Vertical Microarchitecture Assembly Language: VMAL

This document lays out the specification for VMAL, an assembly language for the Vertical Microarchitecture (VM). This document will make constant references to the Vertical Microarchitecture covered in the lecture, and it is advised that you have [this handout](https://drive.google.com/file/d/1H6EdfxeILtwh2hPibSd2Bms5jTJKqnyP/view) with you as you read through this document.

**Microarchitecture Details**

Though the layout of the VM is specified in detail on the handout, there are some aspects of the VM which are not explicitly given, such as the bit width of data busses, or the total amount of accessible memory. For the purposes of VMAL, the microarchitecture is a 32-bit CPU (32-bit words) with a 32-bit word addressed memory. Though this is generally much more than one would expect for a theoretical CPU, this makes understanding and using the VM easier in most cases. Another behavior that is not made very explicit is the behavior of setting the N (negative) and Z (zero) flags on the CPU. Below, Opcode 5 is the only operation capable of doing this. As a result, the behavior of the N and Z flags will be that they can be set by Opcode 5, but will stay whatever they were last set to otherwise. This way, it may be possible to set the N and Z flags with respect to a value in a register, then modify that register, and finally perform a conditional branch based on the N and Z flags for the original value in the register, even though it is no longer the same.

For the main memory, reading and writing are done with the following processes. The VM has two dedicated registers, the MAR and the MBR. The MAR (which stands for Memory Address Register) is used to transmit an address to the main memory. This register is 32-bits (as all other registers), and represents the address of the word in memory (as the memory can only be addressed by word). The MBR (which stands for Memory Buffer Register) is used to hold either data to be written to the main memory, or data that was just read from the main memory. In order to read data from the main memory, the value of the MAR should be set to the address in memory of the data you want to read, and then the RD; operation should be performed. With the current operations available, a read must take 2 cycles, one for Opcode 0, and one for Opcode 2, and the resulting data read from the memory is stored in the MBR. In order to write data to the main memory, the value of the MAR should be set to the address in memory you want to write to, and the value of the MBR should be set to whatever data you want to write to that location. Finally, the WR; operation should be performed in order to actually write the data. Though you might think that this operation would take 3 cycles (Opcodes 0, 4, 3), Opcode 15 is a dedicated “Store Word” operation that should in general always be used to write instead of the 3 separate operations, as it is faster to do so.

**VMAL Registers**

Though the [Vertical Microarchitecture Handout](https://drive.google.com/file/d/1H6EdfxeILtwh2hPibSd2Bms5jTJKqnyP/view) (the registers are on the first page, structure of the Vertical Microarchitecture is on the third page) should be the primary reference for what the registers are, here are a few comments regarding specific behavior of some of the registers. The PC, 0, +1, and -1 registers are set to specific values **after** the register initialization phase, so setting these registers to specific values won’t do anything. You can however overwrite these registers with instructions, though it is recommended that you not do this, especially with the PC. The PC register is actually used to determine which line of code is currently being executed, and what the next line of code to be run is as well. If you modify this register, the program will cease to run normally, and may just terminate.

**VMAL Specification**

In order to keep VMAL as simple and straightforward as possible, very little has been added on top of the original vertical microarchitecture machine code. In fact, the only addition is essentially a labelling system, so that all branching operations can be done to labels as opposed to exact line numbers.

Below is the list of all valid instructions for the main code in VMAL (an example is given at the end)

