Homeworks 3, 4, 5, 6, CSCE 240, Fall 2018

CAVEAT: There is some chance that Homework 5 and 6 might be slightly different from what is written here. You should read the text of those assignments from the pages provided in the test directories, not from this (earlier) version.

The Pullet16 Computer

- The Pullet16 is a machine with 4096 *words* of 16 bits each. There is no distinction between program memory and data memory, no virtual memory, etc. This is a single-user system and the one user at a time able to access the machine gets all of it. There is no memory protection; any user can access all of memory.

- The machine is *word-addressable*, not *byte-addressable*. That is, all addresses refer to 16-bit words, and of course we index zero-up. For example, the (decimal) address of 53 would refer to what might be viewed as the 54-th two-byte word, or bytes at subscript 106 and 107 from subscript zero. We will program this with a word as two bytes, but the Pullet16 doesn't know what a "byte" is.

- There is one 16-bit program counter `PC`. This is the hardware register that contains the address in memory of the next instruction to be executed. This is initialized to 0 at the beginning of execution. The way computers work is as follows: the hardware loads the machine instruction from whatever memory address is stored in the `PC`. It tries to decode and then execute that instruction. Under most circumstances, the load, decode, execute sequence causes the `PC` to be incremented by one (in this computer, by one word, but in other architectures that are Intel-like, perhaps by two, four, six, or eight bytes). Whatever is stored at that memory location is then loaded, decoded, executed, etc. Under some circumstances (like getting to the bottom of a loop and having to branch back to the top of the loop, the instruction itself causes a value other than incremented-by-one to be put in the `PC`, and the hardware then pulls from whatever location that happens to be.

  A "branch" instruction, for example, will have a "target" address. One reads a simple branch instruction as saying to the computer, "Please load the program counter with the target address." After that is done, the hardware pulls the instruction at that new address from the program, which effects the branch.

- We thus assume all executable modules are loaded at raw memory location 0 and all addresses are raw memory addresses.

  (This is in contrast to real computers at present. For some computers, it will appear to the user that the program is loaded at location zero, but behind the scenes there will be an offset register whose contents are added to the program's memory address to find the real address in physical memory.)

  ((And then there is virtual memory, cache, etc. We don't deal with any of that here.))

- You will need to be especially aware of the fact that we have to deal with 12-bit arithmetic and with 16-bit arithmetic here. The Pullet16 is a 16-bit computer and the arithmetic that is done using the `ADD` and `SUB` instructions is 16-bit arithmetic, but memory addresses have to be non-zero integers of 12 bits or less, and your (internal) arithmetic on addresses will be 12-bit arithmetic.

- There is one 16-bit accumulator labelled `ACC`. All relevant operations are of the form:

```
LOAD ACC <--- (contents of memory1)
ACC <--- (contents of ACC) OP (contents of memory2)
STORE ACC ---> (contents of memory3)
```

- Arithmetic is done using 16-bit twos-complement arithmetic. This will require you to be able to handle twos complement.

  We will talk about this.

  YMMV, but my opinion is that it's easier to avoid worrying about twos complement until you actually are doing arithmetic, because if you write code that puts positive unsigned integer 16-bit values into 32-bit variables, you never have to worry about what the compiler will force as twos complement sign extension.

  We will talk about this. It's a lot easier than it might seem at first.

- Memory referencing can be done either with direct addresses or with indirect addresses, in which the contents of a memory location are taken to be not the data but the address at which the data is to be found. The indirection indicator (see layout below) is an asterisk, so, for example,

  ```
  STC   LOC
  ```

  generates an instruction that will store the `ACC` contents at the location of `LOC` in memory, while

  ```
  STC * LOC
  ```

  generates an instruction that will fetch the contents of `LOC` as an address `ADDR` and then store the `ACC` contents at the location of `ADDR` in memory.

There are two machine-instruction formats. These are the binary patterns that are the machine code that is part of the analog to an `a.out` file.

- Machine Code Format I:

bits	0-2	opcode
bit	3	0 value indicates direct addressing, 1 indicates indirect
bits	4-15	memory address in hexadecimal

- Machine Code Format II:

bits	0-2	opcode
bit	3	forced zero
bits	4-15	function selector code

The complete instruction set for the Pullet16 is as follows.

