc** program be executed using the file **sim1.c** as input.

cis-lclient02:~>gcc sim1.c

This creates an executable version of the **sim1.c** program in the file **a.out** (the default name for executable files created by **gcc**.

The following command is used to execute the SIM1 program stored in a file **addv1.sim1.**

cis-lclient02:~>./a.out < addv1.sim1

The command **./a.out < addv1.sim1** requests that the file **./a.out** (the **a.out** file in the current directory) be executed using the file **addv1.sim1** (rather than the keyboard) as input. The result is shown below.

cis-lclient02:~>gcc sim1.c

cis-lclient02:~>./a.out < addv1.sim1

addv1.sim1 - Add 20 to 30 and store the sum in address 006

000 1004 Load the accumulator with the word at 004

001 3005 Add to the accumulator the word at 005

002 2006 Store the accumulator in address 006

003 0000 Halt

004 0020 Address 004 initialized to 20

005 0030 Address 005 initialized to 30

006 0000 Address 006 initialized to 0

Execution at address 000

Starting execution of SIM program at address 000

cnt = 1, ip = 000, inst = 1004, acc = 0000

cnt = 2, ip = 001, inst = 3005, acc = 0020

cnt = 3, ip = 002, inst = 2006, acc = 0050

cnt = 4, ip = 003, inst = 0000, acc = 0050

Processor executed HALT instruction

cnt = 4, ip = 004, inst = 0000, acc = 0050

superman:>

The SIM1 simulator displays the contents of the input file (**addv1.sim1**) as well as the starting address of the program (000). As an aid to debugging, the SIM1 simulator program prints a line each time a machine language statement is executed, including a count of the total number of machine language statements encountered to this point (**cnt**) as well as the values of **ip** and **acc** before the instruction (**inst**) is executed. For example, the

line:

cnt = 3, ip = 002, inst = 2006, acc = 0050

shows that, just before the third instruction was executed, the instruction stored at address 002 was 2006, and the accumulator contained 0050 (sum of 20 plus 30).

**1.2 MACHINE LANGUAGE , ASSEMBLY LANGUAGE, AND HIGH LEVEL LANGUAGES**

When electronic digital computers were developed during the late 1940's and early 1950's, programmers created programs using machine language. It can be difficult for programmers to remember the function of the various operations codes (e.g. 1xxx, 2xxx, etc.) as well as the function of data stored at particular addresses (e.g. memory cells 004, 005, and 006 in the program above).

Assembly languages were developed in the early 1950's to make programming easier. In assembly language programs, the numeric operation codes and address in a machine language program are replaced with symbolic operation codes and symbolic addresses. For example, the listing below displays the contents of the file **addv2.sim1**, the second version of our addition program. Functionally, this program is identical to the **addv1.sim1** program. However, the comments have been changed to illustrate an assembly language and a high

level language version of the program.

The comments under the heading "ASSEMBLY LANGUAGE" show an assembly language version of the addition program. The operation codes **0000, 1xxx, 2xxx, and 3xxx** have been replaced by the symbols **add,** **halt**, **ld**, and **st**. The addresses 000, 004, 005, and 006 have been replaced with the symbols **start**, **a**, **b**, and **c**. The symbol **.word** is an assembly directive that tells the assembler to reserve space in memory for a word of data and to initialized the word to a specified value. The assembly directive **.end** specifies the address at which execution should begin (in this case address 000).

ASSEMBLY LANUGAGE HIGH LEVEL LANGUAGE

000 1004 start: ld a a = b + c;

001 3005 add b

002 2006 st c

003 0000 halt exit(0);

004 0020 a: .word 20 int a = 20;

005 0030 b: .word 30 int b = 30;

006 0000 c: .word 0 int c = 0;

000 .end start

A program called the assembler inputs the assembly language program and outputs an equivalent machine language program that can be executed by the computer hardware. The assembler translates the assembly language program into machine language by substituting numbers for names using two tables: the symbol table and the opcode table.

The Symbol Table The Opcode Table

symbolic numerical symbolic numerical

address address opcode opcode

a 004 halt 0000

b 005 ld 1xxx

c 006 st 2xxx

start 000 add 3xxx

sub 4xxx

lda 5xxx

The opcode table is "built in" to the assembler. The assembler creates the symbol table by assigning successive assembly language statements to successive addresses in memory. The translation from assembly language to machine language is "one to one" in that each assembly language instruction generates a single machine language instruction.

In the middle to late 1950's, high-level languages similar to the **C** language were developed. The comments under "HIGH LEVEL LANGUAGE" in the file listing above illustrate how the **sumv2.sim1** machine language program might be implemented in a high-level language. As the example shows, a single statement in a high-level language (**a = b + c;)** may be translated into several assembly and machine language statements (one to many). In addition to making programming easier, higher level languages make programs transportable between different "brands" of computers. Machine and assembly language programs will only run on a particular computer architecture, while a high level language program can be translated (by different translators or compilers) to run on any computer architecture. We will comment our machine language using both the assembly language and the high level language

These languages types are still used today. For example, when the **gcc sim1.c** command is entered, the **gcc** "driver" executes the C preprocessor, the C compiler, the GCC assembler, and the linker, producing several temporary files as well as the **a.out** machine language file that can be executed.