| Opcode | Usage | Instruction | Description |
| --- | --- | --- | --- |
| 0 | SA X; | MAR := X; | SA stands for “Set Address”, and it sets the MAR register to the value held in the register X. X should be the hexadecimal representation of the register number. |
| 1 | RB X; | X := MBR; | RB stands for “Read Buffer”, and it sets the register X to the value held in the MBR |
| 2 | RD; | RD; | Performs a read from memory. This is done by taking the value in MAR as the memory address being read, and putting the value read into the MBR. |
| 3 | WR; | WR; | Performs a write to memory. This is done by taking the value in the MAR as the memory address, and the value in the MBR as the data to be written at that address. |
| 4 | SB X; | MBR := X; | SB stands for “Set Buffer”, and it sets the MBR register to the value held in the register X |
| 5 | SF X; | ALU := X;  NZ; | SF stands for “Set flags”. Reads the register X and sets the N and Z flags according to whether the value in X is negative and zero respectively. |
|  | LBL L; |  | Used to define a label for the GO operation below. L can be any string that starts with any letter or “\_” (underscore), with no whitespace (ex. LBL \_\_Here16;). |
| 6 | GO L; | GOTO L; | Jumps to the line following the declaration of the label L (L must be defined in your code for this to work, otherwise nothing will happen). |
| 7 | BIN L; | IF N GOTO L; | Jumps to the line following the declaration of the label L IF the N flag is 1. Otherwise, the program continues to the next line |
| 8 | BIZ L; | IF Z GOTO L; | Jumps to the line following the declaration of the label L IF the Z flag is 1. Otherwise, the program continues to the next line |
| 9 | ADD X, Y; | X := X + Y; | Adds the values of registers X and Y, and stores the result to register X |
| 10 | AND X, Y; | X := X & Y; | Performs bitwise AND on the bits of the values or the registers X and Y, and stores the result to register X |
| 11 | MV X, Y; | X := Y; | Sets the value of the register X to the value of register Y |
| 12 | NOT X, Y; | X := NOT(Y); | Inverts the bits of the value of register Y, and stores the result to register X |
| 13 | RS X, Y; | X := RSHIFT(Y); | Performs a right shift (by a single bit) of the value in register Y, and stores the result into register X |
| 14 | LS X, Y; | X := LSHIFT(Y); | Performs a left shift (by a single bit) of the value in register Y, and stores the result into register X |
| 15 | SW X, Y; | MAR := X;  MBR := Y;  WR; | Takes the value stored in register Y, and stores the result at the memory location given by the value in register X. This operation is preferable to the manual set of operations (Opcodes 0, 4, then 3), as this occurs in a single CPU cycle, as opposed to 3 |

In addition to these operations, you have the ability to create comments. Comments in VMAL are identical to comments in Python, in that they start with a #, and can begin at any point in a line (again, an example is given below). Also, since none of the operations allow you to operate with immediates (like setting register A to 5, or adding 13 to register B), VMAL adds special syntax for setting the initial values of all of the registers. The syntax for doing so is “X: number;” where X is the hexadecimal representation for the register number, and “number” is either a decimal number, hexadecimal number (prefixed with 0x), or a binary number (prefixed with 0b). Note that this register assignment can be anywhere in your code, however **these are only initializers, and will only run once at the very start of the code**. For instance, if you have 3 normal instructions and then one of these register initializers, the register initialization will still occur before any of the three operations. Again, they are only meant as a way to set the initial state of the registers.

Again, note that you cannot add constants to register values, only registers to registers. ADD A, 6; Will **not** add 6 to the value in register A. Instead, since register 6 is the +1 register, it will add 1 to register A.

Additionally, VMAL allows you to initialize the main memory with specific data. The syntax for doing so is “[M]: number;” where M is a memory location (can be given as decimal, hexadecimal, or binary), and number is a decimal number, hexadecimal number (prefixed with 0x) or a binary number (prefixed with 0b).

**Running VMAL code with VMAL**

With the given distribution of VMAL, you should see a file called VMAL.bat (which may just show up as VMAL, with the type Windows Batch file). There are two options for running VMAL:

1. You can just double click on VMAL.bat, and a file browser should open up for you to select your code file (by default, it will only show files with the extension .vmal). VMAL will then execute that code.
   1. Once you run VMAL once and select a .vmal file to run, the file browser prompt will open back up to that file location for easy selection when you are running certain code over and over (to debug or test it)
2. If you are familiar with the Windows command prompt, you can open it and navigate to where VMAL.bat is stored, and run “VMAL.bat path\_to\_file” where path\_to\_file is replaced with the path to the VMAL file you want to run. VMAL will then execute that code directly, skipping the file prompt.
   1. You should only use this if you are already familiar with the command prompt, as you should use the first option if you are not

When VMAL executes any code, the first prompt you will receive is if you want to run the program in debug mode. More about debug mode is in the next section. If you just press enter at the prompt, the code will be run until the program counter is set to a value that is out of bounds for the number of operations that your program is. If this never happens, then the program will hang forever (as long as your program is written to never infinitely loop, this should never happen). If your program does eventually terminate however, the final register state will be printed once the code is complete. The values in each register will be printed out as a **signed** 2’s complement binary number.

