Written and Developed by Diogo JC Duarte

Hex Vex Assembly Reference

xasm assembly type 1

Version 1 - 10/04/2021

About

HexVex is a simple and primitive computer architecture. Not a true 16 bit system, yet capable of handling 16 bit numbers through special mechanisms if so needed.

It is designed to be used with the same 16 bit address × 8 bit data ROMs for program memory and decoders.

It doesn't use a RAM, but one might be used in place of the ROM for program memory, if so preferred.

To Do:

- Add the block diagram to this document.
- Emulator instructions section.

Terminology notes

Some terms used in this document are used differently from what might be usually expected for other computer architectures.

Ticks	1 Tick is one clock cycle period.
Cycles	1 Cycle is the number of clock cycles to execute an instruction.
Program Memory	The device where instructions are loaded from.
Store	How it's referred to the pair composed of the Main Bus and Auxiliary Bus. Together they can transport a 16 bit number or two 8bit numbers being used independently.
Cache	Two internal 8bit registers, each overwatching each Store bus. They can be used as intermediaries when transposing numbers to multiple registers.
Pointer	Address given to the program memory.
Write Address	Mentions to this refer relatively to the Store. IE: The Store writes the device.
Read Address	Mentions to this refer relatively to the Store. IE: The Store reads the device.
LSB	Least Significant Byte
MSB	Most Significant Byte

Conditions to test against

Every instruction is started with a conditional flag, which is used by the computer to check against some flag of the internal state register. If the flag is matched, the instruction is accepted and executed. The internal state register is altered by ALU modules wired accordingly.

Mnemonic	Hex	Description
_	0x00	This instruction will always be accepted in ANY case
:	0x01	CARRY flag is raised (since last time the flag was set)
+	0x02	Bool is TRUE (since last time the flag was set)
-	0x03	Bool is FALSE (since last time the flag was set)
!	0x04	Numbers are DIFFERENT (since last comparison)
>	0x05	Number is GREATER than Accumulator value (since last comparison)
<	0x06	Number is LESSER than Accumulator value (since last comparison)
=	0x07	Numbers are EQUAL (since last comparison)

Assembler Specific Symbols

When programming by writing for the assembler, other symbols may be in place of conditional flags, but don't make functional instructions. They convey meaning only to the assembler. Check example programs to learn how to use them.

#	A comment. The line starting with this symbol will be completely ignored.
|	An import. The assembler will read a program from another file and place it in the line this symbol appears. The text after this symbol is the relative file path.
*	A variable declaration. If an instruction argument isn't a number, the assembler will look for lines with this symbol for a match and use the associated number in place.
$	Specifies some parameter about the program. Most used to build manifests of the hardware configuration of the system (Ie. what modules the program expects).

The Case of «`[label]`»

Strings enclosed in square brackets mark a position which pointer can be referenced as a variable. This is similar to "labels" for jumping in other programming languages, and enables a mechanism for subroutines. You can think of labels as variables which value is automatically determined depending on the position they are declared in the program. Check example programs to learn how to use them.

