CSE 378 - Fall 2001

Machine Organization and Assembly Language Programming

Problem Set #3

Due Wednesday October 24th, 2001

Dynamic translation is becoming of increasing interest in the computing realm. Virtually every desktop with a browser contains a "Just in Time" compiler for Java installed on it. Companies such as Transmeta have been founded on the idea of dynamically translating binaries from one architecture into another. Your task for this assignment is to write such a translator in MIPS assembly code. The source binary will be a very simple architecture (the "378 Machine") and the output "binary" will be MIPS code for execution under SPIM.

The goals of this problem set are many and varied. First, and most basic, it will teach you the basics of writing and debugging assembly language programs. Second, because your program is a binary-code translator, you will learn the software nature of hardware -- that execution is interpretation. Third, because the source binary is from a *stack* machine, you'll be able to apply the general model of a stack that you learned in cse143 to a particular arena, computer architecture. And lastly, because writing code in assembly language requires you to design algorithms at a lower level than is done when programming in a source language, you will learn the guts of jump tables.

The assignment is to write a program in MIPS assembler that is a translator from binaries written for a very simple computer into binaries for SPIM itself! I'll first present the operation of the source machine you'll be translating from, and then describe what you have to do.

The 378 Machine

The machine that the source binary is for (called the 378 machine) is a 32-bit machine with an extremely simple architecture. There are only twelve instructions and no general-purpose registers. Instead, register storage is organized as a stack of 32-bit integers, and all instructions refer to the top of the stack for operands. The 378 CPU has an internal register, the stack pointer (SP), to keep track of the top of the stack. The user's program can affect the SP only indirectly, by adding data to or removing data from the stack. There is also a separate memory to hold instructions, which is addressable only through the 378 PC, and a separate memory for data. In reality, of course, there is only one memory on microprocessors which holds both. In this assignment there are two memories, just to make the coding a little easier for you and to correspond to some of the discussions on implementation (chapter 5) in the text book.

The table below lists the set of instructions available (as well as some other info which will be explained shortly). A user of the 378 machine would write a program using those instructions. (In other words, the input to your program is a 378 binary.)

Opcode	**Operation**	**Example**	**Meaning**
0001	Add	0x10000000	Add
0010	Sub	0x20000000	Subtract
0011	Mult	0x30000000	Multiply
0101	Bgtz	0x5000000C	Branch-greater-than-zero: PC := 12 if stack top > 0
0110	Bez	0x60000010	Branch-equal-to-zero: PC := 16 if stack top == 0
0111	Jump	0x70000000	Jump to addr in top of stack: PC := Stack Top
1000	Load	0x80000000	Load memory to stack top
1001	Store	0x90000000	Store from stack to memory
1010	Interrupt	0xA0000001	Invoke operating system with function X
1101	Clear	0xD0000000	Clear the stack
1110	Push	0xE0000003	Push 3 onto the stack

Stop indicates the end of a program (in the 378 machine). *Add, Sub,* and *Mult* replace the top two integers on the stack with the single integer result of the operation specified. For example, if the stack contains (from bottom to top) {1 2 3}, a *Sub* instruction would result in {1 -1}. *Clear* indicates that the stack (which is the 378 main memory) should be cleared, that is, set to empty. *Bgtz* and *Bez* compare the value at the top of the stack to zero, branch to the target or fall through and pop the stack. *Jump* jumps to the address at the top of the stack. *Load* and *Store* transfer data to and from memory. Load interprets the integer at the top of the stack as the address to load from. It then replaces that integer with the value it loads. Store interprets the integer at the top of the stack as the address to store to. It interprets the next integer on the stack as the value to store, and removes the value after the store. *Clear* indicates that the stack (which is the 378 main memory) should be cleared, that is, set to empty. *Push* adds a new integer to the top of the stack. Notice that only the *Push* instruction specifies an operand.

Interrupt, is the method by which a 378 Machine program calls the operating system. The 378 operating system that these binaries use supports 3 operating system calls: 0xA0000000 terminates the program, 0xA0000001 will print the top of the stack [as an inteteger] and pop that value from the stack, finally 0xA0000002 will read an integer value from the console and put it on the stack.