+-------------+ +--------+ +---------+ +------+

-----> |preprocesseor| -----> |compiler| -----> |assembler| -----> |linker| ---->

sim1.c +-------------+ sim1.i +--------+ sim1.s +---------+ sim1.o +------+ a.out

When the command **./a.out** is entered, the Unix operating system invokes the loader which places the machine language file **a.out** into (virtual) memory and starts execution of your program.

+------+ +---------------------------+ 6

----> |loader| -----> | execution of your program |

a.out +------+ +---------------------------+

The relationships between the computer hardware, machine language, assembly language, and high level languages are the focus of this course. The Intel 32-bit and 64 bit architecture, GCC assembly language for these Intel processors, and the C programming language will be used to illustrate these concepts.

**1.3 THE addv3.sim1 AND addv4.sim1 PROGRAMS**

The program addv3.sim1 illustrates the use of the LoaD Address (**lda)** instruction with format 5xxx.

addv3.sim1 = Initialize A and B, and compute C as the sum

MACHINE LANGUAGE ASSEMBLY LANGUAGE HIGH LEVEL LANGUAGE

000 5123 start: lda 123 a = 123;

001 2008 st a

002 5456 lda 456 b = 456;

003 2009 st b

004 1008 ld a c = a + b;

005 3009 add b

006 2010 st c

007 0000 halt exit(0);

008 0000 a: .word int a;

009 0000 b: .word int b;

010 0000 c: .word int c;

000 .end start

When an **lda** instruction with form 5xxx is executed, the number 0xxx is loaded into the accumulator. (Memory cell xxx is not involved.). For example, in **addv3.sim1**, the lines:

MACHINE LANGUAGE ASSEMBLY LANGUAGE HIGH LEVEL LANGUAGE

000 5123 start: lda 123 a = 123;

001 2008 st a

load the number 0123 into the accumulator and then stores this number into memory word 008. (The equivalent assembly language and high level language are shown along with the machine language.) **addtest3()** is similar to **addtest2()** except that the initial values of memory cells 008 and 009 (variables **a** and **b)** are initialized by machine language statements during the execution of the program.

The program **addv4.sim1** is the same as **addv3.sim1** except that the code and data have been moved to different locations in memory.

addv4.sim1 - Initialize A and B, and compute C as the sum

MACHINE LANGUAGE ASSEMBLY LANGUAGE HIGH LEVEL LANGUAGE

.=100

100 5123 start: lda 123 a = 123;

101 2200 st a

102 5456 lda 456 b = 456;

103 2201 st b

104 1200 ld a c = a + b;

105 3201 add b

106 2202 st c

107 0000 halt exit(0);

.=200

200 0000 a: .word int a;

201 0000 b: .word int b;

202 0000 c: .word int c;

100 .end start

In **addv3.sim1**, the code and data used memory cells 000 through 010, while addv4.sim1 uses 100 through 107 for code and 200 through 202 for data. In assembly language, the symbol "**.**" is called the location counter and it specifies the address in memory where the code generated from the "next" line is to be placed. Normally, the assembler initializes the location counter to 000 and increments the counter after each assembly language statement is processed so that code and data occupy successive locations in memory. As the **addv4.sim1** program illustrates, directly modifying the location counter provides the programmer with a very simple way to relocate (move) a program to another area of memory. Virtual memory (to be covered in CIS 3207) provides a much more powerful and elegant method of relocating programs.

**1.4 SIM1 PROGRAM OUTLINE**

1.4.1 Global Declarations

The program **sim1.c** begins with two **#include** statements:

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

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

The C statement **#include <stdio.h>** is similar to the Java statement **package stdlib.h;** 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 1000 /\* limit execution of 1000 instructions \*/

#define HALT 0 /\* halt the 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 "external" 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.

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

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

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

The declaration **void readcode()** specifies that function **readcode()** accepts no arguments 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 **global** variables that are accessible from any function.

int memory[MEMSIZE]; /\* 1000 words of memory \*/

int ip, acc; /\* processor registers \*/

int inst, cnt; /\* current instruction and limit counter \*/

int digit1, digit2, digit3, digit4, digit234; /\* instruction digits \*/

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.

1.4.2 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 SIM1 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 \*/

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

printf("cnt = %i, ip = %3.3i, inst = %4.4i, acc = %4.4i\n",

cnt, ip, inst, acc);

exit(0);

} /\* 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 reset **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 and an error

message is printed.

1.4.3 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. The **readcode()** function could be written as follows:

void readcode() {

int count, address, contents; /\* local variables \*/

while (1) { /\* loop forever \*/

count = scanf("%d %d\n", &address, &contents); /\* input two integers \*/

if (count == 2) { /\* if two integers found \*/

memory[address] = contents; /\* initialize memory cell \*/

}

else if (count == 1) { /\* if one integer found \*/

ip = address; /\* found starting address \*/

break; /\* break out of while loop \*/

}

else { /\* otherwise, input error \*/

exit(1); /\* exit program \*/

}

} /\* end of while statement \*/

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

Function **scanf()** (available because of the **#include <stdio.h>** directive) attempts to read two integers into variables (address and contents) and returns the number of integers successfully read. The C code in this function is identical to Java code except that:

1. **scanf()** is used for keyboard input rather than a Java class like Scanner.
2. In the **while** statement, "1" is used instead of the Java keyword **true**.
3. The **exit(1)** function is used to terminate the program if an input error occurs.