Operation Codes

Mnemonic	Hex	Arguments		Description
nhalt	0x00	16 bit int		Stop executing. If argument > 0x0, then just sleep for n-16b ticks.
rhalt	0x01	address		Stop executing. Take sleep duration from a register
chalt	0x02	None		Stop executing. Take sleep duration from Cache
nope	0x03	None		Does nothing. Useful to negate the Accumulator 16b mechanism
end	0x04	None		Marks the end of an interrupt subroutine, making it possible to jump to the next subroutine in the interrupt stack
tell	0x05	8bit int	address	Sends a set-up pattern to modify the behaviour of a register, if possible.
ngoto	0x06	16 bit int		Jump to pointer given
rgoto	0x07	address		Jump to pointer taken from register
cgoto	0x08	None		Jump to pointer stored in Cache
rept	0x09	16 bit int		Repeat pointer of current instruction, plus some offset
gobak	0x0a	16 bit int		Return to the pointer where the last jump was performed, plus some offset from the arguments (an earlier 'gobak' doesn't count)
cont	0x0b	16 bit int		Return to the pointer that was left off since last interruption, plus some offset from the arguments. Also acts like the "end" code
nmove	0x0c	8 bit int	address	Write given integer to the register with given address. Will make Accumulator's LSB available in the Auxiliary Bus
rmove	0x0d	address	address	Copy value of one register to another. Will make Accumulator's LSB available in the Auxiliary Bus
cmove	0x0e	address		Copy 16bit value from the Cache to a register (if possible)
xfeed	0x0f	address	address	Swap the values between two registers.
accfet	0x10	None		Copy value in the Accumulator to the Cache
hexfet	0x11	16 bit int		Write given integer into the Cache
utfet	0x12	None		Copy value in the User Timer into the Cache
fedfet	0x13	None		Swap values between Accumulator and Cache
!t	0x15	None		Toggle the state of the User Timer
t>n	0x16	16 bit int		Start counting from the given integer
t>r	0x17	address		Start counting from the value from a given register
t>c	0x18	None		Start counting from the value stored in the cache
t+n	0x19	16 bit int		Load User Timer with given integer, but keep state
t+r	0x1a	address		Load User timer with value from given register, but keep state
t+c	0x1b	None		Load User Timer with value from cache, but keep state
npwm	0x1c	8 bit int	8 bit int	Set high and low period of PWM from a literal value
cpwm	0x1d	None		Set high and low period of PWM from value in the Cache

The conditional flag leads the opcode, which tells the action to perform with the arguments. One of the letters of a mnemonic might suggest how the system will be interpreting the arguments.

n	number is a LITERAL INTEGER (8b or 16b depending on particular opcode)
r	number is a REGISTER ADDRESS
c	number is taken from the internal CACHE (always 16b)

PWM means “Pulse Width Modulation” and it generates a squarewave signal of programmable duty cycle at its dedicated pin. The duty cycle can be configured by writing the duration (in ticks) of each state of the signal with “pwm” opcodes. While generating PWM signal, the User Timer will be counting such as it can control the that signal so it might interfere with other timer operations.

Assembling Instructions

The structure of each instruction is like:

```
CONDITIONAL OPCODE ARGUMENT (ARGUMENT)
```

For example:

```
+ nmove 2 10
```

Meaning that, if the boolean flag is True, move the number “2” to the register in the write address “10”. The example shows the arguments in decimal notation, but it can be mixed with binary or hexadecimal also. Some opcodes have 2 8-bit arguments, others have a single 16-bit argument, but there’s also opcodes that ignore the value set for the arguments if any is supplied. In absence of an expected argument a value of zero will be implied. Check the table in «**Operation Codes**» to know how the different opcodes interpret the arguments given. The available conditional codes can be found in «**Conditions to test against**».

Every instruction is composed of 4 addresses in the program memory. The first address to be read by the system is a byte for control and it's composed of the conditional flag number (3 bits) concatenated to the opcode number (5 bits).

Example:

```
+nmove → 2_10 → 0b010_0b10000 → 0b01010000
```

The second and third bytes are arguments to the opcode. Whatever they mean depends on a particular opcode.

The instruction is executed when the 4th byte is pointed to, but the value of that byte isn't processed by the computer. It can potentially be used to hold metadata.

The following example instruction will move the number 0x42 (decimal 66) to a register in the location 0xa4 if, and only if, the Boolean flag is positive:

```
+nmove 0x42 0xa4 → 01010000 01000010 11110001 00000000
```

Registers & Arithmetic & Logic Functions

Hex Vex is designed to be modular, with minimal internal registers. The registers required to complete a given task can be connected to the exposed Store and Address busses. The address to which each responds to is completely set by the programmer by some physical mechanism in the module, like DIP switches. Internal registers are either addressed by direct wiring from the control unit or also DIP switches set up by the programmer. It's advised to import the addresses from «defaults.xasm» when in doubt.

Under normal operation, a 8-bit register is wired to receive and send data through the Main Bus. A combinational logic circuit is wired to the Address Bus and compares its signals to the register's set address. If there's a match, the register will read or write the Main Bus accordingly. Several registers may be written to at the same time but only one may be read. Hardware faults may result otherwise.

