MinVM Take Home Test

The goal of this test is to implement a minimal virtual machine. A small harness, some sample programs, expected output and this spec are included to get you started.

Machine summary

- 8 bit word size
- 256 words of memory
- 4 general registers, 1 flag register and a program counter

Test guidelines

- Your test submission should be either minvm_test.c by itself or a folder containing minvm_test.c at the root level, this file alone will be linked with the test driver.
- The test will be evaluated on readability, correctness and performance in that order
- All of the samples should run to completion and produce the expected output
- Pay attention to the details
- Keep track of how you spend your time during the test, we want to know how you solved it

Machine Specifications

Memory Model

The word size is 8 bits and 256 words of memory can be accessed directly. There are no paging modes or stack pointer manipulation. All instructions and data must fit into 256 words of memory. The memory does not have any protection modes.

Instruction Encoding

- All 16 instructions are encoded in a single word
- Some instructions have one or more arguments after the opcode
- The upper 4 bits of the instruction specify the opcode
- The lower 4 bits are the first argument for the instruction

Example Instruction Encoding

The **LOADI** instruction is short for 'Load Immediate Values'. **LOADI** is represented with ordinal 0 and the lower 4 bits specify which registers should be loaded from immediate data.

```
|  7..4  |  3..0   |  7..4   |  3..0   |  7..4   |  3..0   |
 0 0 0 0   0 1 0 1   0 0 0 0   0 0 0 1   0 0 0 0   0 0 1 0
```

In the example above the 3 words encode a **LOADI** with bits 0 and 2 set in the register mask portion indicating a load into register A and C. The following two words with values 1 and 2 will be loaded into register A and C respectively.

LOADI is special in that if the instruction is specified without a register mask it will halt the machine without exception. This means any single byte 0 in the stream will terminate execution.

Table 1: Instruction Summary

Mnemonic	Code	Arguments	Short Description
LOADI	0x0R	*b,…	Load immediate words into R in order
INC	0x1R		Increment with overflow all registers in R
DEC	0x2R		Decrement with underflow all registers in
LOADR	0x3R	$abcd	Load data at register offsets $abcd into targets in R
ADD	0x4R	$x, $y	R = $x + $y, overflow in order
SUB	0x5R	$x, $y	R = $x - $y, underflow in order
MUL	0x6R	$x, $y	R = $x mul $y, overflow in order
DIV	0x7R	$x, $y	R = $x div $y, no underflow
AND	0x8R	$x, $y	R = $x bitwise-and $y, write to all registers in R
OR	0x9R	$x, $y	R = $x bitwise-or $y, write to all registers in R
XOR	0xAR	$x, $y	R = $x bitwise-xor $y, write to all registers in R
ROTR	0xBR		Rotate register right
JMPNEQ	0xCR	*index	Jump to *index for any inequality in R
JMPEQ	0xDR	*index	Jump to *index for all equal in R
STOR	0xER	*index	Write each R to successive values at *index
ITR	0xFI		Execute interrupt I

Table 2: Instruction Summary Legend

Symbol	Description
'$'	Register argument encoded as a mask in the low bits of a single word
'*'	Immediate values
R	Register index mask, all registers in the mask are the target or source

Symbol	Description
$x, $y	Indicates 2 registers by mask, these will be encoded in the low bits of an extra word
b,...	Multiple words in the instruction stream

Flags Summary

The flags register is updated by the machine as instructions are executed, they are currently limited to failure and termination conditions.

Table 3: Machine Flags

Flag Name	Bit Pattern	Description
HALT	0x01	Indicates that the machine should stop running
EXCEPTION	0x02	Indicates that the machine encountered an invalid instruction

Register Argument Encoding

Each instruction is encoded in the upper 4 bits of the instruction byte while the lower 4 bits are an optional argument.

The register mask in the instruction will normally be the target registers of the instruction. Additional bytes may have source masks, index values or immediate data to be operated on.

Some instructions place limits on the number of operand registers. For example ADD, SUB, MUL, DIV, AND, OR, and XOR all require 2 operand registers.

Multiple registers may be the target of an instruction. How the multiple registers are treated depends on the instruction. Many operations overflow or underflow in order.

