Lab8: SAM Assembler and Simulator

Due Date: Wednesday April 29th 2009 by midnight

Background:

The *Instruction Set Architecture (ISA)* provides a view of a processor's features as seen from the perspective of an assembly or machine language programmer. For our purposes, this means that the ISA describes the instructions that the processor understands, the way those instructions are presented to the processor, the register set, and the way memory is organized. A real-world processor's ISA would also include a few additional items, such as its interrupt and/or exception handling facilities, basic data types, and different modes of operation, e.g. supervisor vs. normal mode.

Registers are a special type of memory built into the processor. They basically serve as special variables accessible to the *Arithmetic-Logic Unit (ALU)*, the brains of the processor that actually executes most instructions. You'll find that most instructions operate on the values in the registers instead of operating directly on memory. As a result, you'll *load* values from memory into registers, operate on them, and then *store* them back into memory. This arrangement is called a *load-store architecture*.

Assignment:

You are provided with a description of a simple computer's Instruction Set Architecture (ISA), including its instruction set, instruction format, and register list. Your task is to write an assembler (regular assignment) and a simulator (for extra credit) for the described architecture. After completing this assignment, it will be possible to write programs in assembly, process them with your assembler, and execute them on your simulator. We will provide a simulator that can be used to test your assembler

This assignment is designed to help build your understanding of simple processors and the fetch-decode-execute cycle as well as the role of assembly language in software development. It will also provide reinforcement in C, especially in the use of the bit-wise operators and function pointers if you complete the simulator part.

The SAM Instruction Set Architecture

There are seven (7) *general purpose* registers that can be used for any purpose. Additionally, there is a *zero register* that always contains a constant value of 0 to be used for initializing other registers to 0. It is possible for the same register to be read and written within the same instruction, e.g., `A = A + 5` is legal, as is `A = A + A`.

The *program counter (PC)* is a special purpose register that keeps track of the current address in memory, the address that the processor is currently executing. Since instructions are 4 bytes wide, the PC moves forward by four bytes with each instruction cycle. The *instruction register (IR)* is a scratch register used to decode instructions. The PC is 24 bits wide. The IR is 32 bits wide.

The simulated machine has a 2-byte word size, so registers and immediate values are 2 bytes wide. Integers are signed using the high-bit. In other words, the highest bit is 0 if the number is positive and 1 otherwise. This bit is set correctly by the mathematical operations.

The simulated system also has two flags, *overflow* and *compare* which are set by various instructions:

- when executing a mathematical operation, the overflow flag is set to true if an operation overflows (carries) outside of 16 bits, and to false otherwise
- when executing a comparison operation, the compare flag is to true if the comparison operation is true, and it is set to false otherwise.

The flags cannot be set directly.

Ports are a mechanism for accessing input and output devices that are independent from main memory. Port #15 is a terminal device console used for output. Port #0 is a terminal device console used for input. Each reads or writes one character at a time, translating from that character to its corresponding ASCII value. The terminal device has sufficient buffering to avoid dropping character in normal applications.

The system's main memory is byte-addressable. In other words, bytes are addressed and addresses range from byte 0 through byte 2^{24}-1 (more than big enough for our purposes).

Ports

Purpose	Binary	Notes
Input	0000 0000	Returns ASCII code of character read from terminal
Output	0000 1111	Writes ASCII code of character to terminal

Registers

Register	Number	Notes
Z	000	Constant: Always zero (0)
A	001	
B	010	
C	011n	
D	100	
E	101	
F	110	
G	111	
PC		Program Counter. 24 bits wide. Not addressable
IR		Instruction Register. 32 bits wide. Not addressable.