The version of **readcode()** used in **sim1.c** is more complicated to allow for program comments and additional error checking.

1.4.4 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.

1.4.5 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) so the result of the addition is taken modulo 10,000. Similarly, as subtract instruction could produce a number less than zero. In this case, an **if** statement is used to correct the result.

/\* 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 \*/

1.5 SIM1 QUESTIONS AND PROBLEMS

Assuming that execution begins at address 000, what value will the accumulator (**acc**) contain when each of the machine language programs executes a HALT instruction. (If the answers are not obvious, execute the programs with the "sim1.c" simulator.

1. In machine language, instructions can be used as data (usually a bad idea).

000 1000

001 3001

002 0000

000

2. The SIM memory words have no place for a + or - sign. However, numbers between 5000

and 9999 can be interpreted as negative numbers. If we were using the signed

interpretation, how should the number 9999 be interpreted?

000 1003

001 4004

002 0000

003 0050

004 0051

000

3. Program 3.

000 5002

001 2012

002 1012

003 3012

004 2012

005 1012

006 3012

007 2012

008 1012

009 3012

010 2012

000

4. Even short programs can be very confusing.

000 5002

001 3000

002 4002

003 2004

004 0000

005 0000

000

5. As program 2 illustrated, the number 9999 can play the role of -1 in the SIM1

machine language. The ADD instruction is implemented with the following code.

case ADD:

acc = acc + memory[digit234];

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

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

break;

As a result:

1 + 9999 = 0

2 + 9999 = 1

20 + 9999 = 19

500 + 9999 = 499

1. What number would play the role of -10 (so that adding this number to, for example, 50 would yield 40). (Hint - if you drove a new car backwards for a mile, the odometer (if it is mechanical) would go backwards from 000000 to 999999. What would happen if you went backwards for 10 miles?).
2. Can you give a simple formula for taking the negative of any SIM1 number between 1 and 4999.
3. What happens if you apply the same formula to numbers between 5001 and 9999?

6. Write assembly language programs equivalent to four machine language programs above.

7. Can you replace the **if** statement used in implementing the subtract instruction with additions, subtractions, and the modulo operator. ( **if** statements can significantly slow the execution of your program.)

**2.0 SIM2 WITH SKIP AND JUMP INSTRUCATIONS**

**2.1 INTRODUCTION**

The SIM1 simulator can only implement "straight line" code without loops or conditional statements. SIM2 adds seven instructions: a jump (**jmp**) instruction that allows transfer to an instruction located anywhere in memory and a family of six skip instructions which can conditionally skip over the following instruction (which is typically a **jmp** instructions. For example, the following code sets variable **a** (in memory cell 100) equal to the minimum of variables **b** (in 101) and **c** (in 102).

000 1101 ld b a = b; ;

001 2100 st a

002 4102 sub c if (b - c < 0)

003 8500 sklt {

004 7007 jmp done

005 1102 ld c a = c;

006 2100 st a }

007 ... done: ...

The first two instructions set **a** equal to **b**. Since the accumulator still contains the value of **b**, the **sub c** instruction leaves **b - c** in the accumulator. If the accumulator is greater than or equal to zero, the **sklt** (SKip if accumulator Less Than zero) does not skip, and the **jmp** (JuMP) instruction is executed. If the accumulator is less than zero, the **sklt** instruction in address 003 causes the **jmp** instruction in 004 to be skipped, and the instructions in address 005 and 006 are executed, setting **a** equal to **c**.

The code above is the typically use of the skip and jump instructions. If you want to jump somewhere when some condition is true, you precede the jump instruction with a skip instruction that skips the jump instruction when the condition is false.

The new opcodes are described at the beginning of the SIM2 program using additional "#define" statements.

#define JMP 6 /\* jump (branch) instruction \*/

#define SKIPSET 7 /\* skip instructions \*/

/\* skip operation codes (format 7n00 where n is as follows) \*/

#define SKIP 0 /\* unconditional skip \*/

#define SEQ 1 /\* skip if acc equals 0 \*/

#define SNE 2 /\* skip if acc not equal 0 \*/

#define SGT 3 /\* skip if acc greater than 0 \*/

#define SGE 4 /\* skip if acc greater or equal to 0 \*/

#define SLT 5 /\* skip if acc less than 0 \*/

#define SLE 6 /\* skip if acc less than or equal to 0 \*/

**2.2 IMPLEMENTATION OF JUMP AND SKIP INSTRUCTIONS**

The new instructions are implemented by adding cases to the **switch** statement in the main program.

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

switch ( digit1 ) {

case HALT:

...

case JMP:

ip = digit234;

break;

case SKIPSET:

skipop();

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".