Format I

Mnemonic opcode	Binary opcode	Description
`BAN`	000	Branch on `ACC` negative
`SUB`	001	Subtract contents of memory from `ACC`
`STC`	010	Store `ACC` in memory and then clear `ACC`
`AND`	011	And `ACC` with contents of memory
`ADD`	100	Add contents of memory to `ACC`
`LD`	101	Load `ACC` from contents of memory
`BR`	110	Unconditional branch

Format II

Mnemonic opcode	Binary opcode	Hex opcode	Description
`RD`	1110 0000 0000 0001	E001	Read from standard input into `ACC`
`STP`	1110 0000 0000 0010	E002	STOP execution
`WRT`	1110 0000 0000 0011	E003	Write from `ACC` to standard output

Note that although there appear to be two formats, the "real" opcode comprises the first three bits, which decode to integers zero through seven. The second format has `111` as those three bits, and a forced zero bit for the next bit.
(Note that I have put in blank spaces in the Format II binary opcode to make it more readable.)

There are also assembler pseudo-op instructions. We won't deal with these until homework 5.

- `ORG` – Set program counter to the value of the operand.
- `END` – End of input.
- `HEX` – Define a constant to be stored at the current `PC` location.
- `DS` – define storage of n words beginning at the current `PC` location.

The program format for lines of input is as follows.
If column 1 is an asterisk, the entire line is a comment.
Otherwise, the format is for fixed columns:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
l	l	l	b	m	m	m	b	a	b	s	s	s	b	pm	h	h	h	h	b	c

with

- lll = optional label, left justified, which is alphanumeric begining with an alpha character
- b = blank space
- mmm = the mnemonic opcode
- a = blank (direct addressing) or asterisk (indirect addressing)
- sss = optional symbolic operand, same rules as for labels

- pm = plus sign or minus sign for the optional hex operand
- hhhh = the four hex digits of the optional hex operand
- c = comment (and continuing beyond column 21)

All operands are in hex. Legal hex operands are all five characters in length. The first character is either + or −. The next four characters are the four hex digits of the operand and must be one of `0123456789ABCDEF`.

Direct addressing means that the contents at the operand location is the value of the operand. Indirect addressing means that the contents at the operand location is itself an address at which the value is stored. Thus

```
ADD   XX1
```

means that the contents at `XX1` are to be added to the accumulator.

```
ADD * XX1
```

means that the contents at `XX1` are to be taken as an address. If that contents is, say 12, then whatever is stored at location 12 (hex) is to be added to the accumulator.

A Simplification

Note that the hex operands are

```
+xyzw
```

or

```
-xyzw
```

of 16 bits in length, and that the digits x, y, z, w, are as above. But since this is twos-complement arithmetic, the lead bit is what indicates whether the value is positive or negative. That is, the hex value

```
FFFF
```

that is all 1-bits is the twos-complement representation of -1 in the Pullet16.

So how should

```
-FFFF
```

be interpreted? The programmer who wrote this would write the hex for a number that is inherently negative, and then negate that.

The simplification that you should probably adopt, and that is perfectly ok to adopt, is to treat the programmer as a total idiot but not to second-guess him. Just do what he says. That is, read and process the $FFFF$ to create a 16-bit number that is -1, and then negate that value to produce a $+1$. This way you don't have to do logic in parsing the hex.

A (Probable) Simplification

It may very well simplify things if you DON'T in fact try to deal with binary code right at first. If you do that, you will have to deal with binary bit patterns in bytes and/or short integers, and as you develop your code you will have to read these binary bit patterns to determine where the errors are.

If instead you treat the "machine code" as 16-character strings of zeros and ones, then you will be able to look at the "machine code" and probably make sense of it more easily than if you went to binary. This will require you have helper functions for decimal to string and string to decimal conversion, but it will allow you to see bit patterns without thinking in hex arithmetic.

If you really wanted to kluge the binary output, you could do all the assembly as strings of zeros and ones and then only at the last minute convert to two-byte binary to be written out as the executable. And similarly you would need only read the binary and convert to strings on input to the interpreter. That sort of thing is a useful exercise, but not really relevant to being able to create machine code from assembler and thus understand exactly how things work after all the symbols have disappeared.

Your next programming assignment will be to do the binary read and write. For now you have been given both ASCII versions and binary versions of assembled and executable code. For simplicity, use the ASCII version.

In the next programming assignment, you will be asked to read both versions and to verify that you can read the two and get the same values.

Acknowledgement

The original version of this assignment came to me from Dr. Anne Marie Walsh Lancaster when I was teaching at Bowling Green State University. A more complicated version of this was a six week assignment, also in the third

course in the computer science major. The students there were writing this assignment in IBM 360 assembly language.

