CS 4380 Project 4: Procedures and Beyond

Introduction

At this point in the semester, you have created a working emulator and assembler for the 4380, and have used it to investigate the implications of program behavior and caching. For this, the fourth and final project of the semester you will complete your emulator and assembler by implementing the 4380 Mark 4. The Mark 4 includes instructions to support procedure calls and dynamic memory management, as well as string and array manipulation. With these added features in place, you will exercise the full range of your assembler and emulator's capabilities by implementing several programs that will be described below. These programs begin simply, but increase in complexity, with the final program being worth 50% of your final exam grade.

Part 1: 4380 Emulator and Assembler Mark 4

To meet the requirements of this lab it will be necessary to extend the 4380 emulator and assemblers with support for the following additional instructions:

Logical Instructions: (NOT BITWISE)

Value	Operator	Operand 1	Operand 2	Operand 3	Immediate Value	Description
27	AND	RD	RS1	RS2	DC*	Perform RS1 && RS2, store result in RD (1 = T, 0 = F)
28	OR	RD	RS1	RS2	DC*	Perform RS1 || RS2, store result in RD (1 = T, 0 = F)

*This operand shall be omitted in 4380 assembly instructions but is required in 4380 bytecode.

Procedure/Stack Instructions:

Value	Operator	Operand 1	Operand 2	Operand 3	Immediate Value	Description
35	PSHR	RS	DC*	DC*	DC*	Set SP = SP - 4; Place the word in RS onto the stack
36	PSHB	RS	DC*	DC*	DC*	Set SP = SP - 1; Place the least significant byte in RS onto the stack
37	POPR	RD	DC*	DC*	DC*	Place the word on top of the stack into RD, update SP = SP + 4
38	POPB	RD	DC*	DC*	DC*	Place the byte on top of the stack into RD, update SP = SP + 1
39	CALL	DC*	DC*	DC*	Address**	Push PC onto stack, update PC to Address
40	RET	DC*	DC*	DC*	DC*	Pop stack into PC

*DC = "Don't Care". This operand shall be omitted in 4380 assembly instructions but is required in 4380 bytecode.

** When represented in an assembly file the Address shall be represented as a label.

Heap Instructions:

Value	Operator	Operand 1	Operand 2	Operand 3	Immediate Value	Description
32	ALCI	RD	DC***	DC***	Imm*	Allocate Imm bytes of space on the heap (and increment HP accordingly). The Imm value is a 4-byte unsigned integer. Store the initial heap pointer in RD.
33	ALLC	RD	DC***	DC***	Address**	Allocate a number of bytes on the heap according to the value of the 4-byte unsigned integer stored at Address (and increment HP accordingly). Store the initial heap pointer in RD.
34	IALLC	RD	RS1	DC***	DC***	Indirectly allocate a number of bytes on the heap according to the value of the 4-byte unsigned integer at the memory address stored in RS1 (and increment HP accordingly). Store the initial heap pointer in RD.

* When represented in an assembly file the Imm value shall be represented as a numeric literal.

** When represented in an assembly file the Address shall be represented as a label in the data section.

*** DC = "Don't Care". These operands shall be omitted in 4380 assembly instructions but are required in 4380 bytecode.

Traps/Interrupts:

Value	Operator	Operand 1*	Operand 2**	Operand 3**	Immediate Value**	Description
31	TRP	#5	DC**	DC**	DC**	Write the null-terminated Pascal-style string whose starting address is in R3 to stdout.
31	TRP	#6	DC**	DC**	DC**	Read a newline terminated string from stdin and store it in memory as a null-terminated Pascal-style string whose starting address is in R3. (Do not store the newline)

*Note: The leading '#' is required in 4380 assembly programs; only the following/trailing numeric value is represented in 4380 bytecode.

**DC = "Don't Care". These operands shall be omitted in 4380 assembly instructions but are required in 4380 bytecode.

Directives:

Directive	Operand	Examples	Behavior
.BTS	Required unsigned decimal value. This value is the number of bytes to be allocated.	.BTS #25 .BTS #5 a_label .BTS #20	Allocates the specified number of bytes in place and initializes them all to 0. If an optional label is present the label shall be associated with the address of the first byte allocated.
.STR	Required double quote delimited string OR a numeric literal. 255 shall be the maximum length of the string and the maximum value of the numeric literal.	.STR "Example!" name .STR "Fred" .STR #40 .STR #255	Allocates a number of bytes equal to the length of the string + 2. The first byte is initialized to the length of the string, and the last byte to 0 (null character). The remaining bytes (in the middle) are initialized to the ascii values for the characters in the string. **Note:** escape sequences such as \n must be properly handled. OR Allocates a number of bytes equal to the numeric literal + 2. The first byte is initialized to the value of the numeric literal and the remaining bytes are initialized to zeros. In both cases if an optional label is present the label shall be associated with the address of the first byte allocated.