The register masks are defined as:

Table 4: Register Masks

Register	Mask
A	0x01
B	0x02
C	0x04
D	0x08

Example Instruction Sequences:

```
1       0x03, 0x00, 0x01    ;; a = 0, b = 1
```

```
    0xDF, 0x10          ;; if(a == b && b == c && c == d) goto offset 16;

    0x01, 0x00          ;; a = 0x00
    0x32, 0x01          ;; b = memory[a]

    0xEF, 0x04          ;; memset(&memory[4], byte[]{ $a, $b, $c, $d }, 4)

    0x03, 0xff, 0x00    ;; a = 0xff, b = 0x00
    0x13                ;; a++ (overflows to b: a is 0x00, b is 0x01)

    0x03, 0x00, 0x02    ;; a = 0x00, b = 0x02
    0x23                ;; a-- (underflows from b, a is 0xff b is 0x01)

    0x6C, 0x03          ;; cd = a * b (with low bits in c and high bits in d)

    0x00                ;; halt (flags = 0x01)

    0x41                :: exception (flags = 0x3)

    0x10                ;; noop
```

Initialization

The following is the initialization sequence of the machine:

1. All 256 words of memory are initialized to zero
2. Memory is loaded with the binary contents of the program in the .bin file
3. The program counter, all registers and flags are set to 0
4. Control is passed to the vm_exec() function which will begin the read, decode and execute cycle

Execution starts at the first word in memory, with all 256 words of memory addressable.

The function **vm_exec()** is called with a **virtual_machine_t** structure containing the initial machine state after the above state. When **vm_exec()** returns the machine should be halted with all valid instructions decoded.

Shutdown

Stopping the machine is indicated by setting the **HALT** flag. If an invalid condition is encountered the **EXCEPTION** flag should also be set.

Program Counter

Instructions are processed in order using the fetch-decode-execute cycle. The program counter must be moved past the instruction and any operands before executing. The program counter should always point at the instruction that should be executed next. In cases where the machine must halt, the program counter should point to the place in memory immediately after the instruction and operands that halted the machine.

When moving past the last word in the instruction stream the program counter will wrap around to 0 and continue to decode the instruction and operands. Wrapping around at the memory boundary must not halt the machine.

Overflow and Underflow

Many instructions specify specific overflow and underflow conditions.

Overflow will progress from one register to the next in order. Overflow will reset the value of the target register with the overflow value. For instance **INC** a,**b,**$d * *implementsa24bitwidecountervalueignoring * *$**c**.

Note: While **$d** 0xff + 1 == 0x00 there is no overflow target to place bits.

```
Starting register values:
A: 0xff
B: 0xfe
C: 0x00
D: 0xfe

Instruction and resulting register values:
INC $b, $d      -> A: 0xff, B: 0xff, C: 0x00, D: 0xfe
INC $b, $d      -> A: 0xff, B: 0x00, C: 0x00, D: 0xff
INC $b, $d      -> A: 0xff, B: 0x01, C: 0x00, D: 0xff
INC $a, $b, $d  -> A: 0x00, B: 0x02, C: 0x00, D: 0xff
INC $d          -> A: 0x00, B: 0x02, C: 0x00, D: 0x00
```

When under flowing each register will take bits from the previous register in the underflow chain. Register **$a** does not underflow.

```
A: 0x01
B: 0x00
C: 0x00
D: 0xff

DEC $a, $b, $d -> A: 0x00, B: 0x00, C: 0x00, D: 0xff
DEC $a, $b, $d -> A: 0xff, B: 0xff, C: 0x00, D: 0xfe
DEC $c         -> A: 0xff, B: 0xff, C: 0xff, D: 0xfe
```

Sample Output

The samples have the following output state:

Table 5: Sample Program Ouput

Sample Name	Output	PC	Flags	A	B	C	D
countbits	5	0x1A	0x01	0x0A	0x7F	0x30	0x05
loop		0x0C	0x01	0xFF	0x02	0xFF	0x01
blizzard	Blizzard!	0x1E	0x01	0x00	0x2A	0x00	0x0C
mathops		0x53	0x01	0x00	0x02	0x00	0x00
unique_num		0x19	0x01	0x06	0x00	0x00	0x0A
eight_queens	92 positions	0x80	0x01	0x00	0x00	0x07	0x00

The samples exercise every instruction but not all combinations of instruction argument. There are additional programs not included in the samples that may need to run using your virtual machine.

