Virtual eXecuter – The VX platform

A virtual machine for virtually any microcontroller

Table of contents

Preface

The VX virtual machine is designed for small embedded systems originally intended for 8 bit microcontrollers.
When building the virtual machine (VM) core a user provided configuration file is used to configure the system. This configuration file specifies which type of program memory is available, how (if at all) errors must be presented and handled and all the specifics of the hardware.

The preliminary focus is to define and implement a working VM. Execution speed and core size is not the focus of the first version and so it's not expected to outperform any existing controller.

The instruction set is designed to allow for a short leaning period and will therefore look fairly similar to that of a standard 8 bit microcontroller. Two exceptions from this are that the instructions are stack based and that threading is supported directly on the lowest level (threading is not supported yet!).

Inspiration for the VX has been taken from the AVR, the PIC, the 8051, the x86, and the ARM hardcore microcontrollers, and the MicroBlaze and PicoBlaze softcore microcontrollers. The Java VM has also contributed a great deal to the development.

Revision track

0.01	Jan 30. 2008	Document started. Created from the first bits and pieces of scribbled down notes and ideas. This is written before any code has been implemented and so represents preliminary thoughts and I-think-this-is-how-it-should-be.
0.02	Feb 12. 2008	First published version of this document. Instruction set seems to be complete... Threading has been left out for now.
0.03	Feb 28. 2008	Decided to change the way the VM works. It's now a little more complex than the average 8 bit microcontroller in the way functions works. The tool chain has been expanded quite a bit and a compiler is now under development (thanks go to Arild Boes). The new assembler (still under development) produces linkable .part files instead of directly generating executables (.vxx files). This is really the way it should have been made from the start.
0.04	April 5. 2008	The status flags have been removed and instructions using these flags have either been modified or removed. This was done to make the VM more consistently stack based and to raise the abstraction level of the instruction set. This should also make it a lot simply to program the VM.

Programs

VX applications stored on offline storage are called programs. Programs consist of two sections

- Meta
- Code

The meta section holds information on the program that enables the program loader verify the file as a valid VX executable and to load the program into memory.

The code section holds the program code bytes.

Processes

A program that has been loaded into memory it is called a process.
When the loader runs it copies the code section of the file into memory and creates the stack and the global data sections.

Functions

Program execution

When a program is loaded three base pointers are initialized. Each pointer points to the start of each section of the program. At the same time four size variables are loaded with the size of each section. This way sections can be placed freely in memory by the loaded as the base pointers are added to all memory location values in the program. The size variables are used to detect out-of-range access attempts. If out-of-range access is attempted the program is terminated prior to access.

Data types

The VX instruction set operates on the following data types

- Single. A single byte representing an integer number.
- Double. Two bytes representing an integer number.
- Quad. Four bytes representing an integer number.
- Float. Four bytes representing a floating point number in IEEE 754 format.

Instruction set

Each instruction is defined by a unique number in the range 0 through 255. I.e. each instruction op code takes up exactly one byte in the program storage media (not including the constants required by some instructions).
Not all op codes are currently in use. Executing an unused op code terminates the program.

Values passed to the instruction directly from program memory are called constants. Far from all instructions require constants as most use arguments pushed from the stack.
In assembly language constants are placed right after the instruction mnemonic separated by a space.
Constants are placed on the memory location(s) immediately following the op code – little endian style where appropriate.

As the VX VM is stack based all arguments for the instructions must be pushed onto the stack prior to executing the instruction.
Each instruction requires a specific number of values being pushed onto the stack. This number is referred to as the instructions argument count.
The argument with the lowest number is the first one to be pushed onto the stack.
When the instruction has completed all arguments will have been popped from the stack and return values, if any, will have been pushed in place. As before, the first value pushed will be number one.

How to read the instruction set

The instructions are grouped together depending on the nature of the action they perform.

