CMPT 201 - Practical Programming Methodology
Winter 2014 - Assignment 2
Due Date: TBA

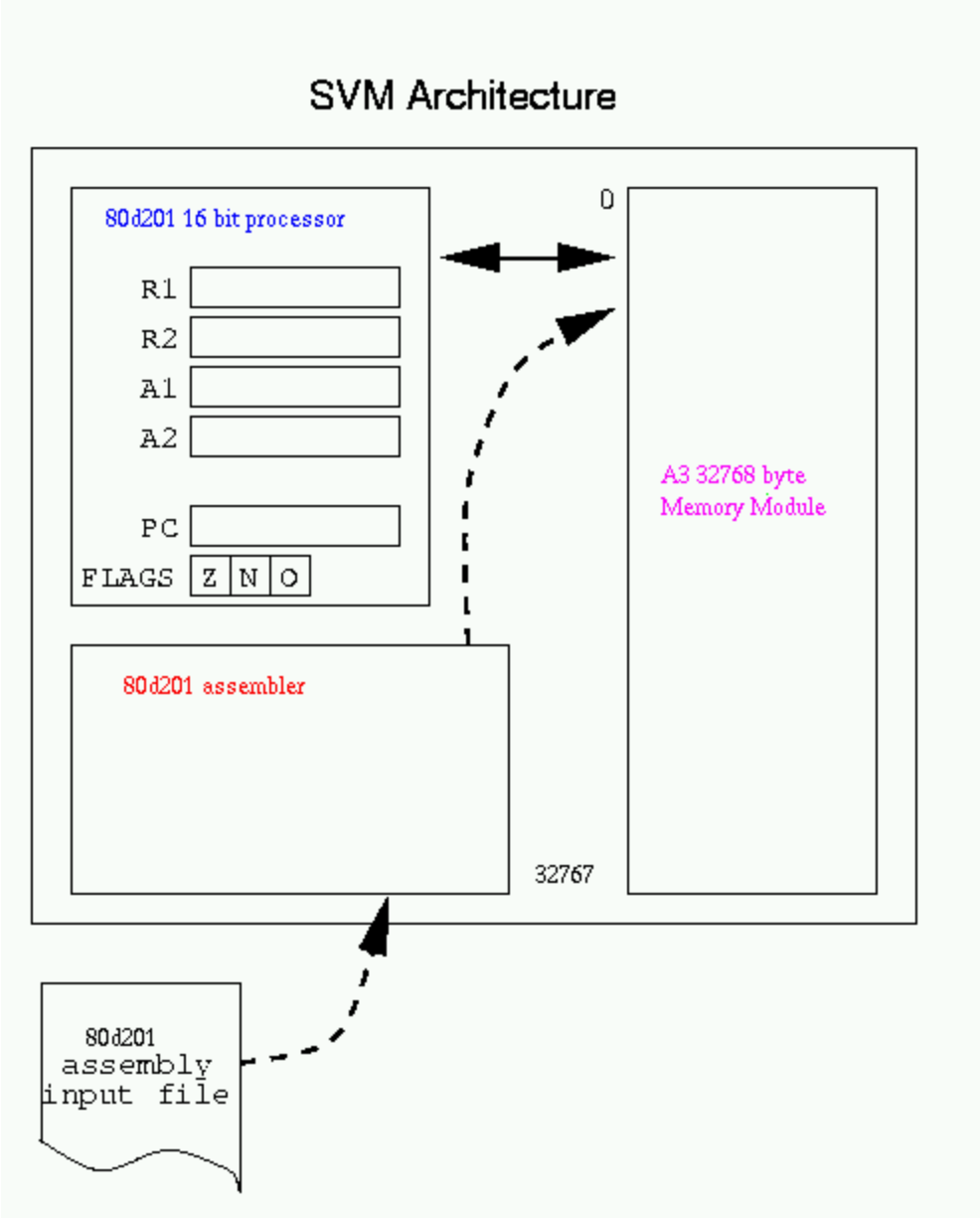
Objectives: A simple machine: its assembly language and its assembler

Key concepts: practical programming and design

The most important aspect of programming is the design phase. The assignment is an exercise in practical programming and design in that you must design and implement a program to solve a problem according to specifications. You are to do this assignment individually using C (C99).

