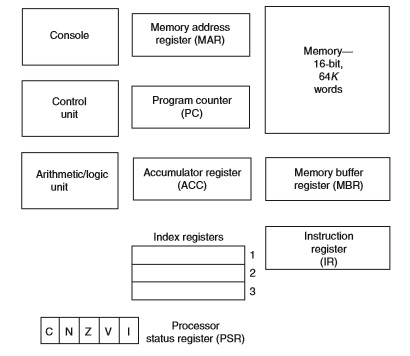
Your task is write an emulator for a simple computer (we’ll call it NARC for “Not A Real Computer) operating on 16-bit integers with the following instructions:

|  |  |  |
| --- | --- | --- |
| Mnemonic | Opcode | Description |
| HLT | 0 | Halt |
| LDA | 1 | ACC ← M[MEM] |
| STA | 2 | M[MEM] ← ACC |
| ADD | 3 | ACC ← ACC + M[MEM] |
| TCA | 4 | ACC ← !ACC + 1 (2’s Complement) |
| BRU | 5 | Branch unconditional |
| BIP | 6 | Branch if ACC > 0 |
| BIN | 7 | Branch if ACC < 0 |
| RWD | 8 | Read a word into ACC |
| WWD | 9 | Write a word from ACC |
| SHL | A | Shift left ACC once |
| SHR | B | Shift right ACC once |
| LDX | C | INDEX ← M[MEM] |
| STX | D | M[MEM] ← INDEX |
| TIX | E | Test index increment INDEX ← INDEX + 1 Branch if INDEX = 0 |
| TDX | F | Test Index Decrement INDEX ← INDEX - 1 Branch if INDEX ≠ 0 |

The figure below shows the hardware components of NARC.

****

**Instruction Format**

Each instruction in a NARC program occupies a 16-bit word. An instruction word has four fields, as shown in the figure below:



Bits 15 through 11 of the instruction word are used for the operation code (opcode). The *opcode* is a unique bit patter n that encodes a primitive operation the computer can perform. Thus, NARC can have a total of 32 instructions. We use an instruction set with only 16 instructions for simplicity for the base project. The opcodes for these 16 instructions occupy bits 15 through 12, and bit 11 is set to 0. If, for an extra credit exercise, the instruction set were to be expanded beyond the current set of 16, the opcodes for the new instructions would have a 1 in bit 11. Bit 10 of the instruction word is the *indirect* flag. This bit will be set to 1 if indirect addressing is used; otherwise it is set to 0. Bits 9 and 8 of the instruction word select one of the three index registers when indexed addressing is called for or if the instruction manipulates an index register:

|  |  |  |
| --- | --- | --- |
| Bit 9 | Bit 8 | Index Register Selected |
| 0 | 0 | None |
| 0 | 1 | 1 |
| 1 | 0 | 2 |
| 1 | 1 | 3 |

Bits 7 through 0 are used to represent the memory address in those instructions that refer to memory. If the instruction does not refer to memory, the indirect, index, and memory address fields are not used; the opcode field represents the complete instruction.

With only 8 bits in the address representation, NARC can directly address only 256 memory locations. That means the program and data must always be in the first 256 locations of the memory. Indexed and indirect addressing modes are used to extend the addressing range to 64K. Thus NARC has direct, indirect, and indexed addressing modes. When both indirect and indexed addressing mode fields are used, the addressing mode is as indexed-indirect (pre-indexed indirect.)

**Addressing Modes**

NARC addressing modes are described here with reference to the load accumulator (LDA) instruction. Here, Z is assumed to be the symbolic address of the memory location 10. For each mode , the assembly language format is shown first, followed by the instruction format encoded in binary (i.e., the machine language) . The *effective address* calculation and the effect of the instruction are also illustrated. Note that the effective address is the address of the memory word where the operand is located.

*Direct Addressing.*

Instruction format: LDA Z

|  |  |  |  |
| --- | --- | --- | --- |
| 00010 | 0 | 00 | 00001010 |

Effective address: Z ( = 0x0A )

Effect: ACC ← M[Z].

*Indexed Addressing.*

Instruction format: LDA Z, 2

|  |  |  |  |
| --- | --- | --- | --- |
| 00010 | 0 | 10 | 00001010 |

Effective address: Z ( = 0x0A ) + Index register 2

Effect: ACC ← M[Z + Index register 2].

The number in the operand field after the comma denotes the index register used. Note that the address field of the instruction refers to Z and the contents of the index register specify an offset from Z. Contents of an index register can be varied by using LDX, TIX, AND TDX instructions, thereby accessing various memory consecutive memory locations dynamically, by changing the contents of the index register. Further, since index registers are 16 bits wide, the effective address can be 16 bits long, thereby extending the memory addressing range to 64K from the range of 256 locations possible with 8 address bits. The most common use of indexed addressing mode is in referencing the elements of an array. The address field in the instruction points to the first element. Subsequent elements are referenced by incrementing the index register.