Unless otherwise specified below - all requirements from projects 1 , 2, and 3 remain in force for the assembler and emulator. The following additional requirements must also be met.

Req1) Project 4 work must be completed in a separate branch called project-4. When all requirements are met, the branch should be merged into main (BUT NOT DELETED).

Req2) Upon startup, and after reading the input file into prog_mem, the 4380 emulator shall initialize SB to the highest available memory address + 1. **Note:** SB can be changed by a program at runtime.

Req3) Upon startup, and after reading the input file into prog_mem, the 4380 emulator shall initialize SP to the highest available memory address + 1. **Note:** SP can be changed by a program at runtime.

Req4) Upon startup, and after reading the input file into prog_mem, the 4380 emulator shall initialize SL to the address of the first byte past the last instruction read into program memory. **Note:** SL can be changed by a program at runtime.

Req5) The 4380 emulator shall not allow SP to be set to a value higher than SB, or lower than SL. If at any time a program attempts to set SP to a value outside the range of [SL, SB], the emulator shall print the message "INVALID INSTRUCTION AT: <address>" where <address> is the address in Program Memory of the first byte of the offending instruction. The emulator shall then exit with a return value of 1.

Req6) At runtime, SP shall store the address of the last/latest byte allocated on the stack. In other words, the next available byte on the stack shall be located at SP - 1.

Part 2: Programming the 4380

Now that you have a working 4380 Mark 4 - it's time to run it - and you - through your paces. For part 2 of this project you will write several assembly programs. To "warm you up" we will begin with simple programs and demonstrate strategies you may find useful for writing assembly programs in general. As the programs progress, they increase in difficulty/complexity, and we offer less direct aid. For the final program you are expected to be able to "stand on your own". You may discuss the first three programs with your classmates (and share code with your teammate(s) if teams are allowed by your instructor), but the 4th and final program (Program D) should be considered an exam question and should be completed without any outside help (note: you are free to ask the instructor clarifying questions).

Program A: Fibonacci Sequence

Create a simple program that:

1. Prompts the user with the string "Please enter the Fibonacci term you would like computed: "
2. Reads an integer value from stdin
3. Computes the appropriate Fibonacci term (as indicated by the integer read from stdin) in the sequence beginning with initial values 0, 1. (i.e. 0, 1, 1, 2, 3...)
4. Prints the message "Term N in the Fibonacci sequence is: M\n" where N is the integer read in 2, and M is the value computed in 3.

To assist you in this process we have provided 3 files ProgA_0.cpp - ProgA_2.cpp which demonstrate a useful strategy for writing assembly programs. The strategy utilizes an incremental process that begins with solving the problem first in a language you are familiar with (such as C or C++) and then incrementally modifying the program to "look and feel"

increasingly like a program written in the target assembly language. At each stage of modification (and for as long as possible) you should attempt to keep the program "working" - which in the example provided means the source can be compiled and the resulting program behaves as desired. Continue the process until you reach a point at which you can "comfortably" write an assembly code version of the program.

Program B: A Modulus Function

Program B introduces the idea of procedure calls. This program must ask a user for 2 input values (an integer dividend and divisor) and then report the remainder of dividing the first input by the second.

You will recall that the 4380 ISA lacks a modulus operator. Thus, you will need to implement this functionality by utilizing a combination of existing 4380 operators. To this end you are required to implement a function called "mod" that accepts two integer parameters, performs the necessary arithmetic to compute the remainder of the first divided by the second - and returns this value. Note: The creation and utilization of this function in your program is a requirement to receive credit.

To get you started we have provided you with a sample source file called ProgB.cpp. Please use it as a starting point to perform the kind of C++ to 4380 assembly "conversion" process that was outlined for Program A. The behavior of your final program (including the output of text prompts) must match that of this sample C++ program. Consequently, you may find it useful to compile and run it to familiarize yourself with its behavior.