The Address Bus is 8-bits wide. 6 bits are the actual address, the 7th bit distinguishes between a reading or writing operation for the target register, and the most significant bit is an acknowledgement signal for modules which need to reply back to the system.

The ALU is composed of an internal Accumulator register which reads from its dedicated Bus. Much in the same way as registers, there are no internal ALU operations, so the ALU bus is exposed for the wiring of desirable ALU modules.

ALU modules are treated like a general purpose register and thus written to by the same addressing system. Each time something is written into an ALU module, it will pull the acknowledgement bit high through the Address Bus, along with the operation result in the ALU Bus, causing the Accumulator to record that result.

Note that whenever a 8bit value is written into the Main Bus, the system automatically loads the Accumulator's current value into the Auxiliary Bus, to be used by modules requiring two operands, as most ALU modules do.

ALU modules aren't intended to store their own value and as such they can't be read, so the Accumulator serves the common register for them. Some opcodes take the whole 16bit number from the Accumulator, but you can also supply addresses for individual halves of the Accumulator number.

Despite the ALU bus being 8bit wide, the "Upper Byte Mechanism" or "Upby" enables modules to access the full 16bits of the Accumulator. Whenever the Accumulator is triggered to read the ALU bus, it'll raise the Upby flag and drop it after the next cycle. During the cycle when the flag is raised, the accumulator assumes that its upper half of data is being computed, so any value incoming from the ALU bus is stored as the MSB of the Accumulator. To suppress this behaviour, if for example the programmer intends to perform two 8bit calculations in a row, they should have a «nope» opcode in between.

The Hex Cache and Timers

The Cache and User Timer work much like the Accumulator, in that each half of the hexadecimal digit they hold can be individually accessed.

The Cache allows for the mediation of 16-bit numbers with a single instruction, using any of the opcodes with “c” prefix or “fet”, for “fetching”. This includes «accfet» to read from the Accumulator, «hexfet» for direct writing into the Cache and «utfet» to read from the User Defined Timer. This potentially enables much faster processing for some algorithms. But each of the two Cache registers can still be accessed individually like normal peripheral registers.

The User Defined Timer is a register which automatically increments its value with the ticks. It can be read like a normal register, for each byte of its value, but it requires the use of timing opcodes to write to it. Once it overflows, it wraps back to zero.

The Sleep Timer can’t be accessed like any regular register. The halting opcodes will write a starting value for it, but then it automatically decrements. Upon reaching zero, it prompts HexVex to exit the halted state. This behaviour is used to make HexVex wait for a moment before running code. If the starting value is zero, it won’t decrement. This way the system is halted indefinitely

Internal State & Flags

The Carry flag is set at the discretion of ALU modules. Those which set it, should also reset it at every operation. That way, the carry signal is always truthful relative to the last performed ALU operation, of those that can cause a carry event.

When the Store is carrying a 16bit number instead of two 8bit numbers, the Auxiliary Bus will have the Most Significant Byte and the Main Bus is the Least Significant Byte. Of course, it’s still possible to fudge it while using «FET» operations in combination with juggling with each byte individually.

System Interruptions

It is possible to direct the system to different points of the program than what is sequenced by default by issuing hardware interruptions. At any time an external device can raise the Interrupt signal and insert a pointer to the Store. The system will pause whatever operation is going on and clear the Store for writing of the pointer to occur. This point is then loaded into a First-In-First-Out chain register in the program counter. The pointer will wait in the FIFO until the system is ready to load the next instruction.

Essential ALU modules

HexVex is designed to come without any ALU operation by default. It’d be to the discretion of the user to plug in the proper operations for the intended job, This is the spirit of having a very modular and “suckless” ALU. Of course a computer without ALU doesn’t make for a very useful one. The operations deemed as the most important are the following and should be default with a full-fledged HexVex configuration. A combination of these can solve any other logical and arithmetic problem given enough computing time.

NOT gate array

The most basic of boolean operations. It’s necessary to transform the behaviour of other logic gates.

XOR gate array

It’s an advanced logic gate that can solve many mathematical and decoding problems. Can be produced with a combination of other logic gates, but it’s relatively complicated.