Errors

Your programs must catch all the usual errors, including:

- (interpreter) The program counter can't be larger than 4095.
- (interpreter) The `RD` statement must read a legal hex operand.
- (interpreter) The `RD` statement cannot read past end of file.
- (interpreter) The "instruction" pointed to by the program counter must be a valid opcode. Any three leading bits are possible, but the only legal opcodes for a leading `111` are the opcodes for `RD`, `STP`, and `WRT`.
- (interpreter) Note that trying to execute data that starts with bits other than `111` will fail (maybe) for reasons like address out of bounds, etc., but might execute and just give bogus results.
- (assembler and interpreter) Addresses must be less than 4096 decimal, so not all legal 4-digit hex values can be used as legal addresses. One cannot `ORG` below 0 or past 4096. One cannot `DS` below 0 or more than 4096.
- (assembler) An invalid symbol. Symbols are one to three characters long, start with an alpha character, include only alphanumeric characters, and can't have a blank space in the middle.
- (assembler) Opcode mnemonics must be legal machine instruction or pseudo-instruction mnemonics.
- (assembler) Symbols in columns 1-3 refer to memory locations and thus cannot be defined more than once.
- (assembler) Symbols used as symbolic operands must be defined.
- (assembler) Every program needs an `END` statement.

- (assembler) Hex operands must be 5 characters in length, with a plus or minus sign followed by four legal hex characters. Any such hex operand is legal as a 16-bit hex value, but not all hex operands are legal when used by instructions.

The Interpreter

You are to write an interpreter for the Pullet16. This comes first, because it's easier to write the interpreter than it is to write the assembler.

The assembler (described below) will output a file that is the equivalent of an `a.out` file. Your interpreter will read that machine code, decode the instruction, and interpret the execution of that instruction. Your interpreter will detect the obvious errors that might come up, such as accessing memory locations less than zero or greater than 4095, trying to execute a data word that isn't a machine instruction, and so forth.

I would strongly recommend that you keep a counter of how many instructions you have executed, and that if that counter gets too big (say bigger than 100), you terminate the interpretation. That way you won't just run on forever if you have an infinite loop (and one of the sample programs has an infinite loop).

The Assembler

Your last assignment is to write a two-pass assembler for the Pullet16 assembly language.

Your assembler should read a file from standard input and produce as output three blocks of information.

The first block is the annotated input (see examples), with errors output on lines immediately following the offending input line. This will contain the decimal line number of the input, the value of the PC in hex, the assembled code, and the original source line.

The second block is the symbol table.

The third block is the machine code as bit strings, as indicated in the examples. (You don't have to do this in binary for this part of the assignment. You can do all the machine code as strings of zeros and ones. This will make it much easier to read as you are writing and testing your code.)

This is a two-pass assembler. The purpose of the first pass is to create the *symbol table* of the labels that are used (in columns 1-3 of the input lines) and the program counter values where those labels are in the code, so that in the second pass, when a symbol is used as a symbolic operand, such as the `AAA` in

```
     STC  AAA
```

you will know the program counter value (the offset in memory from location zero in the computer) where the store is to be made.

The purpose of doing two passes is to make the second pass easier. You could do it all in one pass, but that would require you to keep track of forward references to symbols and then unwind those references when you finally detected the symbols.

It will be the case, however, that you will have to make decisions in the first pass about what the program counter should be in case of errors. For example, you will have to decide what to do with the program counter if you see a line of input with an invalid label, such as

```
999  STC  AAA
```

My suggestion would be that you go ahead and bump the program counter by one word, as if to assume that the `999` is a typo and that the `STC AAA` part of the instruction makes sense. If you did something different, then all the program counter values from there on to the end would be out of sync, because the `999` probably is a typo that can be fixed.

Similarly, if you have two instructions with the same label, then the label is multiply defined and thus illegal. But it makes more sense to flag the symbol as multiply defined and thus illegal, but still assume that the instruction is wanted and the multiple definitions are programmer carelessness. Bump the program counter as if the instruction were correct.

Among the errors you must catch in the assembly process are the following:

- Invalid symbols, either as labels (columns 1-3) or as symbolic operands (columns 11-13).
- Invalid hex operands in `DS`, `HEX`, or `ORG` statements. Note: the use of four hex digits and a plus/minus sign is slightly redundant or confusing, given that the Pullet16 uses twos complement arithmetic. You should only accept four-digit values for which the lead bit is a zero. Thus, a hex operand must be five characters. The subscript zero character

must be either a + or a − sign, the subscript 1 through 4 characters must be valid hex digits, and the subscript 1 character must have a leading zero bit and thus be a digit from 0 through 7 inclusive.