*Indirect Addressing.*

Instruction format: LDA \*Z

|  |  |  |  |
| --- | --- | --- | --- |
| 00010 | 1 | 00 | 00001010 |

Effective address: M[Z] ( = M[0x0A] )

Effect: MAR ← M[Z].

ACC ← M[MAR].  
 i.e., ACC ← M[M[Z]].

The asterisk next to the mnemonic denotes the indirect addressing mode. In this mode, the address field points to a location where the address of the operand can be found. Since a memory word is 16 bits long, the indirect addressing mode can also be used to extend the addressing range to 64K. Further, by simply changing the contents of location Z, we can refer to various memory addresses using the same instruction. This feature is useful, for example, in creating a multiple jump instruction in which contents of Z are dynamically changed to refer to the appropriate address to jump to. The most common use of indirect addressing is in referencing data elements through pointers. A pointer contains the address of the data to be accessed. The data access takes place through indirect addressing, using the pointer as the operand. When data are moved to other locations, it is sufficient to change the pointer value accordingly, in order to access the data from the new location.

*Indexed-Indirect Addressing.*

Instruction format: LDA \*Z, 2

|  |  |  |  |
| --- | --- | --- | --- |
| 00010 | 1 | 10 | 00001010 |

Effective address: M[Z + index register 2] ( = M[0x0A] )

Effect: MAR ← M[Z].

ACC ← M[MAR + index register 2].  
 i.e., ACC ← M[M[Z + index register 2]].

Indexing is done first, followed by indirection to compute the effective address whose contents are loaded into the accumulator

The above addressing modes are applicable to all single-address instructions. The only exceptions are the index-reference instructions (LDX, STX, TIX, and TDX) in which indexing is not permitted.

Addressing Limitations

As discussed earlier, the NARC instruction format restricts the direct-addressing range to the first 256 locations in the memory. Thus, if the program and data can fit into locations 0 through 255, no programming difficulties are encountered. If this is not possible, the following programming alternatives can be used:

1. The program resides in the first 256 locations, and the data resides in higher addressed locations in the memory. In this case, all instruction addresses can be represented by the 8-bit address field. Since data references require an address field longer than 8 bit, all data references are handled using indexed and indirect addressing modes. For example, the data location 300 can be loaded into the ACC by either of the following instructions:
   1. LDA 0, 2 assuming that index register 2 contains 300.
   2. LDA \*0 assuming that location 0 in the memory contains 300.
2. Data reside in the first 256 locations, and the program resides beyond location 255. In this case, all data reference instructions (such as LDA, STA, etc.) can use direct, indirect, and=or indexed modes, but all other memory reference instructions (such as BRU, BIP, BIN) must use indexed and/or indirect modes.
3. If the program and data both reside beyond location 255, all memory reference instructions must be indirect and/or indexed.

Recall that the index reference instructions can use only the direct and indirect modes of addressing.

**Documentation of NARC Instructions:** A description of instructions and their representation follows.

*Zero-Address Instructions*. In this class of instructions, the opcode represents the complete instruction. The operand (if needed) is implied to be in the ACC. The address field, the index flag and the indirect flag are not used.

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | - | -- | -------- |

A description of each instruction follows:

|  |  |  |
| --- | --- | --- |
| HLT | Stop | Halt |

The HLT instruction indicates the logical end of a program and hence stops the machine from fetching the next instruction (if any).

|  |  |
| --- | --- |
| **TCA** | ACC ← !ACC + 1 2’s complement accumulator |

TCA complements each bit of the ACC to produce the 1s complement and then a 1 is added to produce the 2s complement. The 2s complement of the ACC is stored back into the ACC.

|  |  |  |
| --- | --- | --- |
| **SHL** | ACC15-1 ← ACC14-0  ACC0 ← 0 | Shift left |

The SHL instruction shifts the contents of the ACC 1 bit to the left and fills a 0 into the least significant bit of the ACC.

|  |  |  |
| --- | --- | --- |
| **SHR** | ACC14-0 ← ACC15-1  ACC15 ← ACC15 | Shift right |

The SHR instruction shifts the contents of the ACC 1 bit to the right and the most significant bit of the ACC remains unchanged. The contents of the last significant bit position are lost.

*One-Address Instructions*. These instructions use all 16 bits of an instruction word. In the following, MEM is a symbolic address of an arbitrary memory location. An absolute address is the physical address of a memory location, expressed as a numeric quantity. A symbolic address is mapped to an absolute address when an assembly language problem is translated into machine language. The description assumes a direct addressing mode in which MEM is the effective address (the address of the operand). The 8-bit address is usually modified by the indirect and index operations to generate the effective address of a memory operand for each of these instructions.