It is very useful to do a selective negation too by XORing the input number with a number where only the bits intended for negating are «1». For example, the input XOR 0xFF is the same as the NOT module.

NOR gate array

One of the universal logic gates. The other being the NAND. This means it can be used to mimic any boolean operation with enough combinations of itself.

Bitshifting

This moves the digits of a binary number in significance or binary place. Shifting up is the same as multiplying by a multiple of 2 and shifting down is dividing in the same way, thus it can be helpful for fast mathematical problems. This is the behaviour of shift registers and overflowing digits can be forgotten or made to wrap around, what is called “ring shifting”.

A bitshifter can be told to fill in the unknown values with a set default too, instead of filling with the values being wrapped around.

Equality Test

It changes the values of the internal flags to reflect the difference in magnitude of input numbers. This enables to check if two numbers are equal or one is greater or otherwise.

Negator

HexVex doesn’t distinguish negative integers explicitly, so to make a subtraction, for example, a value needs to be formatted accordingly. This module will implement two’s complement rule to its input, resulting in a number that can be interpreted as a negative.

Addition/Sum

It's the most basic arithmetic operation. It adds the number in the Accumulator with that of the input. It is the fundamental operation for calculus integration too.

To subtract is also to add but with negative numbers, if you are using the proper number format.

Multiplication

Better than adding many times the same number and dedicating a register to count the additions. A module which can do both addition and multiplication is feasible too. This is better than bitshifting because it allows an arbitrary multiplier to the input and not only multiples of 2.

Beginner's Quick Start

Up until now, this Manual has been a quick reference. Helpful to take facts about the HexVex assembly when you already know what you need to find. For new people first beginning to learn it, this section goes over the structure and thinking behind programming HexVex.

Programming is written on any other common computer capable of running the assembler program. As to time of writing, the assembler is a Python language script. A plain text file is written with the assembly code described in this manual and then the assembler is executed to produce binaries that can be loaded into a ROM or RAM read by HexVex as Program Memory.

Each line of assembly code translates into an instruction. The Opcode of that instruction tells what action to take with the values of the arguments. The instruction is only read if the conditional symbol matches the state of HexVex at the time the instruction is processed. You can read about this with more detail in «Assembling Instructions».

You have basically 4 categories of action: System miscellaneous stuff (like halting execution), jumping to instructions, moving a number (copying) to a register, and timer controls.

Various variants on each category will interpret the arguments differently. For example, when moving a number to some register, the source can be other register with `rmove`, in which case the first argument is the address of the source register.b On the other hand `nmove` will take the first argument literally as the number to be put in the target device. The last argument of these two will be the address of the target device. So, for example, `rmove 1 2` can be read as "move from register 1 to 2 " where «1» and «2» are the addresses of some two registers. Of course it's often cumbersome to have to remember the addresses all the time so we make a variable like `*reg_a = 1`, Thus the instruction becomes `rmove reg_a reg_b`. Literal numbers can also be made into a variable for easy reference. For example, the division example program has `*divisor = 4`. Just change the value in that variable definition and it automatically changes wherever else it is referenced.

In HexVex the ALU is also interpreted as a set of registers and you move numbers into the modules you want to perform a calculation. The result is automatically stored in the Accumulator register. The modules themselves, in principle, don't have a read address, as they won't store any number themselves. ALU operations which require 2 input operators, will have one of them be the value already stored in the Accumulator. For example, to perform addition of 2+3, if the number three is already in the Accumulator you do `nmove 2 adder`, so “move number 2 to the adder”. The result, “5”, will replace the value of 3 in the Accumulator. Operators with a single input, like the boolean NOT would take the 2 put it in the accumulator, replacing something already there. Because it's common to have to calculate on multiple values in the same programmer, you'll have to switch the value in the accumulator with some other register many times times. To help on this juggling of data, there's the special move operation `xfeed`, so you can swap value of the Accumulator and a target register in a single instruction.

The number comparison modules don't write to the Accumulator, they will just change the internal state flags instead.

Design Details and Tips

1 – From this page forward the document really needs more TLC. It’s still a work in progress.