Simple Virtual Machine (SVM)

Implement an assembler for a simplified virtual machine. The svm is capable of taking an input file containing svm machine code, converting this to a program in the machine's simulated memory, and then executing the program. The machine code is created by your assembler from assembly langauge source code as specified below. The svm provides only limited support for screen output, and no run-time input facilities are available. The virtual machine works much like a real system.



- **SVM Specification:**
  - **StunTel 80d201 Processor:**

    The SVM makes use of the fictional StunTel 80d201, a 16-bit processor (instructions and data are limited to 16-bit values) with 2 data registers (called R1 and R2), 2 address registers (called A1 and A2), a program counter PC and 3 flags (Z, N and O). The processor only performs arithmetic operations on values in the data registers, while address registers, as the name suggests, are used to hold addresses so that data values can be referenced indirectly (a pointer if you like).

A *cycle* for the processor consists of the following steps:

1. Fetch the next instruction from memory (indexed by the program counter) and increment the program counter
2. If the instruction requires fetching additional data, perform this movement of data (the destination register for the result of the operation for instance)
3. Perform the operation specified by the instruction, setting any flags to reflect the state of the result (Z = result is 0, N = result is negative, O = result is overflow---NOTE: not all flags will have meaning for all operations), possibly adjusting the program counter in the case of a JMP instruction.

- **A3 Memory Module:**

The SVM uses the imaginary A3 32768-byte memory module. This memory is 8-bits wide, meaning that each memory location corresponds to a 1-byte chunk of data, addressed from 0 to 32767.

- **Loader:**

The SVM uses a very simple loader to get a program into memory and execute it. All the loader does is read in 80d201 instructions from a file and load them into memory starting at location 0. It then starts execution of the program which it loaded.

- **80d201 Assembler (created by you):**

The SVM utilizes an assembler that converts 80d201 assembly language input into a sequence of 80d201 instructions and outputs the instructions in a form suitable for the loader to place into memory. Instructions with operands have their operand data appearing in successive locations immediately following the instruction itself. Program instructions should begin at address 0, and can occupy a maximum of 32767 bytes, because of memory size constraints.

- **80d201 Instruction Set:**

There are seven basic classes of instructions, and one way of describing data items.

- **LOAD** - copies values into registers

LOAD reg1,immSRC

Load immediate to register reg1 - load the provided constant value "immSRC" into the specified register (data or address)
LOADI reg1,reg2
Load indirect register - load the data contained in the memory address specified in address register reg2 into the specified register (data or address)

- **STORE** - copies values into memory locations

  STORE reg1,immDEST
  Store immediate from register - store the data contained in the specified register (data or address) into the memory address specified by the provided constant value "immDEST"
  STOREI reg1,reg2
  Store indirect from register - store the data contained in the register reg1 (data or address) into the memory address specified in address register reg2

- **JMP** - jumps (conditional/unconditional)

  JMP immDEST
  Unconditional jump - jump to constant valued address "immDEST"
  JMPZ immDEST
  Jump if zero - jump to constant valued address "immDEST" if Z flag is set (i.e. as result of previous operation)
  JMPN immDEST
  Jump if negative - jump to constant valued address "immDEST" if N flag is set (i.e. as result of previous operation)
  JMPO immDEST
  Jump if overflow - jump to constant valued address "immDEST" if O flag is set (i.e. as result of previous operation)

- **ADD** - addition to a data register destination

  ADD reg1,immSRC
  Add immediate to register - add constant value "immSRC" to the value, and store the result, in data register reg1. Z, N and O flags should be set appropriately as the result of the operation
  ADDR reg1,reg2
  Add register to register - add value in data register reg2 to the value, and store the result, in data register reg1 (reg2 and reg1 cannot be the same register). Z, N and O flags should be set appropriately as the result of the operation