**2.3 SIGNED AND UNSIGNED NUMBERS**

As shown in the table below, SIM numbers can be interpreted as either unsigned numbers (with a range from 0 to 9999) or as signed numbers (with a range from -5000 to 4999).

value in unsigned signed

memory interpretation interpretation

0000 0 0

0001 1 1

0002 2 2

... ... ...

4998 4998 4998

4999 4999 4999

5000 5000 -5000

5001 5001 -4999

5002 5002 -4998

... ... ...

9997 9997 -3

9998 9998 -2

9999 9999 -1

To see the logic behind the signed interpretation, consider a new car with an odometer reading of 000000. As you drove the car from the auto showroom, the odometer would register 000001, 000002, 000003, etc. to record the total mileage driven. But consider what would happen if you backed the car out of the showroom and drove home backwards. The odometer (or at least an old-style mechanical odometer) would register 999999, 999998, 999997, etc. to record the "negative" distance the car was driven. In this case, an odometer reading of 999999 represents -1, 999998 represents -2, 999997 represents -3,etc.

The rules for adding (and subtracting) signed numbers are identical to the rules for adding (and subtracting) unsigned numbers, so the SIM computer only needs a single ADD (and SUB) instruction. The skip instructions selected by the programmer determine the interpretation being used for a particular number. For unsigned numbers, only SKEQ and SKNE (which, for unsigned numbers, means greater than zero) are used, For signed numbers, all of the skip instructions can be used (but positive numbers are limited to the range 0 to 4999).

**2.4 SAMPLE PROGRAM LOOP1()**

The file **loopv1.sim2** contains a SIM2 program to sum the integers from 1 to 4. The machine language program uses a down-counting loop (4, 3, 2, 1, 0) because it uses fewer instructions than an up-counting loop. If the machine language program in **loopv1.sim2** is executed, the following results are produced.

superman>gcc sim2.c

superman>./a.out < loopv1.sim2

loopv1.sim2 - Sum the integers from 1 to 4 using a loop

000 5000 start: lda 0 sum = 0;

001 2015 st sum

002 5004 lda 4 for (cnt=4; cnt>0; cnt--)

003 2016 st cnt {

004 5001 lda 1

005 2017 st one

006 1015 loop: ls sum sum = sum + cnt;

007 3016 add cnt

008 2015 st sum

009 1016 ld cnt

010 4017 sub one

011 2016 st cnt

012 7100 skeq

013 6006 jmp loop }

014 0000 halt exit(0)

015 0000 sum: .word int cnt;

016 0000 cnt: .word int sum;

017 0000 one: .word

000 .end start

Starting execution of SIM program at address 000

cnt = 1, ip = 000, inst = 5000, acc = 0000

cnt = 2, ip = 001, inst = 2015, acc = 0000

cnt = 3, ip = 002, inst = 5004, acc = 0000

cnt = 4, ip = 003, inst = 2016, acc = 0004

cnt = 5, ip = 004, inst = 5001, acc = 0004

cnt = 6, ip = 005, inst = 2017, acc = 0001

cnt = 7, ip = 006, inst = 1015, acc = 0001

cnt = 8, ip = 007, inst = 3016, acc = 0000

cnt = 9, ip = 008, inst = 2015, acc = 0004

cnt = 10, ip = 009, inst = 1016, acc = 0004

cnt = 11, ip = 010, inst = 4017, acc = 0004

cnt = 12, ip = 011, inst = 2016, acc = 0003

cnt = 13, ip = 012, inst = 7001, acc = 0003

cnt = 14, ip = 013, inst = 6006, acc = 0003

cnt = 15, ip = 006, inst = 1015, acc = 0003

cnt = 16, ip = 007, inst = 3016, acc = 0004

cnt = 17, ip = 008, inst = 2015, acc = 0007

...

cnt = 30, ip = 013, inst = 6006, acc = 0001

cnt = 31, ip = 006, inst = 1015, acc = 0001

cnt = 32, ip = 007, inst = 3016, acc = 0009

cnt = 33, ip = 008, inst = 2015, acc = 0010

cnt = 34, ip = 009, inst = 1016, acc = 0010

cnt = 35, ip = 010, inst = 4017, acc = 0001

cnt = 36, ip = 011, inst = 2016, acc = 0000

cnt = 37, ip = 012, inst = 7001, acc = 0000

cnt = 38, ip = 014, inst = 0000, acc = 0000

Processor executed HALT instruction

cnt = 38, ip = 015, inst = 0000, acc = 0000

superman>

The line generated when **cnt** was equal to 33, namely:

cnt = 33, ip = 008, inst = 2015, acc = 0010

shows the sum of 4, 3, 2, and 1, namely 0010, being stored in address 015 (**sum**).