The description of each instruction has the following fields

- **Name**. The name of the instruction or instruction group if several variants of the instruction exist.
- **Description**. A description of the instruction and how it works.
- **Variants**. The variants of the instruction. The number in brackets indicates the variants op code.
- **Argument count**. The number of arguments popped from the stack by the instruction and the number pushed by the instruction.
- **Size**. The number of bytes used to represent the instruction and constants (if any). If all variants have the same size a single number indicates this otherwise the size of each instruction is written.
- **Usage.** A demonstration of how to use the instruction. For instructions with several options only one variant is demonstrated.

The assembler is case insensitive and the mnemonics are only written with mixed case letters to increase readability (label names ARE however case sensitive).

Arithmetic

Add

Add the first argument to the second.

Variants AddS [], AddD [], AddQ [], AddF []

Arg. count 2 / 1

Size 1

Usage
```
PushLD 102   ; arg 1 (the double at memory location 102)
PushCD 20000 ; arg 2 (a constant)
AddD         ; perform an addition
PopLD 102    ; place the result back at memory location 102
```

Subtract

Subtract the second argument from the first.

Variants SubS [], SubD [], SubQ [], SubF []

Arg. count 2 / 1

Size 1

Usage
```
PushGQ 100 ; arg 1 (the quad at memory location 100)
PushLQ 104 ; arg 2 (the quad at memory location 104)
SubQ       ; sub 'em
PopLQ 108  ; put result at location 108
```

Multiply

Multiply the first argument with the second.

When multiplying integer types the output will always be the double width of the input values.

Variants MulS [], MulD [], MulQ [], MulF []

Arg. count 2 / 1

Size 1

Usage
```
PushLQ 100 ; arg 1 (the quad at memory location 100)
PushLQ 104 ; arg 2 (the quad at memory location 104)
MulQ       ; Multiply them
PopLQ 108  ; put result at location 108
PopQ 112  ; ...and 112
```

Divide

Divide the first argument by the second.

When dividing integer types the output will always be the double width of the input values (integer part pushed first followed by the fractional part).

Variants DivS [],DivD [],DivQ [], DivF []

Arg. count 2 / 1

Size 1

Usage

```
PushGQ 100 ; arg 1 (the quad at memory location 100)
PushGQ 104 ; arg 2 (the quad at memory location 104)
DivQ       ; Div 'em
PopLQ 108  ; put result at location 108
PopLQ 112  ; ...and 112
```

Increment

Increment the value by one.

Variants IncS [], IncD [], IncQ [], IncF []

Arg. count 1 / 1

Size 1

Usage

```
PushLS 100 ; push the value (the single at memory location 100)
IncS       ; add one
PopLS 100  ; and pop it back
```

Decrement

Decrement the value by one.

Variants DecS [], DecD [], DecQ [], DecF []

Arg. count 1 / 1

Size 1

Usage

```
PushLQ 100 ; push the value (the quad at memory location 100)
DecQ       ; subtract one
PopLQ 100  ; and pop it back
```

Logical

And

Perform a bitwise and of the two values.

Variants AndS [], AndD [], AndQ []

Arg. count 2 / 1

Size 1

Usage

```
PushLS 100   ; arg 1 (the single at memory location 100)
PushCS 0x80  ; arg 2 (the constant 0x80)
AndS         ; do the and
PopLS 100    ; and back with it
```

Or

Perform a bitwise or of the two values.

Variants OrS [], OrD [], OrQ []

Arg. count 2 / 1

Size 1

Usage
```
PushLS 100   ; arg 1 (the single at memory location 100)
PushCS 0x80  ; arg 2 (the constant 0x80)
OrS          ; or 'em
PopLS 100    ; and back with it
```

XOr

Perform a bitwise exclusive or of the two values.

Variants XOrS [], XOrD [], XOrQ []

Arg. count 2 / 1

Size 1

Usage
```
PushLS 100   ; arg 1 (the single at memory location 100)
PushCS 0x80  ; arg 2 (the constant 0x80)
XOrS         ; do the xor thing
PopLS 100    ; and back with it
```

Complement

Perform one's complement of the value (i.e. flips all bits).