2 – Under current iteration of the architecture there isn’t an ALU module which changes the boolean internal flags. The number comparator will write to all the 4 “comparison” flags at once instead. So the purpose of the Boolean conditional flags is to allow provisions for possible future features or streamlining. If you make your own ALU module, like a custom number comparator which writes to Bool instead, you might appreciate this provision.

3 - In a pursuit of reducing the number of instructions overall and keep the amount of instructions even, regardless of mathematical complexity, the ALU works like a mathematical integration function. For each ALU prompt, the Accumulator feeds its Least Significant Bits into the Auxiliary Bus and the where they are picked up by the ALU modules, along with whatever's present in the Main Bus. The result is then stored back into the Accumulator, thus effectively, the Accumulator is in a feedback loop and addresses for function modules pick the desired feedback network. At any moment, the accumulator value can be retrieved into the cache enabling multiple operation to be performed without interference. You can think of HexVex as a digital equivalent to an Operational Amplifier, given the right ALU modules and choice of instructions.

4 - Clearing the Accumulator can be performed by multiplying by zero. Loading a specific number can then be done by addition.

5 - The "nhalt" instruction with the "any" flag are chosen to result in the 0b0 code as a safety for the possibility of the computer to run into empty program memory lines, thus avoiding pointer overflow. So, in such situation, the computer will gracefully stop executing.

6 - Timer codes control the "User Timer" which is dedicated to be used by the programmer for whatever. You can count how many ticks it takes to execute an algorithm, for example.

7 - HexVex was designed to be relatively easy to remember the machine language codes and thus enable programming without an assembler. The choice of the binary numbers associated with each action is such as to enable heuristical/intuitive thinking about the instruction set.

You could pretend there are only 4 opcodes: Execution, Goto, Move and Timer. Thus, 2 of the 5 bits for opcode specify which of these: 0b00, 0b01, 0b10 and 0b11. The 3 bits left specify a modifier of the these four functions: 0b000, 0b001, 0b010, 0b011, 0b100, etc... if modifier is available. You could think as the 2 first bits as specifying which action to perform and the last 3 bits as specifying how to interpret the arguments. The conditional flags are similar: 0b000 it doesn't care about flags, 0b001 it kinda cares; it's usually rejected. 0b01x means we are testing the state of the Bool flag. 0b1xx is comparing two numbers. The Least Significant being the accumulator value and the next bit being the tested number. You could imagine the mathematical comparision symbols between these bits.

0b1-10 --> test number is greater, 0b1-01 --> accumulator is greater

0b1-11 --> means they are equal, thus 0b1-00 must mean they are different.

Under a similar train of thinking, each argument is a segregated byte and a simple number, thus not much to get wrong there.

8 - The microcode still needs to be written by machine. You are screwed if you have an HexVex computer with cleared ROMs. Any Microcode value not in use (usually due an illegal combination of instructions) will be loaded with the "Skip Instruction" pattern. It's conceivable that these values can be hacked for custom opcodes or features. The patterns are the following:

[Fill in another day; for now use the block diagram to get a hint.]

9 – The system interruption mechanism works by dedicating the last addresses of the Program Memory to `ngoto` opcodes which lead the system to whatever program pointer the relevant code is found. When the Interrupt Handling hardware is triggered, it causes a jump to one of the last addresses. The attribution between specific interrupt and associated program address is hardwired. But the pointer where these interrupts lead to, after running their `ngoto` instructions can be freely defined in software and they are interpreted no different from user defined labels.

10 – The standard GPIO devices when in output mode will behave pretty much like any plain register. When writing into them, the register stores the value. When reading, the value is taken from the register. When the GPIO is in Input mode, the writing will hopelessly be registered, while reading will be take whatever value comes from the external interface. At the same time, the internal register will be set to that external value too, so it’s possible to switch the device to output mode and the last externally read value will still be accessible. Otherwise register in input mode could be thought as a write-only device. It must also be noted the quirk that it’s possible to write to an input-mode device and if no read is done afterwards, the written number will still be present.