The file “sample.vmal” should have been provided for you in the folder with VMAL.bat to try running some VMAL code.

**Debugging VMAL code with VMAL**

Following the above instructions, if you instead type “d” or “debug” (or really anything that starts with a d) at the debugging prompt, you will instead enter debugging mode. The first thing that is printed out is the assembled code (note that this is not much different from the code that you chose to execute, but that the labels are removed). For GO and other branch operations, the line number (also listed) of where the branch goes to is listed instead of a label.

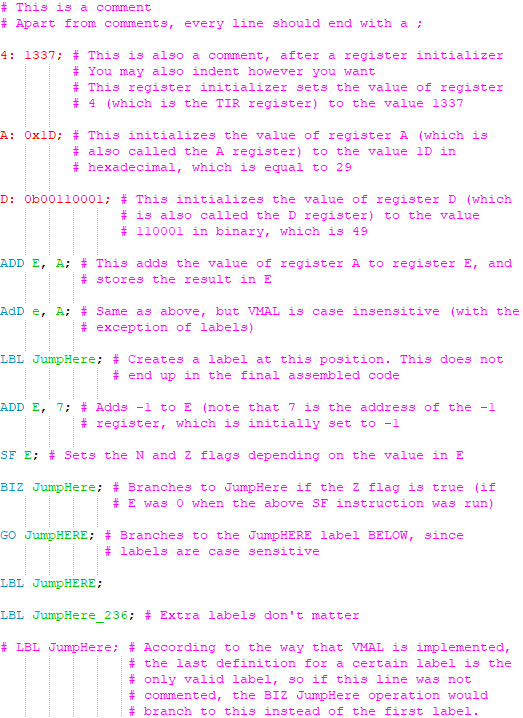
The next thing that is printed is the initial register state, the state of the N and Z flags, and the next operation to be run. Additionally, you are given a prompt. At this prompt, you have

5 options, listed below:

| Option | Action |
| --- | --- |
| n | **n**ext: This will just continue running the code, likely going to the next line. This is also the default behavior if you don’t type anything and just press enter. |
| b | **b**reakpoint: This will toggle a breakpoint at the current operation, without moving to the next operation. If you run an operation that currently has a breakpoint on it, the word BREAKPOINT will show in the terminal. |
| c | **c**ontinue: This will continue to run the code until the next breakpoint is hit. If there are no breakpoints set, then the code will run until it terminates. If the debugger is currently in continue mode you will see “Continue till Breakpoint”. Using the c option again will disable the continue till breakpoint, and resume normal debugging. |
| r | **r**un: This will stop debugger mode, and run the code from the current state until termination. |
| q | **q**uit: This will immediately terminate the code, and print the final register state as usual. It will not even run the next command that was printed. |

The debugger should be used if there is unexpected behavior in your code, especially if it seems to run forever.

**Example VMAL Code:**

****

**Example VMAL Code 2:** [**https://github.com/JCRaymond/vmal/blob/master/vmal\_ex.vmal**](https://github.com/JCRaymond/vmal/blob/master/vmal_ex.vmal)

**Notepad++ Highlighting**

As seen above in the example, there is a syntax highlighting definition provided to you so that you can program VMAL in Notepad++. In order to use the highlighting definition, follow these directions:

1. Open Notepad++
2. Navigate to Language > Define your language…
3. Click on Import…
4. Navigate to the VMAL folder, and select VMAL\_highlighting.xml
5. Click on the x in the top right corner
6. Close and reopen Notepad++

Now, you should be able to choose VMAL under the Language menu. Additionally, the language settings are set to automatically apply to files with the extension .vmal, so be sure to name your code files to something.vmal!