**2.5 SIM2 QUESTIONS AND PROBLEMS**

1. Fill in the missing values in the following table.

value in unsigned signed

memory interpretation interpretation

0010 \_ \_ \_ \_ \_ \_ \_ \_ \_

0200 \_ \_ \_ \_ \_ \_ \_ \_ \_

4000 \_ \_ \_ \_ \_ \_ \_ \_ \_

5000 \_ \_ \_ \_ \_ \_ \_ \_ \_

6000 \_ \_ \_ \_ \_ \_ \_ \_ \_

7000 \_ \_ \_ \_ \_ \_ \_ \_ \_

9000 \_ \_ \_ \_ \_ \_ \_ \_ \_

9900 \_ \_ \_ \_ \_ \_ \_ \_

9990 \_ \_ \_ \_ \_ \_ \_ \_

9999 \_ \_ \_ \_ \_ \_ \_ \_

2. Modify **loopv1.sim2** to compute the sum of the numbers from 1 to 50. Explain the answer computed by the program.

3. Modify **loopv1.sim2** to use an up-counting loop to sum the numbers from 1 to 10.

**3.0 SIM3 WITH ADDITIONAL ACCUMULATOR INSTRUCTIONS**

**3.1 INTRODUCTION**

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

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: /\* 9xxx instructions are not implemented \*/

...

} /\* 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.

/\* accumulator operation codes (format 8n00) \*/

void accop()

{

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

exit(1);

}

switch ( digit2 ) {

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

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

scanf("%i", &acc);

acc = acc%(WORDLIMIT+1)

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 \*/

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

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

break;

case DEC: /\* decrement acc \*/

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

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

break;

case NEG: /\* negate acc \*/

acc = (WORDLIMIT+1) - acc; /\* works for all but 0 \*/

acc = acc % (WORDLIMIT+1); /\* fix 0 \*/

break;

case SHFTL: /\* acc = acc \* 10 \*/

acc = acc \* 10;

acc = acc % (WORDLIMIT+1); /\* fix overflow \*/

break;

case SHFTR: /\* acc = acc / 10 \*/

acc = acc / 10;

break;

default:

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

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

**3.2 SAMPLE PROGRAMS**

The file **readv1.sim3** defines a machine language program that reads two data items and outputs the sum. In the file, the "data" follows the machine language program with one data item (and no comments) on each line.

readv1.sim3 Read 2 integers and print the sum \*/

000 8000 start: in read(a);

001 2009 st a

002 8000 in read(b);

003 2010 st b

004 1009 ld a sum = a + b;

005 3010 add b

006 2011 st sum

007 8100 out print(sum);

008 0000 halt exit(0);

009 0000 a: .word int a;

010 0000 b: .word int b;

011 0000 sum: .word int sum;

000 .end start

10

20

Running this program generates the following output.

superman>./a.out < readv1.sim3

readv1.sim3 Read 2 integers and print the sum \*/

000 8000 start: in read(a);

001 2009 st a

002 8000 in read(b);

003 2010 st b

004 1009 ld a sum = a + b;

005 3010 add b

006 2011 st sum

007 8100 out print(sum);

008 0000 halt exit(0);

009 0000 a: .word int a;

010 0000 b: .word int b;

011 0000 sum: .word int sum;

000 .end start

Starting execution of SIM program at address 000 17

cnt = 1, ip = 000, inst = 8000, acc = 0000

Input a 4-digit number - 10

cnt = 2, ip = 001, inst = 2009, acc = 0010

cnt = 3, ip = 002, inst = 8000, acc = 0010

Input a 4-digit number - 20

cnt = 4, ip = 003, inst = 2010, acc = 0020

cnt = 5, ip = 004, inst = 1009, acc = 0020

cnt = 6, ip = 005, inst = 3010, acc = 0010

cnt = 7, ip = 006, inst = 2011, acc = 0030

cnt = 8, ip = 007, inst = 8001, acc = 0030

output from program - 0030

cnt = 9, ip = 008, inst = 0000, acc = 0030

Processor executed HALT instruction

cnt = 0009, ip = 0009, inst = 0000, acc = 0030

superman>

The program in the file **readv2.sum3** reads n integers and prints the sum. The first **in** statement inputs a count of the number of data items (**n**). A down-counting loop is used to input the **n** data items and sum them. With the data items shown (3, 10, 20, 30), the program will output "60" when it is executed.

readv2.sim3 Read n integers and print the sum \*/

000 8000 start: in read(n);

001 2016 st n

002 8200 clr sum = 0;

003 2017 st sum

004 1016 loop: ld n for (; n>0; n--)

005 7200 skne {

006 6013 jmp done

007 8400 dec

008 2016 st n

009 8000 in sum = sum + read();

010 3017 add sum

011 2017 st sum

012 6004 jmp loop }

013 1017 done: ld sum print(sum);

014 8100 out

015 0000 halt exit(0);

016 0000 n: .word int n;

017 0000 sum: .word int sum;

000 .end start

3

10

20

30

**3.3 PROBLEMS AND PROGRAMS**

1. Rewrite the **readandsum()** program using an up-counting loop.
2. Write a machine language program to input **n** numbers and output the largest number.
3. Write a machine language program to input **n** numbers and output them in reverse order. (If the input data is "3, 10, 20, 30", the output should be "30, 20, 10".) This is difficult because it requires modifying instructions.