Instructions

```
---------------Control------------

Instruction     -Op- -------------Address--------------        Notes

HLT     0000 XXXX XXXX XXXX XXXX XXXX XXXX XXXX        Stop simulation

JMP     0001 0000 AAAA AAAA AAAA AAAA AAAA AAAA        Jump (line number)

CJMP    0010 0000 AAAA AAAA AAAA AAAA AAAA AAAA        Jump if true

OJMP    0011 0000 AAAA AAAA AAAA AAAA AAAA AAAA        Jump if overflow
```

```
------------Load/Store------------

Instruction    -Op- Reg0 ------------Value------------

LOAD    0100 0RRR AAAA AAAA AAAA AAAA AAAA AAAA   Load (hex address)

STORE   0101 0RRR AAAA AAAA AAAA AAAA AAAA AAAA   Store (hex address)

LOADI   0110 0RRR 0000 0000 IIII IIII IIII IIII   Load Immediate

NOP     0111 0000 0000 0000 0000 0000 0000 0000   No operation

-----------------Math--------------
Instruction    -Op- Reg0 Reg1 Reg2 0000 0000 0000 0000

ADD            1000 0RRR 0RRR 0RRR 0000 0000 0000 0000   Reg0 = (Reg1 + Reg2)

SUB            1001 0RRR 0RRR 0RRR 0000 0000 0000 0000   Reg0 = (Reg1 - Reg2)

-----------Device/IO--------------

Instruction    -Op- Reg0 0000 0000 0000 0000 ---Port--

IN      1010 0RRR 0000 0000 0000 0000 PPPP PPPP   Read Port into Reg0

OUT     1011 0RRR 0000 0000 0000 0000 PPPP PPPP   Write Reg0 out to Port

------------Comparison-------------

Instruction    -Op- Reg0 Reg1

EQU     1100 0RRR 0RRR 0000 0000 0000 0000 0000   Cflg = (Reg0 == Reg1)

LT      1101 0RRR 0RRR 0000 0000 0000 0000 0000   Cflg = (Reg0 < Reg1)

LTE     1110 0RRR 0RRR 0000 0000 0000 0000 0000   Cflg = (Reg0 <= Reg1)

NOT     1111 0000 0000 0000 0000 0000 0000 0000   Cflg = (!Cflg)
```

Writing and Assembling a Program by Hand

A program is a text file with one instruction per line. Each line should be a very simple space-delimited line. It can include comments, which begin with a #. When you first write out the program by hand, number the lines, **ignoring blank and comment-only lines**. Use the line numbers in place of addresses for jumps.

```
# This program gets two single-digit numbers, A and B, from the user
# Then prints out the numbers A through B
 0 LOADI       A       1       # Get the number 1 into register A
 1 LOADI       B       48      # 48 is int value of '0', pseudo-constant
 2 IN          C       0       # Get starting point in ASCII
 3 SUB         D       C       B       # Get integer value of input character
 4 IN          C       0               # Get ending point in ASCII
 5 SUB         E       C       B       # Convert ending from ASCII to int val
# Starting value is D, ending value is E
 6 LTE  D      E               # (D <= E)
 7 NOT                         # !(D <= E) --> (D > E)
 8 CJMP {Line 13}              # If (D > E) from above, exit loop
 9 ADD         C       D       B       # Convert D as int into ASCII
10 OUT         C       15              # Print out the number
11 ADD         D       D       A       # Increment D
12 JMP  {Line 6}              # Go back to the top of the loop

13 HLT
```

Once you are done writing out the program, multiple each line number by 4. This will give you the address of that line of code within memory. This is because each instruction is 4 bytes long. Rewrite the program replacing the line numbers with addresses in hexadecimal.

```
# This program gets two single-digit numbers, A and B, from the user
# Then prints out the numbers A through B
 0 LOADI       A       1       # Get the number 1 into register A
 4 LOADI       B       48      # 48 is int value of '0', pseudo-constant
 8 IN          C       0       # Get starting point in ASCII
 C SUB         D       C       B       # Get integer value of input character
10 IN          C       0               # Get ending point in ASCII
14 SUB         E       C       B       # Convert ending from ASCII to int val
# Starting value is D, ending value is E
18 LTE  D      E               # (D <= E)
1C NOT                         # !(D <= E) --> (D > E)
20 CJMP 34                     # If (D > E) from above, exit loop
24 ADD         C       D       B       # Convert D as int into ASCII
28 OUT         C       F               # Print out the number
2C ADD         D       D       A       # Increment D
30 JMP  18                     # Go back to the top of the loop
34 HLT
```