11 – The choice of setup pattern’s second bit to be «0» for output mode and «1» for input mode is deliberate, given that the symbol for these numbers looks similar to the letters “O” and “I”. The choice of «0» for the first bit to be the “toggle mode” action is so that you can either use `poke` or `tell` for the same effect. So the setup pattern doesn’t matter if you just want to toggle the mode.

12 – The `xfeed` operations assume that the target register has the same read and write address. If that isn’t the case, it will put contents of a register into the Accumulator and the contents of the Accumulator into yet another register.

History and Plans for HexVexNxt

Shenzhen I/O
Analogue I/O register.
Mechanism to synchronize with real-time seconds.
Self-programming facilities.
Type 2 assembly.
High level language.

Interesting Programs to try out

Division:

With the lack of floating point capability in HexVex, it isn't possible to have a number division ALU module. But you can emulate it in software. A basic version would calculate the integer quotient approximation and modulus. A more advanced version could calculate floating points by successive approximations.

Integration:

Try writing a program which requires mathematical integration. Like finding out the velocity of a pneumatic piston, predict the time constant of a capacitor filter, the ballistic trajectory of a missile or the velocity of an antipodal train travelling through the centre of the Earth.

Trigonometry:

Trigonometry is tri-cky! Maybe you can do a program to compute the sides of a triangle. Or better, the sine and cosine, which then you can output the result through analogue output and synthesize audio!

Fast Fourier Transform:

If audio synthesis isn't hard enough for you, maybe you'll want to take analogue input instead and make a program to find the frequency components of any signal.

Computer Virus:

A computer virus is a program that, much like a biological virus, uses the host (the program memory) to replicate itself, potentially spreading its own code to to other computers. If you add an EEPROM or a RAM as a peripheral device, could you make a program which writes the same code as itself to that?

Graphical Display:

It is possible to have a peripheral device which is a pixel matrix where each pixel can be selected by two registers representing the horizontal and vertical coordinates for that pixel. This kind of display could work much like a RAM where you can set the values of each row of data individually and they'd persist while other values are being modified. In such way a primitive display for visual output is possible. The fact that each register is 8 bits allows a maximum resolution of 256 by 256 pixels. But with extra trickery it's possible to have larger resolutions at the cost of extra computing cycles. Updating the display with the help of the Auxiliary Bus, using the Cache as intermediary bus management and `poke` for special writing, more complex displays or 16 bit (65536 pixels wide) resolutions!

Videogames:

If you achieved to plug in a display, you might then be able to design a game of Tetris, Snake, Asteroids, Missile Command, Solitaire or Space Invaders. Video games are both the most computationally demanding applications and the hardest to code. They employ trignometric operations, timer, graphics and audio, all in tandem. This is a real challenge of your skills!

Hardware for extra power

Better Multiplication:

Algorithms to calculate multiplication of two numbers abound. Maybe you can make a better multiplication circuit from the default one. Better, faster, harder, stronger.

Differential Engine:

One of the first and still most astounding computing devices was Charles Babbage's Difference Engine. It was used to find the coefficients of polynomials by the method of differences and approximations he devised for use with only addition operations. Theoretically, this mathematical process can be used to approximate any arithmetical equation to an arbitrary degree. HexVex with a hardware implementation of the algorithm of this machine wouldn't need any other ALU module!

Numerical Decoders:

What if one of the devices connected to HexVex is a 7 segment display? To show the numbers properly you'd need a binary-to-7-segment conversion circuits for each digit... There must be a be a better way! An ALU module which takes a binary number and returns the display pattern is exactly the right thing.

Other types of decoding or encoding devices can also be useful. Some sensors send Gray Code, which must be converted to normal binary representation before being used as input for maths, for example.

You could also develop your own code system for cryptography purposes. Cryptography is much faster and safer in dedicated hardware implementations!

Random Number Generators:

Talking about cryptography, a lot of it requires a random number to be unpredictable, thus truly safe. Videogames and simulation software also make a lot of use of random numbers. A computer by itself, using only basic arithmetic can't generate true randomness in software only.

Audio Card:

If you think about it, audio synthesis through software is kinda lame. You can build module to which HexVex just has to set a frequency and duration, then it automatically plays a note.

Table of Contents