CS 4380 Project 1: 4380 Emulator Mark 1

NOTE: READ THIS DOCUMENT IN ITS ENTIRETY BEFORE YOU BEGIN WORKING

Introduction:

In this project you will begin the incremental development of a microprocessor emulator that you will build upon throughout the semester and utilize in CS 4490. This emulator simulates the execution of a hypothetical processor we are calling the 4380. At this stage you will implement the Mark1 version of the 4380, which has the following high-level architectural characteristics….

Instruction Size: Fixed at 8 Bytes (64 bits)

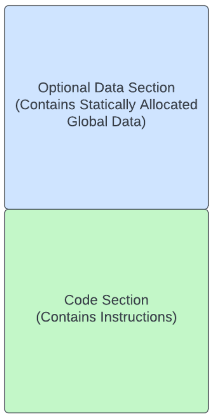
1 Byte Operation	1 Byte Reg Operand 1	1 Byte Reg Operand 2	1 Byte Reg Operand 3	4 Byte Immediate Value

Registers (22):

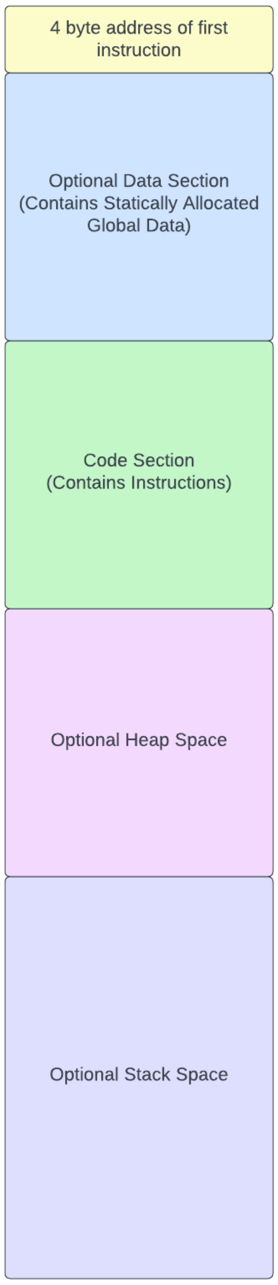
Name	Encoding	Description (all registers are 32 bits in size)
R0 – R15	0 - 15	General purpose registers, **initialized to 0 at startup**
PC	16	Program Counter, **initialized to the address of first instruction in memory**
SL	17	Stack Lower limit (lowest legal address available to the stack)
SB	18	Stack Bottom (highest address)
SP	19	Stack Pointer (latest allocated byte on the stack, grows downward)
FP	20	Frame Pointer (points to the first word beneath the return address)
HP	21	Heap Pointer (initially set to SL, grows upward))

Programmable Memory: 32 bit address space; Byte addressable; Little-endian; User specifiable; Defaults to 2^17 (131,072) bytes of addressable memory.

Programmable Memory Layout: At runtime the first four bytes of the 4380's programmable memory must contain the address of the first instruction to be executed. Strictly speaking this is the only requirement regarding the layout of programmable memory at runtime. However - 4380 assembly programs must adhere to the following code organization:



This reality lends itself to the following convention for programmable memory layout of the 4380 at runtime:


4 byte address of first instruction
Optional Data Section (Contains Statically Allocated Global Data)
Code Section (Contains Instructions)
Optional Heap Space
Optional Stack Space

While much of this information (regarding memory layout) is not critical to successful completion of this project, we have included it here to help establish context and forward thinking. It is worth noting that the stack and heap regions of the 4380 **will** be managed via the associated special purpose registers already described above, but this work will be completed in future projects (in other words it is safe to simply initialize them as specified and then treat them as general purpose registers them for now).

Instructions:
The following tables describe the instructions supported by the 4380 Mark 1. Operators are indicated by the value of the first byte of each 8 byte instruction, and are listed in the first column. The associated assembly code operator is listed in the second column as a string, followed by information concerning the 3 single byte operands (columns 3 - 5), and the 32 bit immediate value (column 6). Column 7 provide a brief description of the effects of the instruction.

Note: Several operands are marked as DC, meaning "Don't Care". This means that while the operand is required in bytecode - its specific value doesn't matter to execution of the instruction.

Jump Instructions:

Value	Operator	Operand 1	Operand 2	Operand 3	Immediate Value	Description
1	JMP	DC*	DC*	DC*	Address**	Jump to Address

*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. This label must be resolved to a 32 bit unsigned integer value in bytecode.

