**EEL 2880: Engineering Software Techniques Copy right thru 2020**

**Dr. Subbarao V Wunnava: Original 10 08 2019/Modified 03 12 2020**

 **Building Your Own Computer: Chapter 7 special section**

**Chapter 7 special section Material from Chp 6, 7, 8 and 10 & 12: Structures**

**[ Courtesy: Intel, Micorsoft, and IBM Corporations]**

In the next several problems, we take a temporary diversion away from the world of high-level language programming. We “peel open” a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs.

To make this an especially valuable experience, we then build a computer (through the technique of software-based

*simulation)* on which you can execute your machine-language programs!

\*\*\*\*\*\*\*\*\*

***(Machine-Language Programming)*** Let’s create a computer we’ll call the Simpletron. As its name implies, it’s a simple machine, but as we’ll soon see, a powerful one as well.

The Simpletron runs programs written in the only language it directly understands—

that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an *accumulator—*a “special register” in which information is put before the Simpletron uses that information in calculations or examines it in various ways.

All information in the Simpletron is handled in terms of *words.* A word is a signed four-digit decimal number such as +3364, -1293, +0007, -0001 and so on.

The Simpletron is equipped with a 100- word memory, and these words are referenced by their location numbers 00, 01,99.

Before running an SML program, we must *load* or place the program into memory.

The first instruction (or statement) of every SML program is always placed in location 00.

\*\*\*\*\*\*\*\*

Each instruction written in SML occupies one word of the Simpletron's memory (and hence instructions are signed four-digit decimal numbers).

We assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus.

Each location in the Simpletron’s memory may contain either an instruction, a data value used by a program or an unused (and hence undefined) area of memory.

The first two digits of each SML instruction are the *operation code,* which specifies the operation to be performed. SML operation codes are:aning

**Input/output operations:**

***Operation code (Op.Code) Description***

**#define READ 10:** Read a word from the terminal into a specific location in memory.

**#define WRITE 11:** Write a word from a specific location in memory to the terminal.

**Load/store operations:**

**#define LOAD 20:** Load a word from a specific location in memory into the accumulator.

**#define STORE 21:** Store a word from the accumulator into a specific location in memory.

**Arithmetic operations:**

**#define ADD 30** Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).

**#define SUBTRACT 31** Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).

**#define DIVIDE 32** Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).

**#define MULTIPLY 33** Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).

**Transfer of control operations:**

**#define BRANCH 40** Branch to a specific location in memory.

**#define BRANCHNEG 41** Branch to a specific location in memory if the accumulator is negative.

**#define BRANCHZERO 42** Branch to a specific location in memory if the accumulator is zero.

**#define HALT 43** Halt—i.e., the program has completed its task.

**FIGURE 7.32: Simpletron Machine Language (SML) Op.Codes**

\*\*\*\*\*\*\*\*

The last two digits of an SML instruction are the *operand,* which is the address of the memory location containing the word to which the operation applies.

Now let’s consider several simple SML programs.

The following SML program reads two numbers from the keyboard, and computes and prints their sum.

cation Number Instruction

00 +1007 *(Read A)*

01 +1008 *(Read B)*

02 +2007 *(Load A)*

03 +3008 *(Add B)*

04 +2109 *(Store C)*

05 +1109 *(Write C)*

06 +4300 *(Halt)*

07 +0000 *(Variable A)*

08 +0000 *(Variable B)*

09 +0000 *(Result C)*

The instruction +1007 reads the first number from the keyboard and places it into location 07 (which has been initialized to zero).

Then +1008 reads the next number into location 08. The *load*

instruction, +2007, puts the first number into the accumulator,

and the *add* instruction, +3008,adds the second number to the number in the accumulator.

*All SML arithmetic instructions leave* *their results in the accumulator.*

The *store* instruction, +2109, places the result back into memory

location 09, from which the *write* instruction, +1109, takes the number and prints it (as a signed four-digit decimal number).

The *halt* instruction, +4300, terminates execution.