Note: You will notice that the example program does NOT do any error checking - your assembly program is not required to either. Simply matching the behavior of the example will be sufficient.

Concerning Procedure Calls

As you know, for procedure calls to work reliably it is necessary for both the caller and callee functions to "work together" to communicate with one another, and to save and restore information as needed throughout the process of exchanging control between them. This is typically done through a division of responsibility that is well documented and adhered to by the compiler(s) and/or programmer(s) involved in the development of a program. The following three figures present a suggested process for stack management that you may use in this project (and which should serve as a good starting point for 4490).

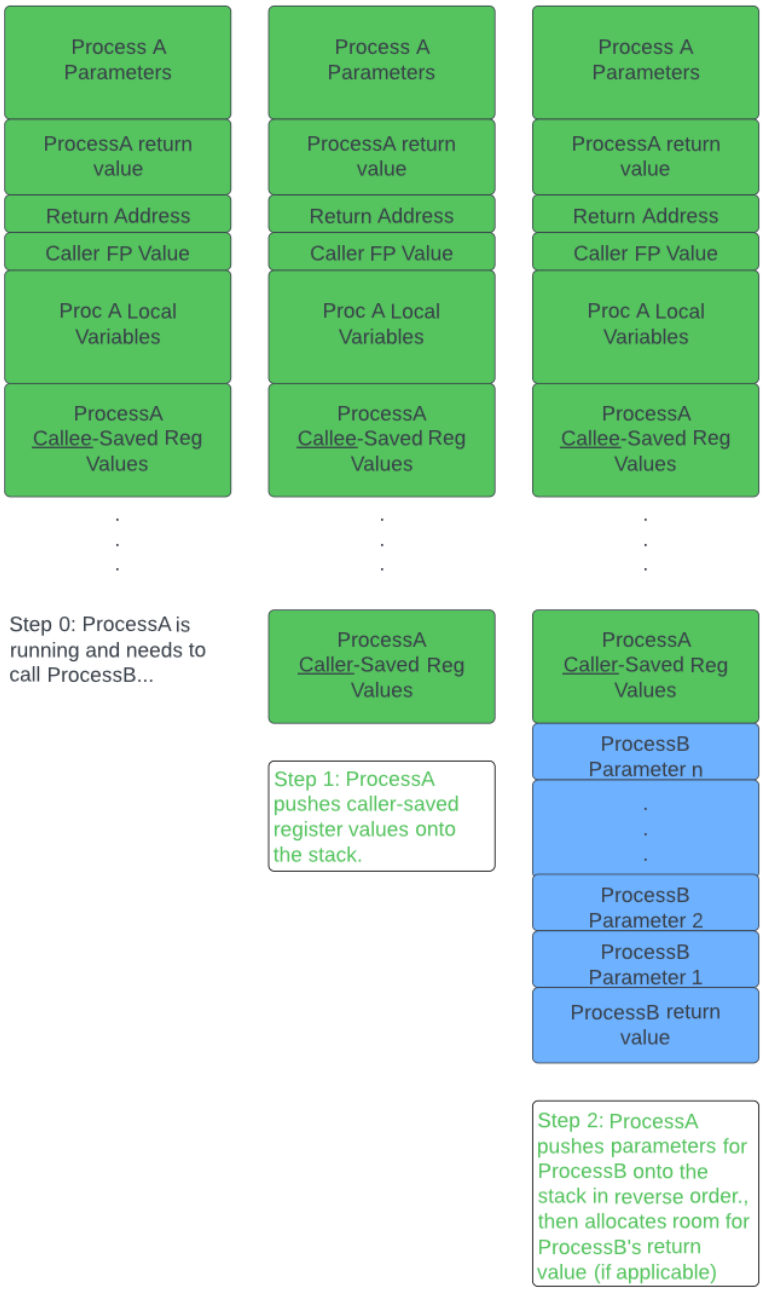

Figure 1: Process for Procedure Calls 1/3

Figure 2: Process for Procedure Calls 2/3

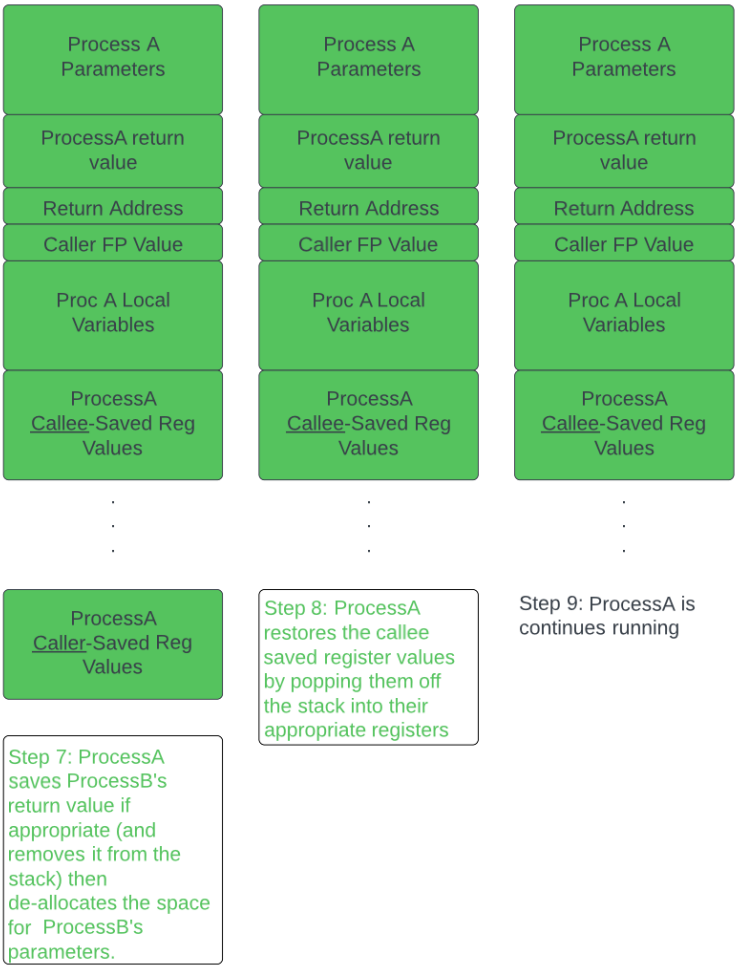

Figure 3: Process for Procedure Calls 3/3

One detail missing from the above diagrams is the division of registers into caller and callee responsible sets. Here is a suggested division of labor:

Caller: R0-R7
Callee: R8-R15, FP

This is merely a suggestion; you are free to divide these registers as you deem appropriate. As a final note - the process diagramed above utilizes the stack exclusively for parameter and return value management. You are welcome to incorporate a standardized (your standard) method of employing registers for these tasks as well, but please note that you will need registers available for performing computation, and that with a limited set you will still need a plan for utilizing the stack for parameter passing when/if you run out...

Program C: Recursive Fibonacci

For Program C you must recreate the behavior of Program A - but this time the computation of a term in the Fibonacci sequence must be accomplished via a recursive function called fib(). Please see the example program contained in ProgC.cpp to get you started. Note: When experimenting with/testing your program, please remember that the recursive calls required increases very quickly as the term you are computing increases, so execution times will increase correspondingly...