**4.0 SIM4 - A GENERAL REGISTER MACHINE**

**4.1 INTRODUCTION**

When Intel Corporation invented the microprocessor (a processor on a chip) in the early 1970's, the largest chips that could be manufactured contained only several thousand transistors. As a result, early microprocessors only contained a small number of special purpose registers such as **acc** and **ip** in the SIM processors. As it became possible to manufacture chips with millions of transistors, microprocessors evolved to contain a larger number of general purpose registers. For example, in 32-bit mode, modern Intel microprocessors allow the programmer to access eight somewhat specialized 32-bit registers. In 64-bit mode, these same processors allow access to sixteen 64-bit general registers.

The SIM4 design adds eight general registers to the two registers (**acc** and **ip**) of the previous design. The ten processor registers appear as follows:

+---PROCESSOR---+

| reg content |

| +------+ |

| 0 | dddd | | %r0 is %acc

| +------+ |

| 1 | dddd | |

| +------+ |

| ... |

| +------+ |

| 8 | dddd | |

| +------+ |

| 9 | 0ddd | | %r9 is %ip

| +------+ |

+---------------+

/\

/||\

||

\||/

\/

MEMORY

As shown in the figure, **%r0** is the new name for the accumulator (**acc**) and **%r9** is the new name for the instruction pointer (**ip**). Because **%r9** serves as the instruction pointer, the first digit is forced to be a zero, so that the contents are always a valid address (0000 to 0999).

**4.2 SIM4 INSTRUCTIONS**

In the SIM4 processor, The HALT, JUMP, and memory instructions (LOAD, STORE, ADD, and SUBTRACT) are unchanged. The memory instructions (operation codes 1aaa through 5aaa) always use **%r0** because there are not any "spare digits" that could be used to specify a general register.

The SKIP family can now test the value in any of the ten processor registers. The table below compares the old family of SKIP instructions with the SIM4 family. The instruction format is changed from **6n00** (where **n** specifies the particular instruction) to **6n0r** where **r** specifies one of the ten processor registers (**%r0** through **%r9**). The assembly language statements are also modified. The assembly statement **skeq** which skips if **acc** is equal to zero, becomes **skeq %r0**. The table below illustrates the changes.

OLD (SIM2, SIM3) SKIP INSTRUCTION FAMILY

machine assembly

name format format effect

JUMP 6aaa jmp symb %ip = aaa

SKIP 7000 skip %ip = %ip + 1

SKEQ 7100 skeq if (%acc==0) %ip=%ip+1

SKNE 7200 skne if (%acc!=0) %ip=%ip+1

SKGT 7300 skgt if (%acc> 0) %ip=%ip+1

SKGE 7400 skge if (%acc>=0) %ip=%ip+1

SKLT 7500 sklt if (%acc< 0) %ip=%ip+1

SKLE 7600 skle if (%acc<=0) %ip=%ip+1

NEW (SIM4) SKIP INSTRUCTION FAMILY (r is 0 to 9 and %rn is %r0 to %r9)

machine assembly

name format format effect

SKIP 7000 skip R[9] = R[9]+1

SKEQ 710r skeq %rn if (R[r]==0) R[9]=R[9]+1

SKNE 720r skne %rn if (R[r]!=0) R[9]=R[9]+1

SKGT 730r skgt %rn if (R[r]> 0) R[9]=R[9]+1

SKGE 740r skge %rn if (R[r]>=0) R[9]=R[9]+1

SKLT 750r sklt %rn if (R[r]< 0) R[9]=R[9]+1

SKLE 760r skle %rn if (R[r]<=0) R[9]=R[9]+1

In a similar way, the accumulator family of instructions becomes the single register family of instructions.

SINGLE REGISTER INSTRUCTION FAMILY (r is 0 to 9 and %rn is %r0 to %r9)

machine assembly

name format format effect

IN 800r in %rn R[r] = input device

OUT 810r out %rn output device = R[r]

CLR 820r clr %rn R[r] = 0000

INC 830r inc %rn R[r] = R[r] + 1

DEC 840r dec %rn R[r] = R[r] - 1

NEG 850r neg %rn R[r] = 0.0 - R[r]

SHFTL 860r shftl %rn R[r] = R[r] \* 10

SHFTR 870r shftr %rn R[r] = R[r] / 10

Finally, the **9xxx** operation code (unused in SIM1, SIM2, and SIM3) is used to implement a new two-register family of instructions. These instructions can move words between registers, add and subtract registers, and use addresses in registers to access memory. Since two registers need to be specified, the letter **s** is used to specify a general register used as a source of a word and **d** is used when the register is the destination of a word.

TWO REGISTER INSTRUCTION FAMILY (s and d are 0 to 9, %rs and %rd are %r0 to %r9)

machine assembly

name format format effect

------> <-------

MVRR 90sd mov %rs,%rd R[d]=R[s]

MVMR 91sd mov (%rs),%rd R[d]=M[R[s]]

MVRM 92sd mov %rs,(%rd) M[R[d]]=R[s]

EXCH 93sd exch %rs,%rd R[d]<=>R[s]

ADDRR 94sd add %rs,%rd R[d]= R[d]+R[s]

SUBRR 95sd sub %rs,%rd R[d]= R[d]-R[s]

MVRR, MVMR, and MVRM are acronyms for MoVe Register to Register, MoVe Memory to Register, and MoVe Register to Memory. EXCH exchanges the words in a pair of registers, and ADDRR (ADD Register to Register) and SUBRR perform addition and subtraction on words in a pair of registers.