Instruction Details

LOADI 0x0R

Load immediate values into registers specified in R

- Loads the words following the instruction into the registers specified in the mask.
- If 0 registers are specified in the mask the HALT flag should be set but the EXCEPTION flag should **not** be set. This triggers a clean termination of the machine.
- While the LOADI instruction itself is only one byte but it can load in from 0 to 4 bytes
- Load order is A, B, C, D
- Loaded bytes will wrap around at the memory boundary

In the following example the LOADI is located in memory location 254. The successive immediate values are loaded into registers A, B, C and D.

```
254: 0x0f
255: 0x01
000: 0x02
001: 0x03
002: 0x04

State:
PC: 0x03, A: 0x01, B: 0x02, C: 0x03, C: 0x04
```

INC 0x1R

Increment with overflow the registers specified in R

- All specified registers are used as overflow targets in order
- Overflow order is A -> B -> C -> D

DEC 0x2R

Decrement with underflow all registers specified in R

- All specified registers are used as underflow targets in order
- Underflow order is D -> C -> B -> A

LOADR 0x3R 0x0V

Load into registers R the data at offsets V

- Registers specified in V are used as offsets in the code to load from.
- The count of registers in V and R must match
- Mismatch of register counts will trigger an exception
- Load order is A, B, C, D
- Reading from V should wrap around at memory boundary

For example if register B is loaded with '0x00' executing '0x31, 0x02' will load code[0] into register A.

ADD 0x4R 0x0V

Add the two operand registers V and store in target registers R

- Only two operands are allowed in V
- Any argument count in V other than two will trigger an exception
- The result is stored with overflow in the registered specified in R
- Overflow order is A -> B -> C -> D
- The result is stored in the registered specified in R using all available bits
- If insufficient space is available to store the result the upper bits are truncated

Example:

```
A: 0xff
B: 0xff
C: 0x03
D: 0x04

0x4F, 0x03  ; $abcd = $a + $b -> A: 0xfe, B: 0x01, C: 0x00, D: 0x00
```

SUB 0x5R 0x0V

Subtract two operand registers V and store in target registers R

- Underflow order of the registers in V is D -> C -> B -> A
- Only two operands are allowed in V.
- The operands must be processed in register order (a - b, not b - a, a - d)
- Any argument count in V other than two will trigger an exception
- Underflow will only consider in order the registers in R

- The result is stored in the registered specified in R using all available bits
- If insufficient space is available to store the result the upper bits are truncated

MUL 0x6R 0x0V

Multiply the two operand registers V and store in target registers R

- Overflow order is A -> B -> C -> D
- Only two operands are allowed in V.
- Any argument count in V other than two will trigger an exception
- Overflow will only consider in order the registers in R
- The result is stored in the registered specified in R using all available bits
- If insufficient space is available to store the result the upper bits are truncated

DIV 0x7R 0x0V

Divide the two operand registers V and store in target registers R

- Only two operands are allowed in V
- Any argument count in V other than two will trigger an exception
- Dividing by zero will cause an exception
- The result is stored in the registered specified in R using all available bits
- If insufficient space is available to store the result the upper bits are truncated

AND 0x8R 0x0V

Bitwise and the two operand registers V and store in target registers R

- Any argument count in V other than two will trigger an exception
- All targets in R will receive the calculated value

OR 0x9R 0x0V

Bitwise or the two operand registers V and store in target registers R

- Any argument count in V other than two will trigger an exception
- All targets in R will receive the calculated value

XOR 0xAR 0x0V

Bitwise xor the two operand registers V and store in target registers R

- Any argument count in V other than two will trigger an exception
- All targets in R will receive the calculated value

ROTR 0xBR

Rotate registers right

- Rotated registers are selected by the register mask
- Each register is rotated to the next register in order
- Rotation order is A -> B -> C -> D -> A
- Rotating one, or no registers has no effect
- Rotating any two registers has the effect of swapping them

Example:

```
Before A: 1, B: 2, C: 3

ROTR $a, $b, $c

After A: 3, B: 1, C: 2
```

JMPNEQ 0xCR 0xII

Jump to the offset II if any registers specified in R are not equal

- If 0 registers are specified, JMPNEQ is an unconditional jump
- If 1 register is specified JMPNEQ jumps if register is not equal to 0
- Jumping will alter the program counter of the virtual machine
- Any register inequality specifed in the mask R will trigger a jump