Now, convert this program into binary, by translating each mnemonic into the binary equivalent shown in the "Instructions" section. Do the same with each value.

```
# This program gets two single-digit numbers, A and B, from the user
# Then prints out the numbers A through B
# Get the number 1 into A register

# 0     LOADI   A       1
0110    0001    0000 0000 0000 0000 0000 0001
# 4     LOADI   B       48              # Subtract 48: ascii char -> int value
0110    0010    0000 0000 0000 0000 0011 0000
# Get starting point in ASCII from port 0
# 8     IN      C       0
1010    0011    0000 0000 0000 0000 0000 0000

# Get integer value of input character
# C     SUB     D       C       B
1001    0100    0011    0010 0000 0000 0000 0000

# Get ending point in ASCII from point 0
# 10    IN      C       0
1010    0011    0000 0000 0000 0000 0000 0000

# Convert ending from ASCII to int val
# 14    SUB     E       C       B
1001    0101    0011    0010 0000 0000 0000 0000



# Starting value is D, ending value is E
# (D <= E)
# 18    LTE     D       E
1110    0100    0101 0000 0000 0000 0000 0000

# !(D > E) --> (D > E)
# 1C    NOT
1111    0000 0000 0000 0000 0000 0000 0000

# If (D > E) from above, exit loop
# 20    CJMP    34
0010    0000 0000 0000 0000 0000 0011 0100

# Convert D as int into ASCII
# 24    ADD     C       D       B
1000    0011    0100    0010 0000 0000 0000 0000

# Print out the number to port 15
# 28    OUT     C       F
1011    0011    0000 0000 0000 0000 0000 1111

# Increment the number
# 2C    ADD     D       D       A
1000    0100    0100    0001 0000 0000 0000 0000

# Go back to the top of the loop
# 30    JMP     18
0001    0000 0000 0000 0000 0000 0001 1000

# 34    HLT
0000    0000 0000 0000 0000 0000 0000 0000
```

Lastly, convert the binary representation into hexadecimal. This is a fully assembled program and is what you will output as `output.o`. Each line represents a single 4-byte instruction. The first line resides at address 0, the second line resides at address 4, the third at address 8, and so on (although real-world computers use an actual binary representation without new lines, we think you'll appreciate this format which captures the same information in a more human-readable, and debuggable, form):

```
0x61000001
0x62000030
0xA3000000
0x94320000
0xA3000000
0x95320000
0xE4500000
0xF0000000
0x20000034
0x83420000
0xB300000F
0x84410000
0x10000018
0x00000000
```

The Assembler

Your assembler is called `samas`. It accepts an assembly source file, parses it, and translates it into an executable object file. It uses the same process as you used by hand. In other words, the program parses each line of the source file and translates it into hexadecimal (i.e., each op code is recognized, looked up in a table, translated, and outputted and then each operand is recognized, translated, and outputted). This is your regular assignment

The name of the input and output files are specified at the command line, for example:

```
samas input.s output.o
```

We have provided two sample input files to start with, input1.s and input2.s. input1.s has no comments or blank lines and input2.s has comments and blank lines (probably easier not to worry about these things in your initial attempt at parsing the file).

The Tester

The second part of your assignment is to write three assembly programs in the simulated language and test to see if they produce the desired output when executed as follows.
1. First write the assembly program (see readme.txt for suggested programs)
2. Second assemble the program using samas **you wrote**
% samas yourprogram.as output.o

3. Third run the program using samsim executable **(that we provided)**
% samsim 10000 output.o -- 10000 is the memory size

In the extra credit section you will get a chance to write your own version of samsim.

EXTRA CREDIT

The Simulator (extra credit)

The simulator models the processor, the main memory, and the described terminal devices via ports. When run, it loads an assembled program into memory and simulates its execution until it halts.

Memory

Memory can be simulated as simply an array of unsigned chars (1 byte) of this size. This provides a byte-indexed memory. In order to interpret the lower addresses that contain the program text, you can assign an `unsigned int` (4 byte) pointer to the same array. This way, when accessing word-oriented instructions, you can use the `unsigned int *` to give you a whole word at a time (just be careful about pointer arithmetic!).