The following SML program reads two numbers from the keyboard, and determines and

prints the larger value. Note the use of the instruction +4107 as a conditional transfer of control (much the same as C’s if statement)

Location Number Instruction

00 +1009 *(Read A)*

01 +1010 *(Read B)*

02 +2009 *(Load A)*

03 +3110 *(Subtract B)*

04 +4107 *(Branch negative to 07)*

05 +1109 *(Write A)*

06 +4300 *(Halt)*

07 +1110 *(Write B)*

08 +4300 *(Halt)*

09 +0000 *(Variable A)*

10 +0000 *(Variable B)*

Now write SML programs to accomplish each of the following tasks.

a) Use a sentinel-controlled loop to read 10 positive integers and compute and print their sum.

00 +1009 (Read Value)

01 +2009 (Load Value)

02 +4106 (Branch negative to 06)

03 +3008 (Add Sum)

04 +2108 (Store Sum)

05 +4000 (Branch 00)

06 +1108 (Write Sum)

07 +4300 (Halt)

08 +0000 (Variable Sum)

09 +0000 (Variable Value)

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

00 +2018 (Load Counter)

01 +3121 (Subtract Termination)

02 +4211 (Branch zero to 11)

03 +2018 (Load Counter)

04 +3019 (Add Increment)

05 +2118 (Store Counter)

06 +1017 (Read Value)

07 +2016 (Load Sum)

08 +3017 (Add Value)

09 +2116 (Store Sum)

10 +4000 (Branch 00)

11 +2016 (Load Sum)

12 +3218 (Divide Counter)

13 +2120 (Store Result)

14 +1120 (Write Result)

15 +4300 (Halt)

16 +0000 (Variable Sum)

17 +0000 (Variable Value)

18 +0000 (Variable Counter)

19 +0001 (Variable Increment)

20 +0000 (Variable Result)

21 +0007 (Variable Termination)

c) Read a series of numbers and determine and print the largest number. The first number

read indicates how many numbers should be processed.

00 +1017 (Read Endvalue)

01 +2018 (Load Counter)

02 +3117 (Subtract Endvalue)

03 +4215 (Branch zero to 15)

04 +2018 (Load Counter)

05 +3021 (Add Increment)

06 +2118 (Store Counter)

07 +1019 (Read Value)

08 +2020 (Load Largest)

09 +3119 (Subtract Value)

10 +4112 (Branch negative to 12)

11 +4001 (Branch 01)

12 +2019 (Load Value)

13 +2120 (Store Largest)

14 +4001 (Branch 01)

15 +1120 (Write Largest)

16 +4300 (Halt)

17 +0000 (Variable Endvalue)

18 +0000 (Variable Counter)

19 +0000 (Variable Value)

20 +0000 (Variable Largest)

21 +0001 (Variable Increment)

**EEL 2880: Software Techniques Fall 2019**

**Computer Simulator Development with C: Reference: Problem 7.28**

**Deitel/Subbarao 03 05 2018 Copy Right extended to 2019**

**1** // **Exercise 7.28 Solution 03 05 18: Subbarao/Dietel**

**2 #include** <stdio.h>

**3**

**4**

**// --------------------------------------------------------------command definitions COMMANDS**

// **define commands**

**5 #define SIZE 100**

**6 #define SENTINEL -99999**

**7 #define TRUE 1**

**8 #define FALSE 0**

**9 #define READ 10**

**10 #define WRITE 11**

**11 #define LOAD 20**

**12 #define STORE 21**

**13 #define ADD 30**

**14 #define SUBTRACT 31**

**15 #define DIVIDE 32**

**16 #define MULTIPLY 33**

**17 #define BRANCH 40**

**18 #define BRANCHNEG 41**

**19 #define BRANCHZERO 42**

**20 #define HALT 43**

**// ----------------------------------------------------------end command definitions**

**21**

**22** // **function prototypes--------------------------declare function prototypes FUNCTIONS**

**23 void** load( **int** \*loadMemory );

**24 void** execute( **int** \*memory, **int** \*acPtr, **size\_t** \*icPtr, **int** \*irPtr,

**25 int** \*opCodePtr, **int** \*opPtr );

**26 void** dump( **int** \*memory, **int** accumulator, **size\_t** instructionCounter,

**27 int** instructionRegister, **int** operationCode,

**28 int** operand );

**29 int** validWord( **int** word );

//------------------------------------------------------------**end function prototype definitions**

**30// start of main program -------------------------------------------------------------------------- start main MAIN**