378 instructions are encoded as 32-bit values. The high order four bits (bits 28 through 31) specify the opcode, as shown in the table. In all instructions other than *bgtz, beq* and *Push,* the low order 28 bits are ignored. In the bgtz and bez instructions, the low 28 bits are used for the branch target; in the *Push* instruction, the low order 28 bits contain a signed integer.

A 378 program lives in the 378 instruction memory, which is separate from the data memory. For purposes of this assignment we'll assume that the 378 machine has only 128 words of data memory and an unlimited amount of code-space memory (up to 4 gigabytes).

The format of a 378 Machine binary is the following:

0: '378 '	: The ASCII text '3' '7' '8' and ' ' which act as "magic numbers" that mark : a 378 binary
4: datasize	: size of the data segment (32 bit word)
8: codesize	: size of the code segment (32 bit word)
12: initial PC	: initial program counter (32 bit word)
16: data	: the data up to size 'datasize' ('datasize' number of 32 bit words)
16+datasize	: code : the code

Conventionally, a 378 Machine would have an operating system that would load this binary into memory and set the program counter to the initial PC as specified. The machine would then begin execution. However, your task is to write a binary code translator which requires you to read this file into SPIM and output a new 'binary'.

Implementing the 378 Machine to SPIM Translator

The high level approach to this assignment is to first correctly parse and understand the format of a 378 Machine binary. Then write output code that outputs data items for those items in the data segment of the 378 binary. Finally, write code that translates the 378 Machine instructions into SPIM symbolic codes. Be careful to translate operating system interfaces from the 378 Machine operating system to the SPIM operating system!

To get you started we've supplied a sample 378 Machine binary at:

/cse/courses/cse378/CurrentQtr/Assignments/378.binary

If you need more documentation on how this binary is formatted, the source code that makes it is also available at:

/cse/courses/cse378/CurrentQtr/Assignments/makebin.cxx

If you were writing this program in C, your program would probably contain a *switch* statement on the 378 instruction opcode, and with thirteen *case*s corresponding to the thirteen instructions. To achieve this effect in your MIPS program, you should use a jump table. The textbook talks about jump tables on page 129. The basic idea is to allocate a table of addresses (in MIPS memory), load an element of the table given by an index (in your case, the 378 instruction opcode) into a register, and then perform a MIPS **jr** instruction to get you to the correct section of MIPS code.

This homework looks daunting, but it really isn't; so you shouldn't find yourself spending painfully large amounts of time on it. You'll see, though, that programming in assembler is somewhat more inconvenient than programming in a higher-level language, like C or C++. If your having trouble writing this assignment in SPIM try prototyping it in C/C++/Java. You can't turn this in for a grade, but it may help solidify what it is you have to write in SPIM.

Running and Testing

In order to run your program, you'll need to do the following:

- Start spim
- Load your program
- Run/step/debug your program
- Load the output of your program into SPIM
- Run/step/debug that program to make sure you are translating correctly!

More Details

It is possible for the 378 Machine to encounter error conditions. You should deal with underflow (performing more operations than there are elements on the stack for the operands). A real machine would cause an exception for these conditions, a somewhat complicated procedure we'll talk about later in the course. Your translated binary should just halt, with a "return code" indicating what happened: 0 means normal termination, 1 means stack underflow. Remember that the output of your program is a text file that contains SPIM codes. You should be able to then load this text file into SPIM itself and watch the 378 Machine binary run.

Evaluating Your Code

Above all, you should write code that is clear, by the normal standards of what is a clear program. You should also strive to write a program that is efficient. When programming in assembler, it is natural to measure efficiency in two ways: low count of instructions executed, and low count of loads and stores to memory.

For this particular program, there is one additional measure that you might keep in mind. Your program is basically a translator for the 378 hardware into MIPS hardware. You should try and make the output as small as possible since you want to execute the 378 Machine as fast as possible!

Turn-In

We'll announce a turn-in procedure for this program, so stay tuned. **TO TEST YOUR CODE, WE WILL ATTEMPT TO TRANSLATE / EXECUTE OUR OWN 378 PROGRAM.** This means your code should be resilient to changes to the 378 Machine binary. You can improve the robustness of your code by testing your code against a variety of 378 programs (for example, that will detect error conditions related to the finite size of the stack) as you code