Variants ComS [], ComD [], ComQ []

Arg. count 1 / 1

Size 1

Usage
```
PushLD 100   ; arg 1 (the double at memory location 100)
ComD         ; complement it
PopLD 100    ; and put it back
```

Negate

Negate the value. Effectively subtracts the value from zero.

Variants NegS [], NegD [], NegQ []

Arg. count 1 / 1

Size 1

Usage
```
PushGD 100   ; arg 1 (the double at memory location 100)
NegD         ; negate the value
PopGD 100    ; and put it back
```

Shift

Shift the value one bit location left or right. A low bit is shifted in.

Variants ShfLS [], ShfLD [], ShfLQ [], ShfRS [], ShfRD [], ShfRQ []

Arg. count 1 / 1

Size 1

Usage
```
PushLD 100   ; arg 1 (the double at memory location 100)
ShfLD        ; shift it left
PopLD 100    ; and put it back
```

Transfer

Load

Push a constant on the stack.

Variants LoadS [], LoadD [], LoadQ [], LoadF []

Arg. count 0 / 1

Size 2 / 3 / 5 / 5

Usage
```
LoadD 12345 ; put the double 12345 on the stack
```

Push from local

Push the contents of the local variable onto the stack.

Variants PushLS [], PushLD [], PushLQ [], PushLF []

Arg. count 0 / 1

Size 3

Usage
```
PushLD 10   ; arg 1 (the double at local memory location 10)
ShfLD       ; shift it left
PopLD 10    ; and put it back
```

Push from global

Push the contents of the global variable onto the stack.

Variants PushS [], PushD [], PushQ [], PushF []

Arg. count 0 / 1

Size 5

Usage
```
PushD 100   ; arg 1 (the double at global memory location 100)
ShfLD       ; shift it left
PopD 100    ; and put it back
```

Pop to local

Pop the top most stack value to the local variable specified.

Variants PopLS [], PopLD [], PopLQ [], PopLF []

Arg. count 1 / 0

Size 3

Usage
```
PushLD 100   ; arg 1 (the double at local memory location 100)
ShfLD        ; shift it left
PopLD 100    ; and put it back
```

Pop to global

Pop the top most value from the stack in to memory at the specified global memory location.

Variants PopS [], PopD [], PopQ [], PopF []

Arg. count 1 / 0

Size 5

Usage
```
PushD 100    ; arg 1 (the double at global memory location 100)
ShfLD        ; shift it left
PopD 100     ; and put it back
```

Branches

Jump

Unconditional jump to the specified address.

Variants Jmp []

Arg. count 0 / 0

Size 5

Usage
```
Jmp 100       ; jump to address 100
```

Jump if zero

Jump to the specified address if the last value pushed on to the stack was zero.

Variants JmpZ []

Arg. count 0 / 0

Size 5

Usage
```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
```

```
PopLD 100
JmpZ 230    ; jump to address 230 if the value was zero
```

Jump if not zero

Jump to the specified address if the last value pushed on to the stack was different from zero.

Variants JmpNZ []

Arg. count 0 / 0

Size 5

Usage
```
PushGD 100  ; arg 1 (the double at memory location 100)
Dec
PopGD 100
JmpNZ 325   ; jump to address 325 if the value was not zero
```

Jump if negative

Jump to the specified address if the last value pushed on to the stack was negative (if viewed as a number in two's complement format).

Variants JmpN []

Arg. count 0 / 0

Size 5

Usage
```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpN 232    ; jump to address 232 if the value was negative
```

Jump if not negative

Jump to the specified address if the last value pushed on to the stack was not negative (if viewed as a number in two's complement format).

Variants JmpNN []

Arg. count 0 / 0

Size 5

Usage
```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpNN 5642  ; jump to address 5642 if the value was not negative
```

Jump if positive

Jump to the specified address if the last value pushed on to the stack was positive (if viewed as a number in two's complement format).