**31 int** main()

**32** {

**33 int** memory[ **SIZE** ]; // define memory array

**34 int** ac = **0**; // accumulator

**35 size\_t** ic = **0**; // instruction counter

**36 int** opCode = **0**; // operation code

**37 int** op = **0**; // operand

**38 int** ir = **0**; // instruction register

**39 size\_t** i; // counter

**40**

**41** // clear memory -----------------------------------**clear memory operation for loop CLEAR MEMORY**

**42 for** ( i = **0**; i < **SIZE**; ++i )

 {

**43** memory[ i ] = **0**;

**44** } // end for-----------------------------------**end for loop**

**45 // load memory -------------------------------------------------load memory operation LOAD MEMORY**

**46** load( memory );

**47** execute( memory, &ac, &ic, &ir, &opCode, &op );

**48** dump( memory, ac, ic, ir, opCode, op );

**49** } // end main-------------------------------------------**end main function**

**50**

**51** // function loads instructions-------------------------------**load function operation LOAD INSTRUCTION**

**52 void** load( **int** \*loadMemory )

**53** {

**54 int** instruction; // current instruction

**55 size\_t** i = **0**; // indexing variable

**56**

**57** printf( **"%s\n\n%s\n%s\n%s\n%s\n%s\n%s\n\n"**,

**58 "\*\*\* Welcome to Simpletron \*\*\*"**,

**59 "\*\*\* Please enter your program one instruction \*\*\*"**,

**60 "\*\*\* ( or data word ) at a time. I will type the \*\*\*"**,

**61 "\*\*\* location number and a question mark ( ? ). \*\*\*"**,

**62 "\*\*\* You then type the word for that location. \*\*\*"**,

**63 "\*\*\* Type the sentinel -99999 to stop entering \*\*\*"**,

**64 "\*\*\* your program. \*\*\*"** );

**65**

**66** printf( **"%s"**, **"00 ? "** );

**67** scanf( **"%d"**, &instruction ); // read instruction **READ INSTRUCTION**

**68**

**69** // while sentinel is not read from user-------------**checking sentinel value with while**

**70 while** ( instruction != **SENTINEL** )

 {

**71**

**72** // test instruction for validity--------------**instruction validity with if loop**

**73 if** ( !validWord( instruction ) )

 {

**74** puts( **"Number out of range. Please enter again.\n"** );

**75** } // end if-------------------------------------------------------------------**end if and start else**

**76 else**

{

 // load instruction

**77** loadMemory[ i++ ] = instruction;

**78** } // end else-------------------------------------------------------------------**end else**

**79**

**80** printf( **"%02d ? "**, i );

**81** scanf( **"%d"**, &instruction );

**82** } // end while----------------------------------------------------------------**end while**

**83**

**84** } // end function load----------------------------------**end function load**

**85**

**86** // carry out the commands--------------------------**carry out commands: execute function EXECUTE INSTUCTION**

**87 void** execute( **int** \*memory, **int** \*acPtr, **size\_t** \*icPtr, **int** \*irPtr,

**88 int** \*opCodePtr, **int** \*opPtr )

**89** {

**90 int** fatal = **FALSE**; // fatal error flag ---------------------------**fatal error flag**

**91 int** temp; // temporary holding space

**92**

**93** puts( **"\n\*\*\*\*\*\*\*\*\*\*\*\*START SIMPLETRON EXECUTION\*\*\*\*\*\*\*\*\*\*\*\*\n\n"** );

**94**

**95** // separate operation code and operand------------------------**opcode and operand separation**

**96** \*irPtr = memory[ \*icPtr ];

**97** \*opCodePtr = \*irPtr / **100**;

**98** \*opPtr = \*irPtr % **100**;

**99**

**100** // loop while command is not HALT or fatal-------------------**Not HALT or fatal; start while**

**101 while** ( \*opCodePtr != **HALT** && !fatal )

 {

**102**

**103** // determine appropriate action---------------------**start switch**

**104 switch** ( \*opCodePtr )

 {

**105**

**106** // read data into location in memory **READ**

**107 case READ**:

**108** puts( **"Enter an integer: "** );

**109** scanf( **"%d"**, &temp );

**110**

**111** // check for validity -------------------------------**and start inner while loop**

**112 while** ( !validWord( temp ) )

 {