JMPEQ 0xDR 0xII

Jump to the offset II only if all registers R are equal

- If 0 registers are specified JMPEQ is an unconditional jump
- If 1 register is specified JMPEQ jumps only if the register is equal to 0
- Jumping will alter the program counter of the virtual machine
- All registers specified in the mask R must match to trigger a jump

STOR 0xER 0xII

Store registers in R to the offset II

- Each register must be written to program data at the specified offset
- When the write extends 256 bytes the write will wrap around available memory
- If 0 registers are specified the instruction is a no-op

ITR 0xFI

Invoke the interrupt function I

- Call the function at index I in the VM interrupt table (see struct VirtualMachine)

Interrupt Functions

The instruction ITR causes the machine to pass control to an interrupt function specified by index. In the virtual machine case the functions are provided in a table with a specific signature and should be called to pass control.

Halting

Halting the VM is done by setting the MINVM_HALT flag on the control structure in vm->flags and returning from 'vm_exec'. The program counter should point to program data after the instruction and operands that caused the halt.

Code, Compiling and Execution

Your virtual machine will be tested by a validation driver, running under linux with gcc >= 4.4 a simple version is provided. You must implement an external function (defined below):

```
extern "C" void vm_exec(struct virtual_machine_t *state);
```

Your code must be valid ANSI C99 code. You can complete the test using cl.exe, clang or gcc. Aim for writing correct and simple code, enable all warnings and avoid platform specific functionality.

Interrupt routines

The interrupts routines defined in 'itr_table' available in the driver are:

```
0 itr_dump_state          - Dumps the state of the vm to stdout
1 itr_print_a             - Prints register A as a character
```

Coding Style

The coding style of the test and included libraries is essentially K&R while using C99 types from stdint.h/stdbool.h. Additionally definition and declaration function parameters are separated by a space to allow grepping invocations separately. Indents should be 4 spaces, line endings should be consistent with the file being edited.

```
void foo (int32_t value, mything_t *thing) {
    if (thing) {
      thing->value = value;
    }
}
```

Optional: Open Ended Questions

- How might you improve the minvm programs for size?
- How might you improve the minvm programs for speed?
- Are any of the instructions redudant? How would you change this spec?
- What are some useful 'interrupt' routines that would be useful?
- Within the framework of the VM what would be some useful feature enhancements?

FAQ

Can I print debug information?

With a default compile the VM itself should **not** emit extra data to stdout. This is to support automatic verification of your VM. Use of pre-processor conditionals is the recommended way to include debug code.

Should I include my work?

Yes, if you do scaffolding work you should include it with your submission. It is not recommended that your extra work be included in your main test source file if it clutters the implementation. Include additional material in a seperate file or folder.

What should happen if the PC goes outside of program data?

The program counter is an unsigned 8 bit register and should wrap around. The VM should not halt when the program counter wraps around. A value of 0 in the instruction stream is used to cleanly terminate the machine.

What should happen if a store or load goes outside of program data?

The store should wrap around. The VM should not halt in this case.

Are INC and DEC destructive?

Yes, the instructions INC, DEC operate on all registers in the instruction register mask in a well defined order from A to D. The values in the registers will be mutated after execution of these instruction completes. It's possible to use a register as a source and destination for these type of instructions.

How do you handle conflicts with the other math operations?

The instructions ADD, SUB, MUL, DIV, AND, OR, and XOR require exactly 2 operands but may have up to 4 destinations. The source operands must all be read first before the operation executes. After calculating the result the machine will write the value into all of the target registers as if they are one storage location.

How do overflow and underflow work?

A section has been added to describe overflow and underflow in more detail above. The instructions INC, DEC, ADD, SUB, and MUL use the destination registers as overflow and underflow targets in order.

The order will always be A -> B -> C -> D but registers may be omitted from this overflow order using the mask. If insufficient registers are provided the operation will lose those bits.

SUB and DEC must borrow from registers in order D -> C -> B -> A. If insufficient registers are provided for underflow the operation will truncate those bits.

```
0x00 - 0x01    = 0xff
0xff - 0x01    = 0xfe
0x00 - 0xff    = 0x01
```

Good Luck!