(The hex digits `FFFF` of the operand `-FFFF` are the digits for −1 in twos-complement notation, so prepending the minus sign would be a double negative and turn that into a +1. We're not going to go there. No educational value in that.)

- Multiply-defined symbols, i.e., that appear multiple times as labels.
- Undefined symbols, i.e., that appear as symbolic operands but not as labels.
- Illegal mnemonics.
- Address values larger than 4095.
- No `END` statement.

Examples

Let's look at some example programs.

Read and write

The following program does two reads, each followed by a write, and then executes a stop. Although technically the program might not need an "END" statement, in order to maintain compliance with the assembler's rules on what makes for a legal program, there is an "END" at the end.

```
    RD
    WRT
    RD
    WRT
    STP
    END
```

When this gets assembled, the executable program (the Pullet equivalent of a Microsoft dot exe file) is, in hex,

```
E001
E003
E001
E003
E002
```

and the binary equivalent would be (with blank spaces inserted for readability)

```
1110 0000 0000 0001
1110 0000 0000 0011
1110 0000 0000 0001
1110 0000 0000 0011
1110 0000 0000 0010
```

although of course there also would be no end-of-line markers in the executable file.
SIMPLIFICATION: In the real world (and for the last programming assignment), you will be required to read this as binary data from a disc. Each line of the hex or binary version as written above is 16 bits long, and there

are five such lines, so the entire executable program would be 10 bytes long. To simplify matters, however, we will use an ASCII version of the binary for your input.

That is, this program on disc for input from you reads as

```
1110000000000001
1110000000000011
1110000000000001
1110000000000011
1110000000000010
```

where we have left in the newlines to simplify that part of the assignment.

Compute squares

The second example is a program to compute and print out the squares of integers. Recall from the formula

$$(n+1)^2 = n^2 + 2 * n + 1$$

that we can compute "the next square" from the previous one without having to multiply. (This is useful for the Pullet16 since it does not have a multiply instruction.)

In the program below, we have to hard code the initialization step to ensure that we start at n and have n^2 loaded into memory. But after that, we could go forever (or at least until we exhausted the arithmetic precision of 16-bit arithmetic).

```
*23 567 9 123 56789 1
*ll mmm a sss hhhhh * comment
    LD    IDX       * load ACC with value at IDX
    LD    NN        * load initial n^2 value
    WRT             * write it
*
TOP LD    NN        * load the initial n^2 value
    ADD   N         * add in N
    ADD   N         * add in N (again)
    ADD   ONE       * add in ONE
    WRT             * write the new value
```

```
    STC   NN         * store the new n^2
    LD    N          * load the addend
    ADD   ONE        * increment it
    STC   N          * store it back
*
    LD    IDX        * reload ACC with the counter
    ADD   ONE        * increment counter
    STC   IDX        * store this new address at STO
    LD    IDX        * reload ACC with the counter
    BAN   TOP        * loop until done
    STP
*
IDX HEX        -0005 * loop index
ONE HEX        +0001 * we need a constant 1
N   HEX        +0003 *
NN  HEX        +0009 *
    END
```

Here's the assembler output of assembling this program.

Line Num	Mem Loc	Machine Code	SYM	INS	SYM	Numb	Comment
2	0	1010 0000 0001 0010		LD	IDX		* load ACC with value at IDX
3	1	1010 0000 0001 0101		LD	NN		* load initial nsquared value
4	2	1110 0000 0000 0011		WRT			* write it
5							
6	3	1010 0000 0001 0101	TOP	LD	NN		* load the initial nsquared value
7	4	1000 0000 0001 0100		ADD	N		* add in N
8	5	1000 0000 0001 0100		ADD	N		* add in N (again)
9	6	1000 0000 0001 0011		ADD	ONE		* add in ONE
10	7	1110 0000 0000 0011		WRT			* write the new value
11	8	0100 0000 0001 0101		STC	NN		* store the new nsquared
12	9	1010 0000 0001 0100		LD	N		* load the addend
13	10	1000 0000 0001 0011		ADD	ONE		* increment it
14	11	0100 0000 0001 0100		STC	N		* store it back
15							
16	12	1010 0000 0001 0010		LD	IDX		* reload ACC with the counter
17	13	1000 0000 0001 0011		ADD	ONE		* increment counter
18	14	0100 0000 0001 0010		STC	IDX		* store this new address at STO
19	15	1010 0000 0001 0010		LD	IDX		* reload ACC with the counter
20	16	0000 0000 0000 0011		BAN	TOP		* loop until done
21	17	1110 0000 0000 0010		STP	...	.....	
22							
23	18	1111 1111 1111 1011	IDX	HEX		-0005	* loop index
24	19	0000 0000 0000 0001	ONE	HEX		+0001	* we need a constant 1
25	20	0000 0000 0000 0011	N	HEX		+0003	
26	21	0000 0000 0000 1001	NN	HEX		+0009	
27	22	....		END	...	.....	