- **SUB** - subtraction from a data register destination

  SUB reg1,immSRC
  Subtract immediate from register - subtract constant value "immSRC" from the value, and store the result, in data register reg1. Z, N and O flags should be set appropriately as the result of the operation

SUBR reg1,reg2
Subtract register from register - subtract value in data register reg2 from the value, and store the result, in data register reg1 (reg2 and reg1 can not be the same register). Z, N and O flags should be set appropriately as the result of the operation

- **OUT** - output to the screen, that is `stdout`

  OUT immSRC
  Output immediate value - output constant value "immSRC" to the screen as an 16-bit signed integer value (i.e. a number).
  OUTC immSRC
  Output immediate character value - output constant value "immSRC" to the screen as an 8-bit unsigned character value (i.e. an ASCII character).
  OUTR reg1
  Output register value - output value in data register reg1 as an 16-bit signed integer value (i.e. a number).
  OUTRC reg1
  Output register character value - output value in data register reg1 as an 8-bit unsigned character value (i.e. an ASCII character).
  OUTI reg1
  Output indirect value - output value in the memory address referenced by the provided address register reg1 as an 16-bit signed integer value (i.e. a number).
  OUTIC reg1
  Output indirect character value - output value value in the memory address referenced by the provided address register reg1 as an 8-bit unsigned character value (i.e. an ASCII character).

- **HALT** - shutdown the machine

  HALT
  The HALT instruction is used to mark the end of program. When your simulator of the 80d201 processor executes the HALT instruction it should exit.

- **DATA** - reserve space for data

  DATA immSRC
  Data word - corresponding memory location will contain the constant value "immSRC" (this isn't really an instruction, and should not be converted to an instruction in memory, the intention is to allow this statement to reserve one 2-byte word at this point in memory for the specified data).

NOTE: Besides being affected by the ADD and SUB instructions, the flag bits will also be set appropriately after loading or storing. Jumps and output instructions do not change the flag bits.

• StunTel 80d201 Assembly Language Format:
An assembly language file input to the **SVM** has the following format:

```
1       89+
```

```
LABEL   INSTRUCTION OP1[,OP2] # comment (newline)
```

Columns 1 to 7 are reserved for "labels" - named memory locations that will be resolved by the assembler as it generates the executables to be later loaded into memory. Labels can be used as "constant values" anywhere one might be expected. In the process of assembling the source code, all occurrences of labels are replaced by the memory address of the instruction they precede. This is convenient for having "named" data storage locations in memory, or as the targets of JMP instructions. Labels in the source code always resolve to addresses in memory. Column 8 must be a space to separate the label from the instruction. The first nonblank will appear in column 1 if a label is present.

Instructions start in column 9, and have the format specified in the preceding section. They are terminated by a newline. The '#' is a comment character and anything appearing after it on a given line is to be ignored. All names in the assembly language are case sensitive. The first nonblank will appear in column 9 if an instruction is present.

For the purposes of this assignment, you are not responsible for the exhaustive checking of the syntax or semantic restrictions of input files--assume they are correct programs with correct formatting. Do note however, that rudimentary sanity checking will help you when you are writing your own little programs to test your implementation.

• **Example:**

Here is an example of a small program that fetches two values from memory, outputs them, adds them together, stores the result back to memory, fetches the result from memory and prints it out:

```
        LOAD  A1,DATA1   # load address DATA1 into A1

        LOADI R1,A1      # load contents of address in A1 into R1

        LOAD  A2,DATA2   # load address DATA2 into A2

        LOADI R2,A2      # load contents of address in A2 into R2

        OUTI  A1         # output contents of address in A1 as a number

        OUTC  43         # output ASCII character 43 ('+')

        OUTI  A2         # output contents of address in A2 as a number

        OUTC  61         # output ASCII character 61 ('=')

        ADDR  R1,R2      # add contents of register R2 to R1 and store in R1

        STORE R1,RESULT  # store contents of register R1 to address RESULT

        LOAD  A2,RESULT  # load the address of RESULT

        LOADI R2,A2      # load the value of RESULT

        OUTR  R2         # output contents of register R2 as a number

        OUTC  12         # output a carriage return
```

```
        OUTC  13        # output a line feed
        HALT            # Adieu, mon ami!
DATA1   DATA  4
DATA2   DATA  3
RESULT  DATA  0
```