The order in which the two register appear can be confusing. For example, in the following instruction:

machine assembly

name format format effect

------> <-------

MVRR 9035 mov %r3,%r5 R[5]=R[3]

the machine and assembly language versions should be read as "move the word in register 3 to register 5" (i.e., the information moves from left to right). However, when the effect of the instruction is described "R[5]=R[3]", the information flow is right to left. We adopt left-to-right for assembly language because the GNU assembler in our textbook moves information left-to-right. The SIM machine language is designed to follow the assembly language convention. In contrast, most documentation uses the right-to-left convention of high-level languages like C and Java so the right-to-left convention is used when describing the effect of an instruction.

**4.3 SAMPLE PROGRAMS**

The **file sum1to10.sim4** contains a program to sum the integers from 1 to 10 and leave the result in memory address **sum**. A down-counting loop is used with the index stored in **%r2** and the sum temporarily in **%r1** before being moved to **sum**.

sum1to10.sim4 - Sum the integers from 1 to 10

000 8201 start: clr %r1 %r1 = 0;

001 5010 lda 10 for (%r2=10;%r2!=0;%r2--)

002 9002 mov %r0,%r2 {

003 7202 loop: skne %r2

004 6008 jmp done

005 9421 add %r2,%r1 %r1=%r1+%r2;

006 8402 dec %r2

007 6003 jmp loop }

008 9010 done: mov %r1,%r0 sum = %r1;

009 2011 st sum

010 0000 halt return;

011 0000 sum: .word int sum;

000 .end start

It requires two instructions to load the constant 10 into register 2 - one to load the constant into **%r0** and one to copy **%r0** to **%r2**.

001 5010 lda 10 for (%r2=10;%r2!=0;%r2--)

002 9002 mov %r0,%r2 {

An assembly statement such as **mov 10,%r2** would be illegal because there is no corresponding SIM4 machine language instruction.

Similarly, it requires two instructions to move the sum from **%r2** into memory cell **sum**.

008 9010 done: mov %r1,%r0 sum = %r1;

009 2011 st sum

The same result could have been achieved by loading the address of **sum** into **%r0** and then

using a **mov** instruction.

008 5011 done: lda sum,%r0 sum = %r1;

009 9210 mov %r1,(%r0)

Instead of loading the word at address **sum** into **%r0**, the instruction **lda sum,%r0** loads the address **sum** (011) into **%r0**. The second operand of the move instruction is **(%r0)** rather than **%r0**. Instead of moving the value in **%r1** into **%r0**, this **mov** instruction moves the value in **%r1** to the memory cell whose address is contained in **%r0**. We can manipulate arrays of memory cells by placing addresses into registers and then manipulating the addresses.

There are three different move machine language operation codes - MVRR (90sd), MVMR (91sd), and MVRM (92sd) but only one assembly language operation code (**mov**). The assembler looks that the operands that follow the **mov** operation code and selects the correct machine language operation code.

As an example of using arrays, the following program reads up to 100 numbers and then prints them out in reverse order. If this program is given the input "3 10 20 30", the output will be "0030, 0020, 0010".

reverse.sim4 - Input n integers and output them in reverse order

000 8001 start: in %r1 n = read();

001 5020 lda array %r2 points to array

002 9002 mov %r0,%r2

003 9013 mov %r1,%r3 for (i=n; i!=0; --i)

004 7203 inlp: skne %r3 {

005 6011 jmp outlp

006 8403 dec %r3

007 8004 in %r4 array[%r2] = read();

008 9242 mov %r4,(%r2)

009 8302 inc %r2 %r2++

010 6004 jmp inlp }

011 9013 mov %r1,%r3 for (i=n; i!=0; --i)

012 7203 outlp: skne %r3 {

013 6019 jmp done

014 8403 dec %r3

015 8402 dec %r2 %r2--;

016 9124 mov (%r2),%r4 write() = array[%r2];

017 8104 out %r4

018 6012 jmp outlp }

019 0000 done: halt return;

020 0000 array: .=.+100 int array[100];

000 .end start

3

10

20

30

Recall that the assembly language symbol "." is called the location counter which specifies the address where the next word of code is to be placed. When the assembly statement:

020 0000 array: .=.+100 int array[100];

is encountered, the location counter is equal to 020. The assembly directive **.=.+100** adds 100 to the location counter producing a value of 120. As a result, addresses 020 through 119 (100 memory words) are reserved for **array**.

As the **reverse.sim4** program illustrates, long machine language programs can be difficult to write, debug, and understand. Rather than trying to write machine language code directly, it is easier to create code by writing a program in SIM4 assembly language and then translating it "by hand" into SIM4 machine language (or you could write a program to do the translation for you).