We note the symbol table (this will be needed in the final programming assignment) for the symbols used in this program, and their (decimal) locations as memory addresses.

```
SYMBOL TABLE
    SYM LOC FLAGS
SYM IDX  18
SYM N    20
SYM NN   21
SYM ONE  19
SYM TOP   3
```

And the machine code, which is the same as what is in the listing above.

```
MACHINE CODE
   0 1010 0000 0001 0010
   1 1010 0000 0001 0101
   2 1110 0000 0000 0011
   3 1010 0000 0001 0101
   4 1000 0000 0001 0100
   5 1000 0000 0001 0100
   6 1000 0000 0001 0011
   7 1110 0000 0000 0011
   8 0100 0000 0001 0101
   9 1010 0000 0001 0100
  10 1000 0000 0001 0011
  11 0100 0000 0001 0100
  12 1010 0000 0001 0010
  13 1000 0000 0001 0011
  14 0100 0000 0001 0010
  15 1010 0000 0001 0010
  16 0000 0000 0000 0011
  17 1110 0000 0000 0010
  18 1111 1111 1111 1011
  19 0000 0000 0000 0001
  20 0000 0000 0000 0011
  21 0000 0000 0000 1001
```

Homework 3

Points

This is a 50 point assignment. The first 5 points are for part A, and the remaining 45 points are for part B.

Part A

You are to read the ASCII input data into an appropriate `vector` and then dump the data using an appropriate function. You will get 5 points for doing this and for having your name and other boilerplate attribution at the top of each program file.

You will get zero points if you don't do the right read/write or if you don't have your name in each file.

Part B Assignment

You are to read the input data, store the ASCII in the `vector` for `memory_`, and then dump the vector along with a decoding of what the ASCII would mean as interpreted as machine code for the Pullet16. Note that some of the ASCII will be meaningful instructions, but that bit patterns that represent data will not be meaningful as instructions (because they aren't instructions). Doing this decoding will require you to set up a `map` of code values and mnemonics.

Homework 4

Points

This is a 60 point assignment. The first 5 points are for part A, and the remaining 55 points are for part B.

Part A

You are to read the ASCII input data into an appropriate `vector` and then dump the data using an appropriate function. You will get 5 points for doing this and for having your name and other boilerplate attribution at the top of each program file.

You will get zero points if you don't do the right read/write or if you don't have your name in each file.

Part B Assignment

You are to write an interpreter for the Pullet16. As with Homework 3, you are to read an ASCII version of the machine executable, not a binary version. And you can leave everthing as code that deals with strings of ASCII 0s and 1s rather than have to deal with binary. This will allow you to read, process, and view the ASCII version of the codes, which will make the programming task much simpler.

(In Homeworks 5 and 6, you will be asked to read binary and work with binary. Given that you will have code that handles the ASCII string equivalents of the binary, you might well consider it easier to use all that code and just do a conversion of binary to string before invoking the rest of your code and do a conversion of string to binary before writing things back out. But that's not necessary for this assignment, which can be done entirely with ASCII strings.)

You will need several classes or functions in order to make all this work smoothly. We will talk about this at length in class.

Homework 5

Points

This is a 45 point assignment. There is no Part A.

Assignment

You are to write the code that shows you can read and write binary files. Your code should

1. read the ASCII version of the machine code;
2. write a binary version of the machine code;
3. read back the binary version of the machine code;
4. expand the binary into ASCII and compare it against the original ASCII.

Homework 6

Points

This is a 60 point assignment. The first 5 points are for part A, and the remaining 55 points are for part B.

Part A

You are to read the ASCII input data into an appropriate `vector` and then dump the data using an appropriate function. You will also read the binary file and dump that in some reasonable fashion. You will get 5 points for doing this and for having your name and other boilerplate attribution at the top of each program file.

You will get zero points if you don't do the right read/write or if you don't have your name in each file.

Part B Assignment

You are to write an assembler for the Pullet16 computer.