**113** puts( **"Number out of range. Please enter again: "** );

**114** scanf( **"%d"**, &temp );

**115** } // end while-------------------------------------------------------------------**end inner while**

**116**

**117** memory[ \*opPtr ] = temp; // write to memory--------------------**memory write operation MEMORY WRITE**

**118** ++( \*icPtr );

**119 break**; // exit switch

**120**

**121** // write data from memory to screen------------------------**memory to screen operation SCREEN WRITE**

**122 case WRITE**:

**123** printf( **"Contents of %02d: %d\n"**, \*opPtr, memory[ \*opPtr ] );

**124** ++( \*icPtr );

**125 break**; // exit switch

**126**

**127** // load data from memory into accumulator------------**memory to accumulator LOAD ACCUMULATOR**

**128 case LOAD**:

**129** \*acPtr = memory[ \*opPtr ];

**130** ++( \*icPtr );

**131 break**; // exit switch

**132**

**133** // store data from accumulator into memory-----**accumulator to memory STORE ACCUMULATOR**

**134 case STORE**:

**135** memory[ \*opPtr ] = \*acPtr;

**136** ++( \*icPtr );

**137 break**; // exit switch

**138**

**139** // add data from memory to data in accumulator **ADD OPERATION**

**140 case ADD**:

**141** temp = \*acPtr + memory[ \*opPtr ];

**142**

**143** // check validity---------------------------------------------------start **if** loop

**144 if** ( !validWord( temp ) )

 {

**145** puts( **"\*\*\* FATAL ERROR: Accumulator overflow \*\*\*\n"** );

**146** puts( **"\*\*\* Simpletron execution "** );

**147** puts( **"abnormally terminated \*\*\*\n"** );

**148** fatal = **TRUE**;

**149** } // end if---------------------------------------------------------**end if**

**150 else**

{

**151** \*acPtr = temp;

**152** ++( \*icPtr );

**153** } // end else----------------------------------------------**end else**

**154**

**155 break**; // exit switch

**156**

**157** // subtract data in memory from data in accumulator **SUBTRACT OPERATION**

**158 case SUBTRACT**:

**159** temp = \*acPtr - memory[ \*opPtr ];

**160**

**161** // check validity----------------------------------------------**start if loop**

**162 if** ( !validWord( temp ) )

 {

**163** puts( **"\*\*\* FATAL ERROR: Accumulator overflow \*\*\*\n"** );

**164** puts( **"\*\*\* Simpletron execution "** );

**165** puts( **"abnormally terminated \*\*\*\n"** );

**166** fatal = **TRUE**;

**167** } // end if-----------------------------------------------------**end if and start else**

**168 else**

{

**169** \*acPtr = temp;

**170** ++( \*icPtr );

**171** } // end else--------------------------------------------------**end else**

**172**

**173 break**; // exit switch

**174**

**175** // divide data in memory into data in accumulator **DIVIDE OPERATION**

**176 case DIVIDE**:

**177**

**178** // check for divide by zero error-------------------------**start if loop**

**179 if** ( memory[ \*opPtr ] == **0** ) {

**180** puts(**"\*\*\* FATAL ERROR: Attempt to divide by zero \*\*\*\n"**);

**181** puts( **"\*\*\* Simpletron execution "** );

**182** puts( **"abnormally terminated \*\*\*\n"** );

**183** fatal = **TRUE**;

**184** } // end if---------------------------------------------------------**end if and start else**

**185 else**

{

**186** \*acPtr /= memory[ \*opPtr ];

**187** ++( \*icPtr );

**188** } // end else------------------------------------------------------**end else**

**189**

**190 break**; // exit switch

**191**

**192** // multiply data in memory by data in accumulator **MULTIPLY OPERATION**

**193 case MULTIPLY**:

**194** temp = \*acPtr \* memory[ \*opPtr ];

**195**

**196** // check validity---------------------------------------**start if**

**197 if** ( !validWord( temp ) ) {

**198** puts( **"\*\*\* FATAL ERROR: Accumulator overflow \*\*\*\n"** );

**199** puts( **"\*\*\* Simpletron execution "** );

**200** puts( **"abnormally terminated \*\*\*\n"** );

**201** fatal = **TRUE**;

**202** } // end if-----------------------------------------------**end if and start else**

**203 else**

{