The result is the following output:

```
4+3=7
```

• **Instruction format in Memory:**
The 80d201 instructions are encoded as opcodes - simply a number that corresponds to each instruction. Registers are also encoded as numbers. The mapping is as follows:

```
/*  opcodes for 80d201 processor */


#define HALT 0x61


#define LOAD 0x80


#define LOADI 0x81


#define STORE 0x82


#define STOREI 0x83
```

```
#define JMP 0x84

#define JMPO 0x85

#define JMPZ 0x86

#define JMPN 0x87

#define ADD 0x88

#define ADDR 0x89

#define SUB 0x8a

#define SUBR 0x8b

#define OUTI 0x8c
```

```
#define OUTR 0x8d

#define OUTIC 0x8e

#define OUTRC 0x8f

#define OUT 0x90

#define OUTC 0x91



/* Registers for 80d201 processor */



#define A1 0

#define R1 1
```

```
#define A2 2
```

```
#define R2 3
```

16 bit integer and address values are represented in *little-endian* order. This means that the byte ordering for the 80d201 is the same as the Intel machines you are working on, and so a 16 bit integer on the 80d201 should be the same as a `short` on the Intel machines (experiment to verify :-)).

Instructions in memory all occupy either 2 bytes, or 4 bytes. Instructions with an immediate argument occupy 4 bytes, those with only register arguments occupy 2 bytes. The DATA statement reserves one 2 byte word (remember DATA is not an instruction)

- The first byte is the opcode of the instruction
- The second byte is used to encode both register arguments, with the first register argument in the rightmost 2 bits of the byte, and the second in the leftmost 2 bits of the byte. This means that the middle 4 bits of the second byte are not used. These bits are reserved for possible future enhancements to the 80d201 processor line if enough units are sold to whoever is crazy enough to buy them.
- For instructions with immediate operands, two bytes following the instruction are used to hold the 16 bit data or address value used by the instructions. The result of this is all instrutions encode to either 2 or 4 bytes.

The second byte of register-argument instructions is somewhat tricky. To encode two register values into one byte, you must use the rightmost 2 bits to encode the first register argument (reg1), and you use the leftmost two bits of it it to encode the second argument (reg2). Since the registers are only numbered 0 - 3 it will be possible to represent them in 2 bits each. Again, this means that the middle 4 bits of the byte are not used. You will need to use bitwise operators to place a representation of registers into the appropriate bits of the second byte, and to extract them correctly.

As previously mentioned the DATA instruction doesn't have an opcode or registers. It simply puts a 16 bit value into the two bytes of memory where it occurs. (This means of course, that programs for the machine probably shouldn't try to execute those bytes as instructions - but this is the programmer's problem, not yours).

- **Design issues:**

With the assembler you must do text replacement with labels. This is easiest to accomplish with the use of a symbol table (a hash table mapping character strings to corresponding addresses will

suffice; of course, a linked list of pairs might work too, it's up to you to decide.). Note that you will probably need to do two passes over the input file, because you will not have a complete mapping of labels to addresses before the first one is complete.

It is essential that you formalize your interfaces as early as possible to minimize potential problems later on. You will also benefit from considering each aspect of the system as independently as possible, rather than trying to solve the entire problem at once and in one place. It is quite straightforward to build dummy routines that can feed pre-processed data to each part so that they can be developed somewhat independently, without the entire system necessarily working together at first. It is important that you make progress and develop as much of the system as you can properly, since this will be worth much more than an attempt at everything that produces just a mess.

One of the first issues you will have to deal with is the representation of the instructions in memory. The assembler converts the instructions to their opcode values, which are later fetched by the processor which will switch on the value to perform the appropriate operation for the instruction in question. Common constants and data structures should be placed in a shared header file that is finalized as early as possible in the development process, in the interest of standardizing the interface between different components

- **Implementation Guidelines:**

The svm assembler should be invoked to assemble assembly language source file `myprog.svm` as follows (use the .svm extension, why not?):

```
cat myprog.svm | sasm > myprog
```

That is, the `sasm` assembler should read the program to assemble on standard input, and write out 80d201 instructions on standard output.