Since relatively few programs will require a simulation of the entire physical memory, the program should accept the size of physical memory as a command line argument. It need only simulate the requested amount of memory.

Loading A Program Into Simulated Memory

To load the program, read each input line into memory. We suggest that you first do this by reading it into a temporary variable, an unsigned int, and then copying this into the unsigned char array simulating memory. The code below illustrates the idiom:

```
unsigned int instruction;
fscanf(file, "0x%x", &instruction);
memcpy(memory + address, &instruction, sizeof(instruction));
address += 4;
```

You are, of course, free to take a different approach. But, we strongly suggest the technique above — it dodges some potentially complicating issues.

The Register File

Since general purpose registers are 16 bits wide, they can be implemented as an array of unsigned shorts (2 bytes). This way the register number can serve as the index. Since the special purpose registers are larger, they should be implemented using unsigned ints (4 bytes).

The Processor and Execution

Assuming that all of the instructions have been loaded into memory, the processor can be simulated using a fetch-decode-execute work loop. During execution, the processor fetches the next instruction by loading the instruction referenced by the PC from memory into the *instruction register (IR)*. This is simply a scratch register used by the processor to decode the instruction.

Once this is done, the PC is incremented to prepare for the eventual next processor cycle. The next step is to decode the op code, the 4-bit number associated with the operation. This can be done by shifting right to eliminate the other bits. Careful here — unless you are using unsigned ints, you'll get bitten while shifting because of the sign bit.

At this point, you are able to dispatch the instruction. Because one goal of this assignment is to reinforce your understanding of function pointers, you are asked to use an array of function pointers for this purpose.

For each instruction, you should create a function. Then, each of these functions should be mapped into an array of function pointers, where each function's index is its op code. This makes the dispatch very easy. For example, if your array is called `ops`, the dispatch is as easy as `(* ops[opcode])()`, i.e.,

```
ops

0 --> HLT()
1 --> JMP()
2 --> CJMP()
3 --> OJMP()
4 --> LOAD()
...
15 --> NOT()
```

Once within the instruction, you'll need to decode the operands and execute. Before considering the implementation, take a second look at the instructions. Notice that there are six different instruction formats. It will probably be helpful to write macros that use bit masks to decode the operands present in each of the six formats:

```
ADDRESS
IMMEDIATE
REG0
REG1
REG2
PORT
```

For example:

```
#define REG0 ((IR >> 24) & 0x0F)
```

Once the operands have been decoded, you are free to implement the logic of the instruction. Perform any needed computation (be careful about math and negative numbers!), then write back the values to the registers. If the instruction is a jump you'll need to update the PC. The

simulation ends when the HLT instruction is called. This instruction should exit rather than returning. Once the function implementing the operation returns, the simulation can loop back to the beginning of the fetch-decode-execute loop and repeat.

The Terminal Devices via Ports

The terminal devices are simulated only within the IN and OUT instructions. Implementing the OUT instruction is as simple as `printf("%c", ...)`. The IN instruction can similarly be implemented using `ch = getchar( )`.

Running the Simulator

The simulator should be called `samsim`. It should actually load and then execute a correct, assembled program. It should be implemented in accordance with the model described above.

The physical memory size and executable file name should be specified as command-line arguments:

- argv[1] - provides the size of your physical memory in bytes
- argv[2] - provides the name of the assembled program

For example:

```
samsim 0x1000 output.o
```

The simulator model does not include an exception handling facility. As a consequence, it cannot handle error states, such as invalid executables, bad memory accesses, and the like. Should any of these circumstances arise, it should simply terminate with an informative error message.

Downloading Files: As usual, download files from /afs/andrew/course/15/123/downloads/lab8

Handing in your Solution

Your solution should be in the form of a .zip file. Just zip and HANDIN all the .c AND.h and .as (assembly) files for your program.

/afs/andrew/course/15/123/course/handin/lab8/ID

Don't hand in any input or .txt or .o or exe files.

Modified from a write up by Greg Kesden