**4.4 CALLING AND RETURNING FROM FUNCTIONS**

Up to this point, there has been no reasonable way to implement function calls. The problem is not in calling a function - the **jmp** instruction would accomplish that. The problem is in returning to the calling function when the called function completes. This requires some method for saving the value in the instruction pointer (**ip** or **%r9**) when the function is called.

SIM4 relies on the fact that register 9 (**%r9**) is the instruction pointer. To call a function called mult(), we can load the address "mult" into a general register (say register 8) and then exchange the values in registers 8 and 9. In assembly language:

lda mult ; address "mult" in %r0

mov %r0,%r8 ; address "mult" in %r8

exch %r8,%r9 ; call mult() - addr "back" in %r8, addr "mult" in %r9,

back: ...

halt

mult: ...

mov %r8,%r9 ; return to caller at address back

After the **exch %r8,%r9** instruction is fetched from memory, the SIM4 processor increments the instruction pointer (**%r9 = %r9 + 1**) so that **%r9** contains the address of the next instruction (**back:**). This is the address stored in **%r8** when the **exch %r8,%r9** instruction is executed,

The following program inputs two integers into **%r1** and **%r2**, calls the function **%r0=mult(%r1,%r2)** to compute the product by successive addition, and then prints the product that the function **mult()** leaves in **%r0**.

mult.sim4 - Read 2 integers and print their product

000 8001 start: in %r1 r1 = read();

001 8002 in %r2 r2 = read();

002 5007 lda mult r0 = mult(r1,r2);

003 9008 mov %r0,%r8

004 9389 exch %r8,%r9

005 8100 out %r0 out() = r0;

006 0000 halt return;

007 8200 mult: clr %r0 r0 = 0;

008 7201 loop: skne %r1 if (r1==0)

009 9089 mov %r8,%r9 return;

010 9420 add %r2,%r0 r0 = r0 + r2;

011 8401 dec %r1 r1--;

012 6008 jmp loop goto loop;

000 .end start

3

4

**4.5 IMPLEMENTATION OF THE SIM4 PROCESSOR**

Creating the SIM4 general register processor from the SIM3 processor is straight forward.

Instead the variables **ip** and **acc**, SIM4 uses an array of integers.

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 main program, a **case** is added to the **switch** statement to call function **regop()** when member of the two-register family of instruction (format 9nsd) is encountered.

/\* Process opcodes 0 to 9 with switch statement \*/

switch ( digit1 ) {

...

case ACCSET:

accop();

break;

case GENREGSET:

regop();

break;

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

...

} /\* 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 to function **accop()** to accommodate the new processor registers. Since the register being modified may have been the instruction pointer, **regs[9]**, the instruction

regs[9] = regs[9] % 1000; /\* make sure %ip has legal address \*/

is executed at the end of the function to insure that the value in **regs[9]** is a legal address (000 to 999).

void accop()

{

...

reg = regs[digit4];

switch ( digit2 ) {

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

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

scanf("%i", &reg);

if (reg < 0) reg = 0;

reg = reg % (WORDLIMIT + 1);

break;

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

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

break;

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

reg = 0;

break;

...

} /\* end of "800n" switch \*/

regs[digit4] = reg; /\* update real register \*/

regs[9] = regs[9] % 1000; /\* make sure %ip has legal address \*/

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

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

Finally, the function **regop()** is added to process operation codes in the two-register family. The implementation is straightforward.

/\* register operation codes (two registers format 9nxy) \*/

void regop()

{

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

sreg = regs[digit3]; /\* source register \*/

dreg = regs[digit4]; /\* destination register \*/

switch ( digit2 ) {

case LDR: /\* load from register, %rd = %rs \*/

dreg = sreg;

break;

case LDRA: /\* load from reg adr, %rd = memory[%rs] \*/

dreg = memory[sreg];

break;

case STRA: /\* store using reg adr, memory[%rd] = %rs \*/

memory[dreg] = sreg;

break;

...

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

regs[digit3] = sreg; /\* only needed for exch \*/

regs[digit4] = dreg;

regs[9] = regs[9] % 1000; /\* make sure %ip has legal address \*/

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

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

**4.6 QUESTIONS AND PROBLEMS**

1. The programs **readv2.sim3** and **addn.sim4** both add **n** numbers and print the sum. However, **addn.sim4** is more efficient than **readv2.sim3** because it uses the general registers rather than memory to store the data words.
   1. If **n** is equal to 3, how many instructions does **readv2.sim3** execute by the time it halts. (Just execute the program).
   2. If **n** is equal to 3, how many instructions does addn.sim4 require.
   3. Find general formulas for the number of instructions executed by readv2.sim3 and addn.sim4 for an arbitrary value of n.
2. Write a machine language program to input **n** numbers and output them in reverse order. (If the input data is "3, 10, 20, 30", the output should be "30, 20, 10".) Use a processor register to point to the array, and access the array by modifying the register.
3. Write a loader program that reads an array into memory beginning at address 100 and then jumps to address 100. (If the data to your program is as follows, your program should output 1000.)

0004

8002

8302

8102

0000

0999