Move Instructions:

Value	Operator	Operand 1	Operand 2	Operand 3	Immediate Value	Description
7	MOV	RD	RS	DC*	DC*	Move contents of RS to RD
8	MOVI	RD	DC*	DC*	Imm***	Move Imm value into RD
9	LDA	RD	DC*	DC*	Address**	Load address into RD
10	STR	RS	DC*	DC*	Address**	Store integer in RS at Address
11	LDR	RD	DC*	DC*	Address**	Load integer at Address to RD
12	STB	RS	DC*	DC*	Address**	Store least significant byte in RS at Address
13	LDB	RD	DC*	DC*	Address**	Load byte at Address to RD

*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. This label must be resolved to a 32 bit unsigned integer value in bytecode.
*** When represented in an assembly file the Imm value shall be represented as a numeric or character literal.

Arithmetic Operations:

Value	Operator	Operand 1	Operand 2	Operand 3	Immediate Value	Description
18	ADD	RD	RS1	RS2	DC*	Add RS1 to RS2, store result in RD
19	ADDI	RD	RS1	DC*	Imm**	Add Imm to RS1, store result in RD
20	SUB	RD	RS1	RS2	DC*	Subtract RS2 from RS1, store result in RD
21	SUBI	RD	RS1	DC*	Imm**	Subtract Imm* from RS1, store result in RD
22	MUL	RD	RS1	RS2	DC*	Multiply RS1 by RS2, store result in RD
23	MULI	RD	RS1	DC*	Imm**	Multiply RS1 by Imm, store the result in RD
24	DIV	RD	RS1	RS2	DC*	Perform unsigned integer division RS1/ RS2. Store quotient in RD. Division by zero shall result in an emulator error
25	SDIV	RD	RS1	RS2	DC*	Store result of signed division RS1 / RS2 in RD. Division by zero shall result in an emulator error
26	DIVI	RD	RS1	DC*	Imm**	Divide RS1 by Imm (signed), result in RD. Division by zero shall result in an emulator error

*This operand shall be omitted in 4380 assembly instructions but is required in 4380 bytecode.
** When represented in an assembly file the Imm value shall be represented as a numeric literal.

Traps/Interrupts:

Value	Operator	Operand 1	Operand 2**	Operand 3**	Immediate Value**	Description
31	TRP	DC**	DC**	DC**	#0	Executes the STOP/Exit routine
31	TRP	DC**	DC**	DC**	#1	Write int in R3 to stdout (console)
31	TRP	DC**	DC**	DC**	#2	Read an integer into R3 from stdin
31	TRP	DC**	DC**	DC**	#3	Write char in R3 to stdout
31	TRP	DC**	DC**	DC**	#4	Read a char into R3 from stdin
31	TRP	DC**	DC**	DC**	#98	Print all register contents to stdout

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

Requirements:

As you design and build your emulator you must adhere to the following requirements:

Req1) The emulator must reside in a PRIVATE GitHub repository called cs4380-emulator-assembler.

Req2) Project 1 work must be completed in a separate branch called project-1. When all requirements are met, the branch should be merged into main (BUT NOT DELETED).

Req3) The emulator must be written in C++.

Req4) The emulator shall NOT be implemented as a C++ class. The principle data structures and functions listed in these requirements shall have global visibility.

Req5) The project must utilize cmake/make so that the emulator can be built by invoking these two tools. The resulting executable shall be called **emu4380**.

Req6) The project must contain 'src' and 'include' directories/folder. The src folder shall contain all .cpp files for the project. The include folder shall contain all .h files for the project. Both the src and include folders shall reside in the project's root directory.

Req7) The emulator must accept up to two command line arguments. The first argument shall be required, and shall be the name of a binary file containing 4380 byte code (e.g. binary encoded instructions that adhere to the requirements of the 4380 ISA as provided up to this point). The second argument shall be optional, and shall be the desired size of the emulator's program memory (in bytes) specified as a positive integer less than or equal to 4294967295.

Req8) Upon being invoked with an appropriate input file (and optional memory size) the emulator shall read the contents of the binary file into its Program Memory. The first unsigned int's worth of Program Memory shall then be read into the PC register and execution shall commence. (See Req10) In the event that the specified program memory size is insufficient to store the contents of the binary file the emulator shall print the message "INSUFFICIENT MEMORY SPACE\n" and then exit with a return value of 2.

Req9) The emulator shall implement a FETCH, DECODE, EXECUTE loop as its principal execution pattern. (See Req11)

Req10) Once execution begins the emulator shall continue to execute valid instructions until a TRP #0 instruction is encountered, at which point the emulator shall exit with a return value of 0. If at any point an invalid or unrecognized instruction is encountered the emulator shall print the message "INVALID INSTRUCTION AT: <address>\n" 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.

Req11) The emulator must consist of three C++ files: emu4380.h, emu4380.cpp, and main.cpp.