Our `svm` machine simulator should be invoked to run the 80d201 instructions in your `myprog` as follows:

```
cat myprog | svm
```

That is, the `svm` machine simulator should read in the instructions to execute from standard input.

The `sasm` should read the program from the standard input, converting the instructions to their representation in memory and writing them out to standard output.

You should be able to assemble and run a correct program `myprog.svm` by issuing the commands:

```
cat myprog.svm | sasm | svm
```

It is possible that an illegal or incorrect program gets entered---it would be nice if the `sasm` could detect such situations and shut down with an error message, however this will not be possible in

many situations without considerable effort which is not the point of the assignment. Again, you are reminded to assume the input is correct (at least in terms of syntax---debugging the program is another issue...) This means that *it is not required that your assembler needs to detect syntax errors in the program*. However, if you want to stay sane while you are testing your assembler you will probably attempt to detect at least the first syntax error.

- **We give you:**

  - On blackboard (released as the assignment winds on) you will find the following:

    `sasm`
    the executable of our assembler,
    `svm`
    the executable of our simulator,
    `Test1.svm`
    the example program from above,
    `factors.svm`
    a program for finding all factors of a number,
    `pfactors.svm`
    a program for finding all prime factors of a number

  - Your assembler will be tested against our implementation of the machine using both public (see above) and private test programs.

- **Packaging:**

For Assignment 2, submit **as2.tgz** containing the following files:

- a file called README explaining what is the contents of each file you have committed to the repository, this includes where to find all the files mentioned below.
- Makefile (with comments as needed), `make all` should produce all the executable of sasm,
- all source files,
- design document (a text file called `design.txt` or a pdf file called `design.pdf`) describing the design decisions you made; this should include a discussion of your data structure(s),
- testing report (a text file called `testing.txt` or a pdf file called `testing.pdf`, describing how and what you tested)
- your regression tests for this assignment in a directory called `Tests`, (the test files you use from above).

- **Marking Scheme**

Task	Marks
Our testing of your programs	60
Readme file	5
Makefile	10
design document	15
testing report and tests	10
Total marks	100

In addition to the above, up to 50% of the total marks may be deducted for poor coding style or program design.

The late policy applies to this assignment: you have one day past the due date with a 20% penalty, after that assignment will not be accepted, unless there is a special personal emergency recognized by your instructor.

Notes on Marking

1. **Packaging:**
   - Makefile must work perfectly, so that all executables are created when "make all" run with a checked out copy of your submission. You should test this yourself. **NO EXECUTABLE FILES SHOULD BE SUBMITTED - ONLY SOURCES AND TEXT FILES OR MARKS WILL BE DEDUCTED - no executables, core dumps or compiled programs (C or SASM) should be in there)**
   - Code must compile with no avoidable warnings under -Wall -std=c99. We expect to see your Makefile using these flags when compiling. Marks will be deducted if it does not.
2. **Design**
   - Were design decisions documented?
   - Was a coherent design methodology attempted and how well did it work?
3. **Coding:**
   - Produces reasonable output, even with unreasonable input (i.e., exits with error, warns about problems).
   - Prints out warnings upon detection of problems (i.e., `"Couldn't open file 'blah'!"`).
   - Marks will be deducted for significant coding errors, such as:
     - Not checking the return code where it should be checked.
     - Not describing the invariants for a non-trivial data structure
     - Failure to document a non-obvious loop. This documentation must convince the reader of the correctness of the loop.
4. **Testing Strategy**:
   - Any programs (source code) used in order to test programs must be included. We expect you to submit the regression tests you are using and explain why you believe them to be a satisfactory test of your program.
5. **Style**:

- All source files must have headers giving the usage information and purpose of the program.
- "Bad" documentation (spurious, misleading or just plain wrong) will be penalized.