Variants JmpP []

Arg. count 0 / 0

Size 5

Usage
```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpP 232    ; jump to address 232 if the value was positive
```

Jump if not positive

Jump to the specified address if the last value pushed on to the stack was not positive (if viewed as a number in two's complement format).

Variants JmpNP []

Arg. count 0 / 0

Size 5

Usage
```
PushLD 100  ; arg 1 (the double at memory location 100)
Dec
PopLD 100
JmpNP 642  ; jump to address 642 if the value was not positive
```

Call

Calls the function at the location specified (a quad). The address of the instruction immediately after the call is the next to be executed when the function returns.

Variants Call []

Arg. count 0 / 0

Size 5

Usage
```
PushCD 100  ; argument for the function to call
Call 345    ; call function at address 345
...
PushGQ 10   ; origin 100
Dec
PopGQ 10
Ret         ; and return
```

Return

Restores the stack to the state prior to calling the current function and continues program execution at the instruction following the call instruction to the current function.

Variants Ret []

Arg. count 0 / 0

Size 1

Usage
```
PushCD 100  ; push return value
Ret         ; return from function
```

IO

Output

Transfer a value to an output port.

Variants Out []

Arg. count 2 / 0

Size 1

Usage
```
PushLS 100  ; the value to put on the port
PushS 110   ; the port number
Out         ; output it
```

Input

Read the value of an input port.

Variants In []

Arg. count 1 / 1

Size 1

Usage
```
PushS 110  ; the port number
In         ; read it
```

Misc

Wait

Wait the specified number of milliseconds (a double) before proceeding to the next instruction.

Variants Wait []

Arg. count 1 / 0

Size 1

Usage
```
PushCD 10  ; the number of milliseconds
Wait       ; wait
```

Building the VX core

Compile time settings

A number of settings can be adjusted before the VX VM firmware is compiled and programmed into the target controller. Using compile time configuration of the core firmware any program storage media can be used (internal EEPROM or flash, external flash card, USB port using a PC as virtual media to name a few).

The operating frequency of the processor must be configured allowing the firmware to adapt to any system speed. This affects the internal timers, the wait instruction, communication baud rates etc.

The VX tool chain

The VX platform is supported by a dedicated tool chain. All programs are made specifically for the VX VM.

The tools contained in the tool chain is

- **VXA.** The VX assembler. Generates .part files from .vxa source files.
- **VXL.** The VX linker. Links one or more .part files to produce an executable .vxx file.
- **VXC.** The VX compiler. A compiler for a C inspired high level programming language.
- **FSIC.** File Store Image Creator. Converts directories on a PC to image files (.vfi) for the file system used on the VX platform.

The VX assembler

The VX assembler (VXA) is a command line tool capable of generating VX executables.

Only one instance of section, label or mnemonic can appear in a line.

Sections

The assembler accepts three sections

- Code
- Stack
- Data

Labels

Labels bind a human readable textual name to a memory address.

Labels can be used in all sections.

Comments

Comments in the assembler source file is allowed by the assembler by removing any instances of ; (semicolon) and the rest of the line on which they were found.

Command line options

Options are passed to the assembler when executing it.

The following lists the options available

- **File.** –f. The file to assemble. This option must be included.
- **Output name.** –o. Using the –o option an alternative name can be given to the generated .part file.
- **Generate list file.** –l. If this option is used a list file is generated displaying details on the generated object file.
- **Generate preprocessor file.** –p. Generate a file containing the output from the preprocessor. The output has the extension .pre.
- **Generate map file.** –m. Generate map file when processing the .vxa file.

Assembling a program

To assemble a file using the default options simply call the assembler with the –f option as shown below:

```
vxa -f hello.asm
```

This will, if the source file is valid, produce a file named hello.part.

The File Store Image Creator

The internal microcontroller EEPROM is used for a tiny file system. Due to the extremely small size a read-only file system is used. Since files then cannot be downloaded or created individually a binary image must be build on a PC and then downloaded. The files can then be read and loaded by the system.