**204** \*acPtr = temp;

**205** ++( \*icPtr );

**206** } // end else------------------------------------------**end else**

**207**

**208 break**; // exit switch

**209**

**210** // branch to specific location in memory **BRANCH OPERATION**

**211 case BRANCH**:

**212** \*icPtr = \*opPtr;

**213 break**; // exit switch

**214**

**215** // branch to location in memory if accumulator is negative **BRANCH ON NEGATIVE OPERATION**

**216 case BRANCHNEG**:

**217**

**218** // if accumulator is negative--------------------**start if loop**

**219 if** ( \*acPtr < **0** )

{

**220** \*icPtr = \*opPtr;

**221** } // end if-------------------------------------------**end if and start else**

**222 else** {

**223** ++( \*icPtr );

**224** } // end else------------------------------------------**end else**

**225**

**226 break**; // exit switch

**227**

**228** // branch to location in memory if accumulator is zero **BRANCH ON ZERO OPERATION**

**229 case BRANCHZERO**:

**230**

**231** // if accumulator is zero-----------------------**start if loop**

**232 if** ( \*acPtr == **0** ) {

**233** \*icPtr = \*opPtr;

**234** } // end if--------------------------------------------- **end if and start else**

**235 else**

{

**236** ++( \*icPtr );

**237** } // end else-----------------------------------------**end else**

**238**

**239 break**; // exit switch------------------------

**240 // --------------------------------------------------------------------------------------DEFAULT**

**241 default**:

**242** puts( **"\*\*\* FATAL ERROR: Invalid opcode detected \*\*\*\n"** );

**243** puts( **"\*\*\* Simpletron execution "** );

**244** puts( **"abnormally terminated \*\*\*\n"** );

**245** fatal = **TRUE**;

**246 break**; // exit switch

**247** } // end switch----------------------------------**final end of switch**

**248**

**249** // separate next operation code and operand

**250** \*irPtr = memory[ \*icPtr ];

**251** \*opCodePtr = \*irPtr / **100**;

**252** \*opPtr = \*irPtr % **100**;

**253** } // end while-------------------------------------**end while loop**

**254**

**255** puts( **"\n\*\*\*\*\*\*\*\*\*\*\*\*\*END SIMPLETRON EXECUTION\*\*\*\*\*\*\*\*\*\*\*\*\*\n"** );

**256** } // end function execute----------------------**end of execute function**

**257**

**258** // print out name and content of each register and memory

**259 void** dump( **int** \*memory, **int** accumulator, **size\_t** instructionCounter,

**260 int** instructionRegister, **int** operationCode,

**261 int** operand )

**262** {

**263 unsigned int** i; // counter

**264**

**265** printf(**"\n%s\n%-23s%+05d\n%-23s%5.2u\n%-23s%+05d\n%-23s%5.2d\n%-23s%5.2d"**,

**266 "REGISTERS:"**, **"accumulator"**, accumulator, **"instructioncounter"**,

**267** instructionCounter, **"instructionregister"**, instructionRegister,

**268 "operationcode"**, operationCode, **"operand"**, operand );

**269**

**270** puts( **"\n\nMEMORY:\n "** );

**271**

**272** // print column headers------------------------------------**start for loop**

**273 for** ( i = **0**; i <= **9**; ++i )

{

**274** printf( **"%5d "**, i );

**275** } // end for---------------------------------------------**end for**

**276**

**277** // print row headers and memory contents-------------**start for**

**278 for** ( i = **0**; i < **SIZE**; ++i )

 {

**279**

**280** // print in increments of 10--------------**start inner if loop**

**281 if** ( i % **10** == **0** )

 {

**282** printf( **"\n%2d "**, i );

**283** } // end if----------------------------**end inner if**

**284**

**285** printf( **"%+05d "**, memory[ i ] );

**286** } // end for--------------------------------------------------------**end for**

**287**

**288** puts( **""** );

**289** } // end function dump----------------------------------**end dump function**

**290**

**291** // function tests validity of word------------------**testing word validity with validword function**

**292 int** validWord( **int** word )

**293** {

**294 return** word >= **-9999** && word <= **9999**;

**295** } // end function validWord---------------------------**end validword function**

297// end SIMPLETRON C simulation

**Program 1: Adds two numbers and prints the sum**

****

****

****

**PROGRAM 2: Prints larger of the two numbers entered:**

****