- emu4380.h shall contain function prototypes as well as declarations (not definitions) for the structural elements of the processor, including at a minimum:
    - A register file represented as an array of unsigned integers called **reg_file**. The general purpose and state registers of the processor shall be represented in order according to the encodings shown in the register table above (i.e. R0 shall be at position 0, SB shall be at position 18 etc.)
    - A program memory implemented as a dynamically allocated array of unsigned char which shall be referenced via an unsigned char* called **prog_mem.** The size of the array shall be specified as the second (and optional) command line argument provided when the emulator is invoked. If no value is provided by the user the size shall default to 131,072 elements/bytes.
    - 5 control registers implemented as an array of 5 unsigned integers called **cntrl_regs** for storing parsed instruction information. These registers shall be utilized to store the instruction operation, register operands, and immediate value respectively.
    - 2 data registers implemented as an array of 2 unsigned integers called **data_regs** for storing register operand values retrieved from the register file.
    - Enum types for accessing/indexing the register file, control, and data registers. The enum types shall be called **RegNames**, **CntrlRegNames**, and **DataRegNames**. The RegNames enum must follow the naming conventions utilized in the register table above, and should associate the appropriate encoding values to the register names.(i.e. R0:0, R1:1… R15:15, PC:16…HP:21). The CntrlRegNames enum must include the following naming associations: OPERATION:0, OPERAND_1:1, OPERAND_2:2, OPERAND_3, IMMEDIATE:4. The DataRegNames enum must include the following naming associations: REG_VAL_1:0, REG_VAL_2.

      It is highly recommended (though not required) that enums be used to associate instruction names with their encoding values within the emulator as well.

- emu4380.cpp shall contain the processor element definitions and function definitions that simulate the execution phases of the 4380 processor, including FETCH, DECODE, and EXECUTE phases. These functions shall have the following prototypes (contained in the emu4380.h file):
  - bool fetch(); // Retrieves the bytes for the next instruction and places them in the appropriate cntrl_regs. Also increments the PC to point to the next instruction. If an invalid fetch address (i.e. out of bounds) is encountered by this function it shall return false. Otherwise it shall return true.
  - bool decode(); // This function shall verify that the specified operation (or TRP) and operands as specified in the cntrl_regs are valid (i.e. a "known" instruction with legal operands). For example: a MOV instruction operates on state registers, and there are a limited number of these; a MOV instruction with an RD value of 55 would clearly be a malformed instruction. This function shall also retrieve register values from the register file and place these values in the appropriate data_regs as indicated by the register operands present in cntrl_regs.

    NOTE: immediate value instructions shall result in immediate value operands being placed in cntrl_regs during the fetch stage. These values shall remain in the cntrl_regs where they shall be sourced during the execute phase vs being placed in data_regs.

    NOTE: Valid instructions are instructions that have been introduced thus far. The Mark 1 emulator shall ONLY accept the instructions described in this document. If an invalid instruction is encountered this function shall return false. Otherwise, it shall return true.
  - bool execute(); // Executes the effects of the decoded and validated instruction or TRP on the state members (regs, memory, etc.) as indicated by cntrl_regs and data_regs, and in accordance with the instruction or TRP's specification. If an illegal operation is encountered the function shall return false (without executing the instruction), otherwise it shall return true.
- emu4380.h and emu4380.cpp shall also include a function declaration and definition (respectively) for the function:

  ```
  bool init_mem(unsigned int size);
  ```

  This function shall dynamically allocate size bytes of memory for the program memory described above, initialize all values in the array to zero, and store the address of this memory in **prog_mem**.
- main.cpp shall #include emu4380.h, and implement a main() function that processes the command line arguments as described in Req7. The first command

line argument is assumed to contain a binary input file containing 4380 byte-code. The function shall read the contents of the file into program memory, initialize the PC register, and then enter an execution loop that calls the fetch(), decode(), and execute() functions() until a TRP 0 or illegal instruction are encountered.

Req12) The output of the TRP 1 and TRP 3 traps shall be printed without any leading or trailing whitespace of any kind.

Req13) The output of the TRP 98 trap shall be formatted as follows:

- One register name and value per line
- The register name shall be in all caps, followed by a tab character, followed by the register value printed as an unsigned base 10 integer. For example:

```
R1	0
R2	128
R3	34
…	…
HP	10045
```

Validation:

As always you are responsible for the validation of your project before submission. Before you begin working take the time to design your emulator. Identify the parts and pieces that will be needed and how they will need to interact. Then figure out how you can validate the behavior of these pieces both in isolation, and when integrated with the other parts of your system. Good modularity lends itself to good testing – and both lend themselves to well built, and maintainable code. Begin building up your test suite NOW, and automate as much of it as possible.

What to Turn In:

When you are completed with 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.**

Note: If you are invoking the Non-Late policy please insert a note at the top of your README file and as a submission comment in Canvas that states “INVOKING THE NON_LATE POLICY”. Within your README file this comment should be followed immediately by your work log. (The rest of you README should still contain any project specified requirements).