VX files

The VX system operates with a number of files. These are all described below. Some might be mostly self explanatory and so their description will be short.

Executables

Virtual eXecuter program files with the extension .vxx (Virtual eXecuter Executable) contain the assembled program code, constants and any auxiliary program information that may be needed to load the program.

The file is divided into two sections

- Load info
- Binary data

Load info

This section contains a number of fields that are used by the program loader to determine how to load the program and if it is at all possible to load and run it.

- **VX file type tag.** A valid VX executable file starts with a tag made up of the five ASCII letters “VXEXE”. The file extension is only to make it more human-friendly.
  This is the only field in the load section that is human readable.
- **Flags.** 32 additional flags. Currently non are used and these should all be ‘0’.
- **Code size.** This is the size of the program code that will be executed. This is a 32 bit integer.

- **Data size.** This is the size of the data section. This is a 32 bit integer.
- **Stack size.** This is the size of the stack section. This is a 32 bit integer.

The load info section takes up exactly 21 bytes.

If the tag is not correctly identified or if the executable version is not supported or if the total memory requirement exceeds the amount of free RAM the program is not loaded and an error message will be produced on the default communication channel.

Binary data

The binary data section contains an image of what must be copied to memory i.e. the code and const sections. This section will be copied directly to the memory.

Assembler source files

These are the source code of programs.

Assembler source files have the extension .vxa.

List files

A list file may be generated when assembling a source file to inspect the generated binary code.

List files have the extension .list.

Map files

A map file may be generated when assembling a source file to inspect the mapping of labels in memory.

Map files have the extension .map.

Preprocessor files

The output of the preprocessor (first step in the assembling processor) may be output to a file.

Preprocessor files have the extension .pre.

Terminal interface

The main interface to a system running the VX VM is through the terminal interface. This terminal provides a simple interface like that of DOS or similar. It enables a user to list and execute files and to upload new disc images. As only the read-only version of the file system is implemented modifications or upload of individual files is not possible.

The file system

VX at first only support the VX File Store file system. This simple but ultra compact file system is read-only which in turn requires a disc image to be prepared on a PC using the VX File Store Image Creator (FSIC) application and then downloaded to the storage media. This download procedure can be done either trough a dedicated microcontroller programming tool or through the terminal interface.

Discs are mapped to single letters (again think DOS). The letter ‘A’ is reserved for internal microcontroller EEPROM (even if no EEPROM is present). Additional drives will be assigned letters in a yet unspecified manner but will depend on which disc interfaces is installed in the core.

What’s next

Compressed executables

In the future compressed executables may be implemented. These should have the VX file type tag “VXCEX” and will be decompressed when loaded. The three size values in the load info section will represent the in-memory size requirements.

FAT16 support

Support for the FAT16 file system must be added ASAP to enable SD memory cards written by a PC to be used as offline storage in VX systems.

More instructions

Some instructions have been left out in this first version of the VX VM. These can all be implemented using the available instructions but would be beneficial to have.

In the future the following instructions might be added

- Modulo. Calculate modulo of two numbers.

Terminal command summary

cd	Change directory. Changes to the specified path or emits an error if path is unavailable. As in DOS '..' denotes the parent directory. Directories are not yet implemented!
<disc>:	Changes to the disc with the letter specified or emits an error if disc is not available. Multiple drives are not yet implemented!
list	Lists files and directories in current directory.
print <file>	Prints the contents of the file specified or an error message if the file was unavailable.
<file>	Loads and executes the specified file if it is a valid VX executable file. Otherwise an error message is printed.
load <disc> <size>	Loads a disc image to the disc specified through the terminal interface. The terminal interface channel is assumed to be reliable and so the image must be sent "as is" with no encoding. The size field is used to determine when the entire image has been transferred and to verify that the image fits on the disc. To abort transfer after the command has been executed (but before the actual file transfer has commenced) press the ESC key.