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | X | XX | XXXXXXXX |

The description of one-address instructions follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **LDA** | **MEM** | ACC ← M[MEM]. | Load accumulator |

LDA loads the ACC with the contents of the memory location (MEM) specified. Contents of MEM are not changed, but the contents of the ACC before the execution of this instruction are replaced by the contents of MEM.

|  |  |  |  |
| --- | --- | --- | --- |
| **STA** | **MEM** | M[MEM] ← ACC. | Store accumulator |

STA stores the contents of the ACC into the specified memory location. ACC contents are not altered.

|  |  |  |  |
| --- | --- | --- | --- |
| **ADD** | **MEM** | ACC ← ACC + M[MEM]. | Add |

ADD adds the contents of the memory location specified to the contents of the ACC. Memory contents are not altered.

|  |  |  |  |
| --- | --- | --- | --- |
| **BRU** | **MEM** | PC ← MEM. | Branch unconditional |

BRU transfers the program control to the address MEM. That is, the next instruction to be executed is at MEM.

|  |  |  |  |
| --- | --- | --- | --- |
| **BIP** | **MEM** | IF ACC > 0 THEN  PC ← MEM | Branch if ACC is positive |

The BIP instruction tests the N and Z bits of PSR. If both of them are 0, then the program execution resumes at the address (MEM) specified; if not, execution continues with the next instruction in sequence. Since the PC must contain the address of the instruction to be executed next, the branching operation corresponds to transferring the address into PC.

|  |  |  |  |
| --- | --- | --- | --- |
| **BIN** | **MEM** | IF ACC <0 THEN  PC← MEM. | Branch if accumulator negative |

The BIN instruction tests the N bit of PSR; if it is 1, program execution resumes at the address specified; if not, the execution continues with the next instruction in sequence.

|  |  |  |  |
| --- | --- | --- | --- |
| **LDX** | **MEM, INDEX** | INDEX ← M[MEM]. | Load index register |

The LDX loads the index register (specified by INDEX) with the contents of memory location specified. In the assembly language instruction format, INDEX will be 1, 2, or 3.

|  |  |  |  |
| --- | --- | --- | --- |
| **STX** | **MEM, INDEX** | M[MEM] ← INDEX. | Score index register |

The STX stores a copy of the contents of the index register specified by the index flag into the memory location specified by the address. The index register contents remain unchanged.

|  |  |  |  |
| --- | --- | --- | --- |
| **TIX** | **MEM, INDEX** | INDEX ← INDEX + 1  IF INDEX = 0 THEN PC← MEM. | Test index increment |

TIX increments the index register content by 1. Next, it tests the index register content; if it is 0, the program execution resumes at the address specified; otherwise, execution continues with the next sequential instruction.

|  |  |  |  |
| --- | --- | --- | --- |
| **TDX** | **MEM, INDEX** | INDEX← INDEX - 1  IF INDEX≠0 THEN PC← MEM. | Test index decrement |

TDX decrements the index register content by 1. Next it tests the index register content; if it is not equal to 0, the program execution resumes at the address specified; otherwise, execution continues with the next sequential instruction.

It is important to note that LDX, STX, TDX, and TIX instructions ‘‘refer’’ to an index register as one of the operands.

Indexed mode of addressing is thus not possible with these instructions since the index field is used for the index register reference. Only direct and indirect modes of addressing can be used. For example:

* LDA Z, 3 adds the contents of index register 3 to Z to compute the effective address EA. Then the contents of memory location EA are loaded into the ACC. The index register is not altered, whereas,
* LDX Z, 3 loads the index register 3 from the contents of memory location Z

*Input /Output Instructions:*  Since NARC has one input and one output device, the address, index, and indirect fields in the instruction word are not used. Thus, these are also zero-address instructions.

|  |  |  |
| --- | --- | --- |
| **RWD** | ACC Input data. | Read a word |

RWD instruction reads a 16-bit word from the input device into the ACC. The content s of the ACC before RWD are thus lost.

|  |  |  |
| --- | --- | --- |
| **WWD** | Output ACC. | Write a word |

WWD instruction writes a 16-bit word from the ACC onto the output device. ACC contents remain unaltered.

Your interpreter should read a set of machine instructions from a binary file (in big-Endian format) whose name is passed as a command-line argument and load those instructions into your code memory, stopping when detecting the end-of-file. Then your program counter should be initialized to 0 and the interpreter should run until a HALT instruction is detected.

For example, if your C program is called vm and your code is named myprog.bin, then use the command line:  
> **vm myprog.bin**

SimilarLy, for a java program, the command line would be   
> **java Vm myprog.bin**

(In the above example, for C, argv[0] = “vm” and argv[1] = “myprog.bin”; for Java, args[0] = “myprog.bin”.)

We will use the von Neumann memory model, with a single 128KB (65,536 16-bit words) for instructions and data. The memory will be word-addressable, that is, a 16-bit word is the smallest addressable memory location. Note that implication of having word addressability means that the location counter will be incremented by 1 rather than by 2 at each step.

You may use Java, C, or Python 2.7 to implement your emulator, but it must interpret the appropriate binary code in Big Endian format. If you use Java, you may use any platform you wish. If you use any other language, you must either provide a makefile or complete compilation instructions for generating an executable under Linux.

Hand in your source code and compilation instructions via Blackboard.