Program D: Prime Number Generator

The fourth and final program you must implement for this project is a prime number generator. This program must meet the following behavioral and structural requirements:

R1) The program must be contained within a single .asm file.

R2) Upon execution the program must prompt the user with the following output:

> Welcome to the Prime Number Generator.
>
> This program searches for and displays the first 20 prime numbers greater than or equal to a user provided lower bound.
>
> Please enter a lower bound: <user input entered here>

R3) After the user has entered the lower bound described in R2, the program must compute and store (in program memory) the appropriate 20 prime numbers.

R4) Upon computing and storing the prime numbers described in R3 the program shall print the following prompt, followed by the list of 20 prime numbers, one per line, and then terminate.

> The first 20 prime numbers greater than or equal to <user's value> are:
> <list of primes, one per line>

R5) The program shall utilize at least two functions to accomplish requirements R2 - R4, one of which shall be called is_prime() and shall accept a single parameter (a number to be evaluated for prime-ness) and return 1 if the parameter is prime, and 0 if it is not.

Please remember that you are not allowed to discuss this program with anyone (student or otherwise) with the exception of the course instructor.

What to Turn In:

When you have completed the project, you will need to submit the URL for your repository on the assignment page in Canvas. Your repository should be private **but shared with your course grader and/or instructor.** Instructions for granting access to your repo for the course grader will be provided by your instructor.

You must also submit your assembly source files (.asm) for programs A - D (submitted in Canvas).

Note: There is no late or non-late policy available for this project. Please start working NOW!